# Development of Web-Based Educational Modules for Developing VHDL Models of Digital Systems

By

Weihong Song

Thesis submitted to the faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Electrical Engineering

**APPROVED:**

Dr. F.G.Gray, Chairman

Dr. J.R.Armstrong

Dr. D.S.Ha

August, 1997

Blacksburg, Virginia

**Keywords**: VHDL, Education, Modeling, Hardware Design.

# Development of Web-Based Education Modules for

# Developing VHDL Model of Digital System

By

Weihong Song

Dr. F.G.Gray, Chairman

Electrical Engineering

(ABSTRACT)

Hardware description languages (HDLs) such as VHDL have made it possible for circuit and board designs to be done without resorting to paper, allowing computers to manage the design database and automate the translation between various representations of the system.

Although VHDL modeling can provide a bonanza of benefits, VHDL models must be used effectively to reduce overall development costs. Air Force acquisition specialist for such agencies as the U.S. Air Force must be able to manage the development of VHDL models to ensure that the models delivered meet both their immediate and their future needs.

In this research study, effort has been made to explain the many facets and advantages of VHDL and the use of VHDL in various stages of the acquisition process. The RASSP design concepts, such as top-down design, reuse and the model year concepts and integrated design environment, are introduced. Different abstraction levels from written specification to gate level in the design process are explained. Design techniques, such as modular design, reuse and sharing of designs, multiple abstraction level simulation and mixed data type simulation, and design trade-offs, are illustrated. The Sobel edge detection system is used as our case study to illustrate multiple abstraction levels and their use in the acquisition process. The methodology of developing an easy access, low-cost and effective VHDL education and guidance materials for Air Force acquisition and maintenance specialists is explained. The primary focus of this thesis is to develop a World Wide Web (WWW) based educational module for the acquisition process.

*To my parents and husband*

# Acknowledgments

Numerous people are responsible for the success of this research work. I would like to take this opportunity to thank them. My sincere appreciation goes to my advisor Dr. F.G.Gray for his unerring guidance and support throughout my thesis research. I would like to thank Dr. J.R.Armstrong, for his valuable ideas and suggestions and Dr. D.S.Ha for serving on my graduate committee.

Special thanks to Dr. Geoff Frank and Mr. Bud Clark of Research Triangle Institute for their numerous ideas and suggestions.

I would like to dedicate this work to my parents who have always encouraged me at all times. Without their constant love and support, I could not have got this far in my career. I also own much to the inspiration and encouragement I received from my husband, Yuwen. His unbounded love and patience have meant more to me than anything else. I specially wish to thank my sister, Weijing, and my sister-in-law, Baolin, for always giving me the right advice and help.

Finally, thanks are due to my research project partner, Sucharita Gopalakrishnan, for her constant assistance and support in research and other matters. Also, I would like to thank Mohammed Osman and Meng-Wei Lin for their valuable suggestions and help throughout this past year.

# TABLE OF CONTENTS

**CHAPTER 1. INTRODUCTION.....1**

1.1 Motivation.....1
1.2 Task Description.....2
1.3 Contributions.....3
1.4 Thesis Organization.....4

**CHAPTER 2. BACKGROUND.....5**

2.1 About VHDL.....5
2.2 Basic VHDL Concepts.....6
2.2.1 Design Entity.....6
2.2.2 Behavioral Modeling.....7
2.2.3 Structural Modeling.....9
2.3 About the RASSP Design Process.....10
2.3.1 Top-Down Design.....10
2.3.2 Reuse and the Model Year Concept.....11
2.3.3 An Integrated Design Environment.....11
2.4 About the Sobel Edge Detection System.....11

**CHAPTER 3. MULTIPLE LEVELS OF ABSTRACTION IN AN ACQUISITION PROCESS.....16**

3.1 Abstraction Hierarchy.....16
3.2 Top-Down Design and Bottom-Up Design.....19
3.3 A Case Study of "Top-Down" Design.....21
3.3.1 Written Specification.....23
3.3.2 Requirements Repository.....23
3.3.2.1 About SGE Design Tool.....23
3.3.2.2 Graphical Capture of the Sobel Edge Detector.....23
3.3.3 System Level Executable Specification.....27
3.3.3.1 Library Declaration.....31
3.3.3.2 Packages Used in the Entity EDGE_DETECTOR in BEH_INT Library.....32
3.3.3.3 The Behavioral Description of the Sobel Edge Detection

System....38
3.3.4 System Decomposition....42
3.3.5 RTL (Register Transfer Level)....46
3.3.6 Gate Level Synthesis....52

**CHAPTER 4. DESIGN TECHNIQUES....55**

4.1 Introduction....55
4.2 Modular Design....55
4.3 Maintaining, Sharing and Reuse of a Design....57
4.3.1 Manage the VHDL code and Make Use of Package....58
4.3.2 Sharing and Reusing an Existing Design....58
4.3.2.1 Library....59
4.3.2.2 Generics....61
4.3.2.3 Coding Style....62
4.3.2.4 Configuration Body....67
4.4 Multiple Level and Mixed Data Type Simulation....69
4.4.1 Multiple Level Simulation....69
4.4.2 Mixed Data Type Simulation....71
4.5 Design Trade-off....74
4.5.1 Introduction....74
4.5.2 Trade-offs Among ASIC Designs and COTS Designs....75
4.5.3 Trade-offs Among Optimized Gate-Level Circuits....76
4.5.4. Case Study: Making Trade-Offs for the Window Processor....76

**CHAPTER 5. DISTANCE TEACHING OVER THE INTERNET....80**

5.1 Introduction....80
5.2 Goal of the Computer Based Educational (CBE) Module....81
5.3 Developing the CBE module Over the Internet....81
5.3.1 Organizing the Teaching Materials....82
5.3.2 Developing the HTML files....82
5.3.3 Web Page Design Techniques....83
5.3.3.1 Making a Comfortable Learning Environment....83
5.3.3.2 Making an Interesting Learning Environment....83
5.3.3.3 Providing Step by Step Explanations....83
5.3.3.4 Providing a Complete Tutorial for Design, Testing, and Results....85
5.3.3.5 Making Use of Frame Sets....85
5.3.3.6 Making a Short HTML for Each Page....85
5.3.3.7 Making Links....87
5.3.3.8 Showing Testing Data in Image.... 87
5.3.4 Developing a Web Site for the CBE Module....89

5.3.4.1 Making a “Table of Contents” (TOC) Page.....89
5.3.4.2 Implementation of the *Executable Specification* Lesson.....91
5.3.4.3 Implementation of the *Library* Lesson.....97
**CHAPTER 6. CONCULSION AND FUTURE WORK.....100**

**REFERENCES.....102**

**VITA.....103**

# LIST OF ILLUSTRATIONS

Figure 2.1: Interface declaration of a horizontal filter.....6
Figure 2.2: An architecture body of a horizontal filter.....6
Figure 2.3: An algorithmic model of a 1-bit full adder.....7
Figure 2.4: A data flow description of a 1-bit full adder.....8
Figure 2.5: A structural model for a twelve-bit full adder.....9
Figure 2.6: Image processing system application.....12
Figure 2.7: The labeling of neighborhood pixels used to explain the Sobel operator.....12
Figure 2.8: The Sobel operators.....13
Figure 2.9: An example of horizontal filtering.....14
Figure 2.10: Direction Assignment.....14
Figure 3.1: The abstraction hierarchy.....17
Figure 3.2: An example of flat design.....18
Figure 3.3: An example of hierarchical design.....18
Figure 3.4: System decomposition of the Sobel edge detection system.....19
Figure 3.5: Cost per system vs. production quantity.....20
Figure 3.6: Structural decomposition [4].....21
Figure 3.7: Flow chart for the top-down design process.....22
Figure 3.8: Block diagram of the Sobel edge detection system.....24
Figure 3.9: Attributes for pin CLOCK.....24
Figure 3.10: Symbol attributes of the Sobel edge detection system.....25
Figure 3.11: VHDL code of the edge detector (EDGE_DETECTOR) generated by the SGE tool.....26
Figure 3.12: VHDL code for the executable specification of the entity EDGE_DETECTOR.....27
Figure 3.13: IMAGE_PROCESSING package declaration.....32
Figure 3.14: IMAGE_PROCESSING package body declaration.....34
Figure 3.15: Updating the window buffer.....39
Figure 3.16: Input image of the M1A1 tank.....41
Figure 3.17: Edge detected output image for the M1A1 tank.....42
Figure 3.18: Decomposition of the Sobel edge detection system.....43
Figure 3.19: VHDL code for the structural model for the Sobel edge detection system.....43
Figure 3.20: VHDL code for the behavioral model of the horizontal filter.....46
Figure 3.21: VHDL code for the WEIGHT function.....47
Figure 3.22: Functional block diagram of the horizontal filter.....48
Figure 3.23: SGE schematic for the RTL horizontal filter.....49
Figure 3.24: RTL model of the horizontal filter.....49
Figure 3.25: Configuration file for the RTL horizontal filter.....50
Figure 3.26: Gate level circuit of the horizontal filter.....53

Figure 4.1: Interface declaration of the memory chip.....57
Figure 4.2: VHDL design library concept [9].....60
Figure 4.3: A portion of the *.synopsys_vss.setup* file.....60
Figure 4.4: Changing the design library.....61
Figure 4.5: VHDL code for a 2-input OR gate using actual lue for the delay time.....62
Figure 4.6: VHDL code for a 2-input OR gate using a generic for the delay time.....62
Figure 4.7: Interface declaration of the structural model of the memory processor.....63
Figure 4.8: Interface declaration of the behavioral model of the memory processor.....63
Figure 4.9: Component declaration of the memory processor corresponding to Figure 4.8.....64
Figure 4.10: A general component declaration for the memory processor corresponding to Figure 4.8.....64
Figure 4.11: The structural model of the Sobel Edge Detector.....65
Figure 4.12: The configuration file for the structural model shown in Figure 4.11.....67
Figure 4.13: Structural model for the window processor.....69
Figure 4.14: An application of multiple level simulation.....70
Figure 4.15: The data conversion function: INTEGER  8-bit STD_LOGIC_VECTOR.....72
Figure 4.16: Associate the data conversion functions with the port maps.....73
Figure 4.17: SGE diagram of the window processor.....76
Figure 5.1: Partial web page of the requirement repository page.....84
Figure 5.2: Using frame sets to improve readability: system decomposition web page.....86
Figure 5.3: Making use of images: testing result page.....88
Figure 5.4: “Table of Content” page.....90
Figure 5.5: The executable specification lesson: introduction  (page 1).....92
Figure 5.6: The executable specification lesson: case study  (page2).....93
Figure 5.7: The executable specification lesson: case study  (page3).....94
Figure 5.8: The executable specification lesson: testing image (page 4).....95
Figure 5.9: The executable specification lesson: testing image (page 5).....96
Figure 5.10: The web page for the *library* lesson.....98
Figure 5.11: The web page for the issue of changing a design library.....99

# TABLES

Table 3.1: Optimization results of the horizontal filter.....54
Table 4.1: Optimization results of the horizontal filter.....77
Table 4.2: Optimization results of the vertical filter.....77
Table 4.3: Optimization results of the diagonal filter.....78
Table 4.4: Circuits with different constraints.....78

# Chapter 1. Introduction

## 1.1 Motivation

The computer-aided engineering revolution of the last 20 years has made tremendous strides in reducing the time to develop electronic systems and components and in increasing the complexity of their functions while at the same time reducing their size, power, and weight. Hardware description languages (HDLs) such as VHDL have made it possible for circuit and board designs to be done without resorting to paper, allowing computers to manage the design database and automate the translation between various representations of the system.

VHDL was developed to provide a standardized and technology-independent way to describe formally the behavior and structure of digital electronic systems. It offers the technical means to provide functional, timing, and other specifications for digital electronic systems in a form that will be useful long after the original system is delivered. Technology independence permits the separation of the behavior function (plus timing) from its implementation, which makes incorporating new technologies easier.

VHDL descriptions specify exactly what functions a new device would have to perform and the timing information associated with it. Through simulation of these descriptions, the design of a new device can be accurately modeled before being physically verified. Also, it allows the detailed structure of a design to be synthesized from a more abstract specification, allowing designers to concentrate on more strategic design decisions and reducing time to market.

In this age of sophistication and emphasis on shorter design time, it is imperative that VHDL and related standards be introduced to acquisition and maintenance specialists such as those that work for the U.S. Air Force, to enable them to appreciate the many facets and advantages of this universal language. An attempt has been made to increase the capability of acquisition specialists to negotiate and supervise the development of VHDL models and the capability of maintenance specialists to use VHDL for the synthesis of replacement parts and for test bench and test data generation.

An emphasis of this thesis is on the advanced paperless acquisition process.

Although VHDL modeling can provide a bonanza of benefits, VHDL models must be used effectively to reduce overall development costs. Acquisition specialists need to be smart buyers of VHDL models. They need to acquire VHDL models with sufficient data to make sound evaluations and decisions without being overwhelmed by unnecessary details. If VHDL is not used carefully, the models it generates are useless or dangerously misleading. Air Force acquisition specialists must be able to manage the development of VHDL models to ensure that the models that are delivered meet both their immediate and their future needs.

The Sobel edge detection system is used as our case study to illustrate multiple abstraction levels and their use in the acquisition process. An executable specification at the system level is developed first from the written specification. This is a high-level model which allows representing the system early in the design process where trade-offs can most efficiently be made. Then the system is decomposed into subsystems which reflect the physical organization of the system. This decomposition is applied until the subsystems are at the RTL (Register Transfer Level). An automatic synthesis tool is then used to generate the gate level model to be fabricated. Design issues such as how to effectively reuse a design, how to share a design, how to compare alternative designs and how to make trade-offs are discussed. The Synopsys simulation and synthesis packages are used in this project.

### 1.2 Task Description

This research work was done as part of the VITAMINS (VHDL Interactive Training for Acquisition and Maintenance Specialists) project funded by Research Triangle Institute (RTI). The objective is to develop easy access, low-cost and effective VHDL education and guidance materials for the Air Force acquisition and maintenance specialists. The aim of the VITAMINS team is to develop low-cost and modular learning experiences for the defense industry, which leverages the VHDL experience of RTI and Virginia Tech, leading-edge technologies such as RASSP, and advanced instructional methodologies.

Interactive training modules describing the use of VHDL to support acquisition and maintenance are made available over the Internet. These Computer-Based Educational (CBE) modules provide a hyperlinked navigational tool for accessing information about VHDL models, the RASSP design process, related standards and other useful resources.

This thesis discusses the development of an educational module for the acquisition process. This module presents the RASSP design process in terms of what VHDL models are developed, when in the process they are needed, and how they are used to validate the design as it proceeds.

## 1.3 Contributions

The development of the educational module for the acquisition process is the main topic of this thesis, including the following specific items.

♦ **Explained the importance of VHDL in the acquisition process**

The importance of VHDL in the acquisition process is discussed. An attempt has been made to highlight the many advantages of the use of VHDL in order to increase the awareness of VHDL and related standards among the Air Force acquisition specialists.

♦ **Integrated RASSP design technique into the acquisition process**

The Rapid Prototyping of Application Specific Signal Processors (RASSP) design process is introduced. Top-down design, the reuse and model year concepts, and the integrated design environment are explained. The Sobel edge detection system is used to as a case study to illustrate these concepts.

♦ **Developed an abstraction hierarchy suitable for the acquisition process**

An abstraction hierarchy, corresponding to the steps in the design process, is developed by modifying and extending published hierarchies. An attempt is made to help Air Force acquisition specialists understand that a particular design activity carried out at a specific level should have sufficient but not excessive detail. The following activities are related to the abstractions levels used in this thesis:

- Requirements repository
- Executable specification development
- System level architecture design
- Register transfer level model development
- Gate level model development
- Developed methodology for using VHDL models in a top-down design process

♦ **Modified and extended existing modeling techniques for use in the acquisition process**

The advantages of sharing and reuse of a design are presented and some techniques that help to shorten the design cycle and save design cost are explained. The Sobel edge detection system is used as a case study.

- Employed the concept of modular design
- Adopted the concepts of design library, packages, generics and configurations to the acquisition process. Reuse was a major emphasis.
- Developed an environment for multiple abstraction level simulation and mixed data type simulation
- Developed a systematic way to compare alternative designs and to make trade-offs

- **Developed a Computer Based Educational (CBE) module on a web site**

A hyperlinked navigational tool for accessing information about the educational module, RASSP design process, the VHDL home page and other useful material is provided. A user can easily access the necessary information and adapt them to his/her own design environment.

### 1.4 Thesis Organization

**Chapter 2.** "Background " gives an introduction to the concept of VHDL modeling. The RASSP design process is briefly described. Finally, the Sobel edge detection system, which is used as a case study in our educational module, is explained.

**Chapter 3.** "Multiple Levels of Abstraction in an Acquisition Process" describes the steps in the acquisition process and explains the benefits of VHDL models in each step. The Sobel edge detection system is used as a case study.

**Chapter 4.** "Design Techniques" discusses some modeling techniques that improve design maintenance, sharing and reuse and how they help to reduce the design time and cost. The concepts of VHDL libraries, packages, generics, and configurations and their use in the design process are introduced. The multiple abstraction level simulation and mixed data type simulation are explained. Also, comparison is made between the COTS (Commercial off-the-Shelf) design and ASIC (Application Specification Integrated Circuit) designs. The Sobel edge detection system is used as case study.

**Chapter 5.** "Distance Teaching Over the Internet" presents the methodology for construction of the educational material and the techniques for remote site teaching over the Internet.

**Chapter 6**. "Conclusions and Future Work" is the concluding chapter of this thesis and it highlights the contribution of this thesis to the acquisition process. Also, suggestions are made for further refinements and improvement.

# Chapter 2. Background

## 2.1 About VHDL

VHDL is a language for describing digital electronic systems. It arose out of the United States government's Very High Speed Integrated Circuits (VHSIC) program. There was a need for a standard language for describing the structure and function of integrated circuits (ICs). Hence the VHSIC Hardware Description Language (VHDL) was developed [1].

VHDL is a formal notation intended for use in all phases of electronic system design [2]. Because it is both machine readable and human readable, it supports the development, verification, synthesis and testing of hardware designs, the communication of hardware design data, and the maintenance, modification, and procurement of hardware [3].

VHDL is intended to cover every step of a design cycle from system specification to netlist. In the early stage of a design process, a behavioral model is developed, which describes the function, timing, and other aspects of the intended design. Then the top level system is decomposed into subsystems. A structural model is used to describe the decomposition and the connection between subsystems. This decomposition progresses until the design reaches the Register Transfer Level. At this level, a transition to the gate level is made. Thus there are typically three phases of a design process which use VHDL. The specification phase, the design phase and the gate level synthesis phase. System modeling is done during the specification phase while register-transfer modeling is done during the design phase. The netlist from synthesis of the RTL (Register Transfer Level) model is obtained in the gate level synthesis phase.

At each level, either a behavioral or a structural model, or both can be developed according to the design needs.

## 2.2 Basic VHDL Concepts

### 2.2.1 Design Entity

In VHDL a given logic circuit is represented as a design entity. The entity can be as complicated as a microprocessor system or as simple as an AND gate [4]. An entity consists of two parts, namely, an interface description and one or more architectural bodies.

Figure 2.1 shows the interface declaration of a horizontal filter. Ports are used to declare the input and output signals of the entity, where the names of the signals and their modes, types and initial values (if any) are specified. A given entity declaration may be shared by many design architectures. Thus, an entity declaration can potentially represent a class of design architectures, each with the same interface.

```
entity HORIZONTAL_FILTER is
        generic(HORIZ_DELAY:TIME);
        port(CLOCK: in  STD_LOGIC:='0';
             P1,P3:   in  PIXEL:=0;
             H:       out FILTER_OUT:=0);
end HORIZONTAL_FILTER;
```

**Figure 2.1** Interface declaration of a horizontal filter

In this example, we declare that CLOCK is an input signal whose type is STD_LOGIC and whose initial value is '0'; P1 and P3 are input signals whose types are PIXEL and whose initial values are 0; H is an output signal whose type is FILTER_OUT and whose initial value is 0. Note that a generic is used to represent the delay time. If one doesn't use a generic delay, the delay value has to be given in the architecture part. One then has to change the architecture code in order to change the delay value. Thus, the model is not reusable. By declaring generic delays in the interface part of an entity, the value of the generic delays can be specified when the entity is used. The benefit of doing so is that it enables the reuse of models.

The architecture body is associated with an entity declaration, and describes the internal organization or operation of a design entity. An architectural body is used to describe the algorithm, data flow, or structure of a design entity. An architecture body for the interface of Figure 2.1 is shown in Figure 2.2.

```
architecture BEHAVIOR of HORIZONTAL_FILTER is
signal TEMP1: FILTER_OUT:=0;          -- intermediate signal
signal TEMP_H: FILTER_OUT:=0;         -- intermediate signal
begin
```

```
  H_FILTER: process
  variable FIRST_LINE: PIX3:=(0,0,0);    -- a 3-stage buffer for the first scan line
  variable THIRD_LINE: PIX3:=(0,0,0);   -- a 3-stage buffer for the third scan line
  begin
        wait until rising_edge(clock);

        ------ store the input pixels in the first 3-stage buffer
        FIRST_LINE :=SHIFT_LEFT(FIRST_LINE,P1);

        ------ store the input pixels in the third 3-stage buffer
        THIRD_LINE :=SHIFT_LEFT(THIRD_LINE,P3);

        ------ apply horizontal filter function ------
        TEMP_H<= WEIGHT(FIRST_LINE(1), FIRST_LINE(2), FIRST_LINE(3))
                  -WEIGHT(THIRD_LINE(1), THIRD_LINE(2), THIRD_LINE(3));

        TEMP1 <= TEMP_H after HORIZ_DELAY;
   end process H_FILTER;

---------------- make H as a delay version of TEMP1 --------------------------
  process(TEMP1)
   begin
     H <= TEMP1 after WAIT_TIME;      -- horizontal filter output
   end process;
end BEHAVIOR;
```

**Figure 2.2** An architecture body of a horizontal filter

### 2.2.2 Behavioral Modeling

In the behavioral domain, a component is described by defining its input/output response. A behavioral model can exist at any level of abstraction. Behavioral descriptions in hardware description languages frequently are divided into two types: algorithmic and data flow.

An algorithmic model is a behavioral description in which the procedure defining the I/O response is not meant to imply any particular physical implementation [4]. It is a description of a system or component that reflects function, timing, and other aspects of the intended design. Figure 2.3 shows an algorithmic model for a 1-bit full adder.

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
```

```
entity FULL_ADDER is
   port(A, B, CIN : in STD_LOGIC:='0';
        SUM, COUT : out STD_LOGIC:='0');
end FULL_ADDER;

architecture ALGORITHMIC of FULL_ADDER is
variable TEMP: STD_LOGIC_VECTOR(2 downto 0);
begin
   TEMP := A & B & CIN;
   case TEMP is
        when "000" => COUT <= '0';
                      SUM <= '0';
        when "001"=> COUT <= '0';
                      SUM <= '1';
        when "010" => COUT <= '1';
                      SUM <= '0';
        when "011"=> COUT <= '1';
                      SUM <= '1';
        when "100" => COUT <= '0';
                      SUM <= '1';
        when "101"=> COUT <= '1';
                      SUM <= '0';
        when "110" => COUT <= '1';
                      SUM <= '0';
        when "111"=> COUT <= '1';
                      SUM <= '1';
        when others => NULL;
 end case;
end ALGORITHMIC;
```

**Figure 2.3** An algorithmic model of a 1-bit full adder

A data flow model is a behavioral description in which the data dependencies in the description match those in a real implementation. Data flow descriptions show how data moves between registers or gates. Figure 2.4 shows a data flow description of a 1-bit full adder.

```
 library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity FULL_ADDER is
   port(A, B, CIN : in STD_LOGIC:='0';
        SUM, COUT : out STD_LOGIC:='0');
```

```
end FULL_ADDER;

architecture DATAFLOW of FULL_ADDER is
begin
  SUM  <= A XOR B XOR CIN;                                  ----- sum
  COUT <= (A AND B) OR (A AND CIN) OR (B AND CIN);          ----- carry out
end DATAFLOW;
```

**Figure 2.4** A data flow description of a 1-bit full adder

### 2.2.3 Structural Modeling

In the structural domain, a component or system is described in terms of an interconnection of its constituent components. These components can be described behaviorally or structurally.

In order to make designs more understandable and maintainable, a design is typically decomposed into several blocks. These blocks are connected together to capture the physical organization of a particular implementation. This is called a structural model of the system. A structural model should follow the physical hierarchy of the system, which reflects the physical organization of a specific implementation**.** Figure 2.5 shows a structural model for a twelve-bit full adder which consists of twelve one-bit full adders.

```
------------------------------------------ library and package declarations --------------------------------
    library IEEE;
    use ieee.STD_LOGIC_1164.all;

    library STRUC_RTL;
    use STRUC_RTL.all;

---------------------------------------------- interface declaration ----------------------------------------
    entity SUM12_PP is
    port(ARG1   : in STD_LOGIC_VECTOR(11 downto 0):="000000000000";
         ARG2   : in STD_LOGIC_VECTOR(11 downto 0):="000000000000";
         RESULT : out STD_LOGIC_VECTOR(11 downto 0):="000000000000");
    end SUM12_PP;

---------------------------------- structural architecture declaration ------------------------------------
    architecture PARTS of SUM12_PP is
    component FULL_ADDER
        port(A, B, CIN : in STD_LOGIC:='0';
             SUM, COUT : out STD_LOGIC:='0');
```

```
end component;
for all : FULL_ADDER  use entity STRUC_RTL.FULL_ADDER(DATAFLOW);
signal C0, C1, C2, C3, C4, C5, C6, C7, C8, C9, C10, C11, C12 :STD_LOGIC:='0';
begin
        C0 <= '0';
        Fa0  : FULL_ADDER port map( ARG1( 0), ARG2( 0), C0,  RESULT( 0), C1 );
        Fa1  : FULL_ADDER port map( ARG1( 1), ARG2( 1), C1,  RESULT( 1), C2 );
        Fa2  : FULL_ADDER port map( ARG1( 2), ARG2( 2), C2,  RESULT( 2), C3 );
        Fa3  : FULL_ADDER port map( ARG1( 3), ARG2( 3), C3,  RESULT( 3), C4 );
        Fa4  : FULL_ADDER port map( ARG1( 4), ARG2( 4), C4,  RESULT( 4), C5 );
        Fa5  : FULL_ADDER port map( ARG1( 5), ARG2( 5), C5,  RESULT( 5), C6 );
        Fa6  : FULL_ADDER port map( ARG1( 6), ARG2( 6), C6,  RESULT( 6), C7 );
        Fa7  : FULL_ADDER port map( ARG1( 7), ARG2( 7), C7,  RESULT( 7), C8 );
        Fa8  : FULL_ADDER port map( ARG1( 8), ARG2( 8), C8,  RESULT( 8), C9 );
        Fa9  : FULL_ADDER port map( ARG1( 9), ARG2( 9), C9,  RESULT( 9), C10);
        Fa10 :FULL_ADDER port map( ARG1(10), ARG2(10), C10, RESULT(10), C11);
        Fa11 :FULL_ADDER port map( ARG1(11), ARG2(11), C11, RESULT(11), C12);
end PARTS;
```

**Figure 2.5** A structural model for a twelve-bit full adder

### 2.3 About the RASSP Design Process

The goal of the Rapid Prototyping of Application Specific Signal Processors (RASSP) is to dramatically improve the process by which complex digital systems, particularly embedded digital signal processors, are designed, manufactured, upgraded, and supported. A key objective of the RASSP design process is to get an improvement of at least a factor of four in the time required to take a design from concept to field prototype or to upgrade an existing design, that is, a 4X decrease in product development cycle-time, a 4X decrease in life-cycle costs and a 4X increase in product quality. This requires the use of interoperable models throughout all stages of the design process, which increase re-use and accelerate the design cycle by enabling library-based design techniques [15]. The following sections introduce some concepts of the RASSP design process. Details will be discussed in other chapters.

#### 2.3.1 Top-Down Design

The objective in RASSP is to produce a top-down design methodology in which a design is successively refined. Typically, a design evolves from word specification to system behavioral description, then to subsystem decomposition, to RTL models, to gate level models and finally to the circuit that can be physically fabricated. This is a complete design cycle in which a design evolves from a high abstraction level to a low abstraction level. As the design evolves, more and more details are introduced.

VHDL models are used to specify and document a particular hardware design. They are also used for testing and simulation purposes. The initial function and requirements of a system/subsystem is captured in VHDL as an executable specification, which can be verified by a VHDL simulator. The initial executable specification may need to be refined to eliminate ambiguities in the written specification. The executable specification can be inserted into a test bench to verify the system function. Also, it provides the requirements for the lower level designs. Synthesis tools can generate gate level models automatically from higher level models.

### 2.3.2 Reuse and the Model Year Concept

Over a life cycle of a design, the refinement and the upgrade of the design is processed in a growing and iterative fashion. The next version of a design is based on the current one to achieve functional improvements, reduction of power, cost, weight, etc., or replacement of an obsolete part. If the design of the previous version can be used as much as possible, then the redesign cycle can be shortened. This is the RASSP Model Year Concept, i.e., upgrading the design by reusing the existing components of the previous design without redesigning the whole system. Even for a new design, reuse of existing components can save design time and reduce design cost.

### 2.3.3 An Integrated Design Environment

This concept has two aspects, horizontal and vertical integrated development.

A horizontal integrated design environment fully supports concurrent design, development and the electronic exchange of product information. Since team members of a particular project may represent different companies, organizations and product development disciplines, there should be an integrated design environment which is able to support whatever tools are used in a heterogeneous computing environment. Also, this environment should be convenient for linking together work by different specialists, such as algorithm designers, hardware designers, system architects, manufacturing specialists, and others.

A vertical integrated design environment provides a distributed database containing all the product life-cycle data for both current and previous designs, as well as modular building blocks for design reuse. Models at different levels of abstraction and back annotation from low level models to high level models provide rapid and seamless changes from one level of design to another. Reuse of previous designs reduces both the design time and design cost.

## 2.4 The Sobel Edge Detection System

Image processing is a technique frequently used in a military weapon system to track targets. Figure 2.6 illustrates an image processing system that receives image data from a sensor and selects targets based on the processed input image data. The raw image is enhanced and then segmented to locate objects or regions of interest. Once an object or region has been located, it is

examined for identifying features that can lead to final classification of targets. Any identified target can then be tracked through subsequent images [5].


**Figure 2.6** Image processing system application

Before an image can be segmented, the objects in that image must be detected and roughly classified as to shape and boundary characteristics (edges).

Edges are significant local changes in the image and are important features for analyzing images. Edge detection is frequently the first step in recovering information from images. Many edge detectors, such as Roberts Operator, Sobel operator, Prewitt Operator, Laplacian Operator, etc., have been developed in the last two decades. Since the Sobel operator is one of the most commonly used edge detectors in image processing systems, we used this operator to illustrate our design methodology.

| | | |
|---|---|---|
| $a_0$ | $a_1$ | $a_2$ |
| $a_7$ | $[i, j]$ | $a_3$ |
| $a_6$ | $a_5$ | $a_4$ |

**Figure 2.7** The labeling of neighborhood pixels used to explain the Sobel operator

Consider the arrangement of pixels about the pixel [*i, j*] shown in Figure 2.7 [5]. The Sobel operator is the magnitude (M) of the gradient computed by

$$M = \sqrt{E_H^2 + E_V^2}\ , \tag{1}$$

where the partial derivatives are computed by

$$E_H = (a_0 + 2a_1 + a_2) - (a_6 + 2a_5 + a_4) \tag{2}$$
$$E_V = (a_2 + 2a_3 + a_4) - (a_0 + 2a_7 + a_6) \tag{3}$$

In order to reduce the complexity of the magnitude calculation, (1) is estimated as

$$M = \max\left[\left|E_H\right|,\left|E_V\right|,\left|E_{DL}\right|,\left|E_{DR}\right|\right] + \left\lfloor \frac{1}{8} \cdot E_{\perp} \right\rfloor \tag{4}$$

where

$$E_{DL} = (a_1 + 2a_2 + a_3) - (a_7 + 2a_6 + a_5) \tag{5}$$
$$E_{DR} = (a_1 + 2a_0 + a_7) - (a_3 + 2a_4 + a_5) \tag{6}$$

$\left|E_H\right|$, $\left|E_V\right|$ ,$\left|E_{DL}\right|$ and $\left|E_{DR}\right|$ are the absolute values of $E_H, E_V, E_{DL}, E_{DR}$, respectively. $E_{\perp}$ is the absolute value in the direction perpendicular to the direction of the maximum absolute value. $E_H, E_V, E_{DL}$ and $E_{DR}$ are called the differences in intensity along the horizontal, vertical, and left and right diagonal direction, respectively. They can be calculated individually by convoluting the image with four 3 x 3 windows (i.e., the Sobel operators) as shown in Figure 2.8. This procedure is also called filtering.

$S_H =$

| | | |
|---|---|---|
| 1 | 2 | 1 |
| 0 | 0 | 0 |
| -1 | -2 | -1 |

$S_V =$

| | | |
|---|---|---|
| -1 | 0 | 1 |
| -2 | 0 | 2 |
| -1 | 0 | 1 |

$S_{DL} =$

| | | |
|---|---|---|
| 0 | 1 | 2 |
| -1 | 0 | 1 |
| -2 | -1 | 0 |

$S_{DR} =$

| | | |
|---|---|---|
| 2 | 1 | 0 |
| 1 | 0 | -1 |
| 0 | -1 | -2 |

**Figure 2.8** The Sobel operators

Figure 2.9 gives an example of horizontal filtering.

The magnitude (M) is then compared to a particular threshold to determine the edge pixels. A pixel is declared to be an edge pixel if and only if M is greater than or equal to the threshold.

The gradient direction is defined as:

$$\Theta = \arctan\left[\frac{E_V}{E_H}\right]. \tag{7}$$

This direction can be estimated according to equation (7) and assigned from a template of eight digitized directions (East, Northeast, North, Northwest, West, Southwest, South and Southeast), as shown in Figure 2.10. Thus the resolution of the direction is $45^{\circ}$.

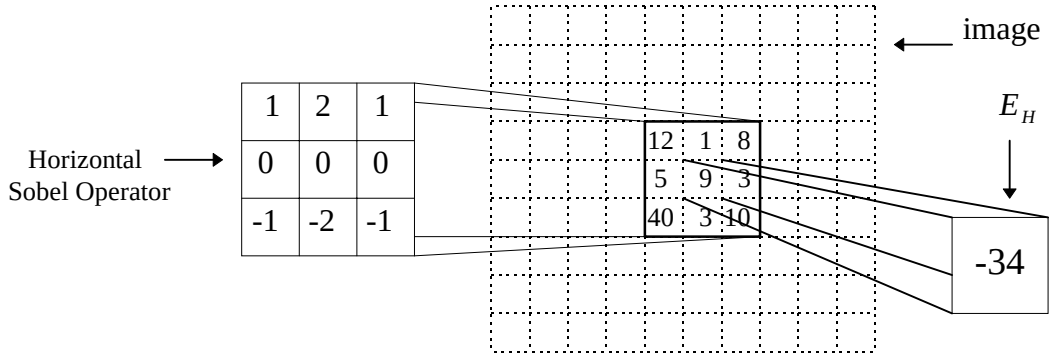



**Figure 2.9** An example of horizontal filtering

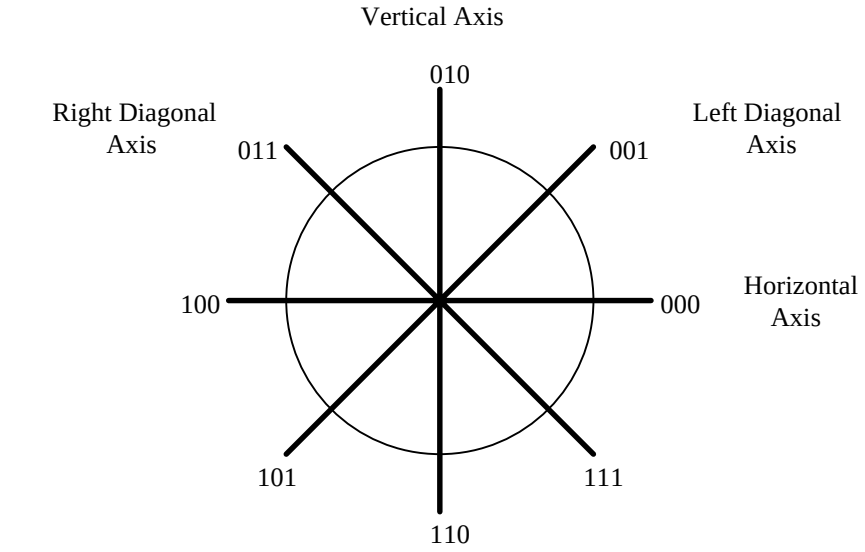



**Figure 2.10** Direction Assignment

For the example of Figure 2.9, $E_H$ = -34, $E_V$ = -38, $E_{DL}$ = -68, $E_{DR}$ =4 . Thus the maximum absolute value of the four filtering outputs is $|E_{DL}|$ = 68, $E_{\perp} = |E_{DR}|$ = 4. M is calculated by the following formula:

$$M = 68 + \left\lfloor \frac{1}{8} \cdot 4 \right\rfloor = 68 + 0 = 68 .$$

Since $E_{DL}$ is negative, the direction is assigned as “101” which stands for southwest.

The objective of our Sobel edge detection system is to perform the process of edge detection, including storing a portion of the input image, filtering, comparing and determining the edge pixels.

Frequently, the gray level of a pixel is assigned a value in the range 0 to 255, with 0 corresponding to black, 255 corresponding to white, and shades of gray distributed over the middle values [6]. This value can be represented by an 8-bit vector. Since the filter outputs are calculated by Equations (2), (3), (5) and (6), we need 2 more bits to represent the partial summation magnitude, and one more bit to represent the sign bit of the subtraction. As a result, we need 11 bits to represent the filtering output. In order to use a standard bus, 12-bit vectors are used to represent the four filtering outputs. After edge detection, each edge pixel is assigned to the *foreground value* (255), while non-edge pixels are assigned to the *background value* (0). In the previous example, M is equal to 68. If the THRESHOLD is less than 68, then the center pixel in the window is declared to be part of an edge and its value is set to be the foreground value (255). Otherwise, the center pixel is not declared to be an edge pixel and the background value (0) is assigned to it.

# Chapter 3. Multiple Levels of Abstraction in an Acquisition Process

## 3.1 Abstraction Hierarchy

An abstraction hierarchy is a set of interrelated representation levels that allows a system to be represented in varying amounts of details [4]. A designer begins with a word specification and ends at a working circuit that can be fabricated. The design time, cost and efficiency are directly affected by the design methodology. Figure 3.1 shows a picture of a typical abstraction hierarchy.

The silicon level is the lowest level in the hierarchy and is the level at which the models can be physically fabricated. The written specification is the highest level. The levels between these two levels are requirements repository, executable specification, several subsystem levels, RTL (Register Transfer Level), gate level, circuit level and silicon level. The subsystem levels are grouped to be the system architecture design level while the RTL down to the silicon level are grouped to be the detailed design level. A design can be represented at any of these levels. Note that the gate level is the lowest level at which we will create VHDL models.

Design consists of a series of transformations performed on the VHDL models that starts at the executable specification level and ends at the gate level. As one descends in the abstraction hierarchy, one captures more detail about the system and gets closer to a physical implementation. A designer needs to balance between the sufficiency and abstraction of a design in each step of the design process. It is important that a particular design activity be carried out at a level which has sufficient but not excessive details. Insufficient details will yield inaccurate results. Excessive details can make the design process too expensive [4].

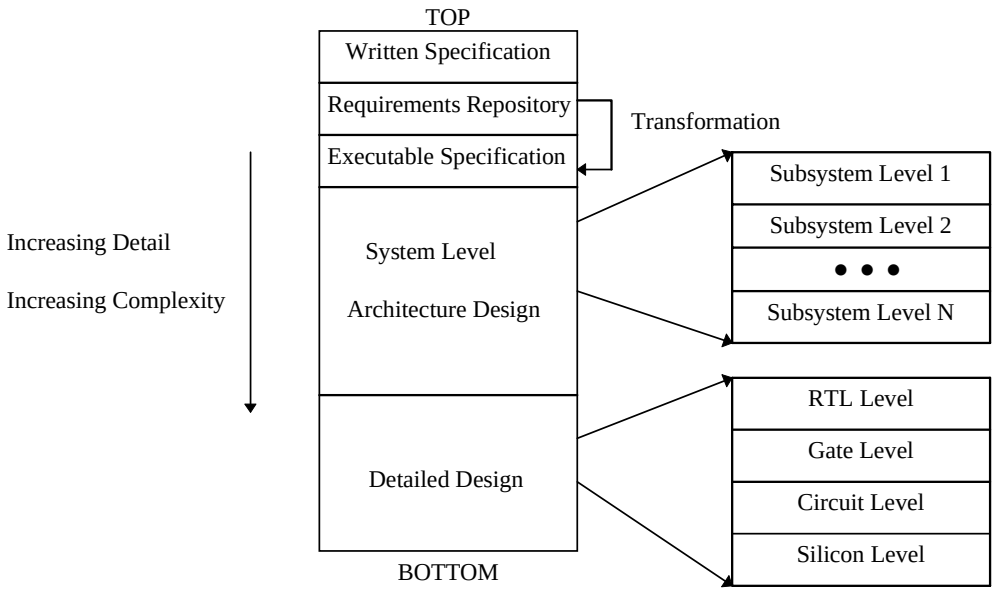



**Figure 3.1** The abstraction hierarchy

Typically, we have a graphical representation for the requirements repository and a behavioral description for the executable specification. Usually, a behavioral description of a model is developed first to verify the functionality. Then the model is decomposed into several components at the next lower level. Since the model is represented in terms of an interconnection of these lower level components, a structural description of this model is developed. Hierarchical design is a good engineering practice since it makes the design much easier to understand and manage.

A system can be represented by hierarchical models or a large flat design. A flat design gives every detail about the system at a given abstraction level in one blue print. All the components are leaf components that are not decomposed further. In order to make some change to a particular component, one must search for that component in a very complicated circuit diagram. This process is tedious and error-prone. Figure 3.2 shows the structure model for the component C which consists of six sub-components from A1 to A6. These sub-components are all leaf components. Assume component C is at level C, and sub-components A1 to A6 are at level A. Note that there are no components between these two levels. So this is a flat design.

On the other hand, a hierarchical design has models at different abstraction levels. Assume that the sub-components A4, A5, A6 are actually the sub-components of another component B. Then these three models can be replaced by component B. A behavioral model of B can be inserted into the structural model of component C, as shown in Figure 3.3. This kind of design is called hierarchical design.

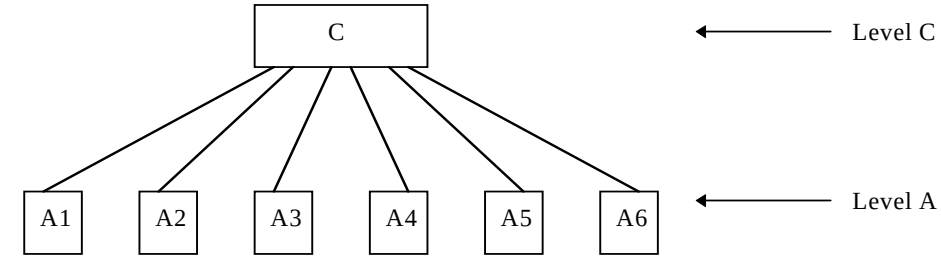



**Figure 3.2** An example of flat design

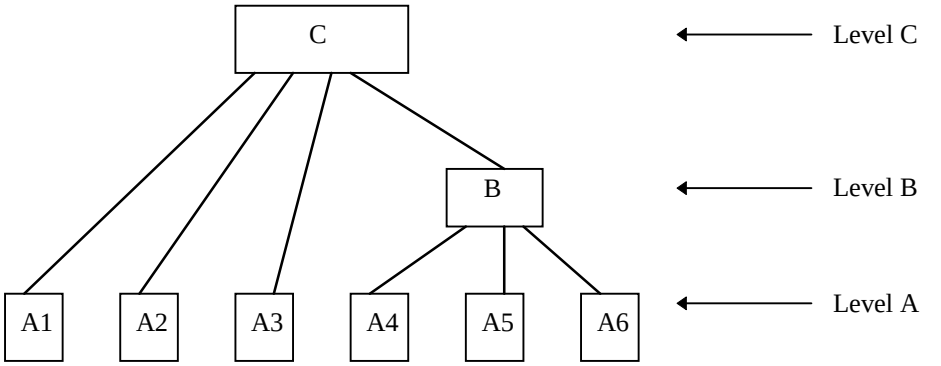



**Figure 3.3** An example of hierarchical design

We designed the Sobel edge detection system using the VHDL language and VSS (VHDL Synopsys Simulator) environment. Figure 3.4 shows the system decomposition of this system. As we can see, more design detail is presented when we navigate from top to bottom in the abstraction hierarchy. The following sections give a detailed explanation of each abstraction level and the corresponding model.

Before we describe the details of the abstraction levels, it is necessary to briefly introduce the concepts of top-down design and bottom-up design.

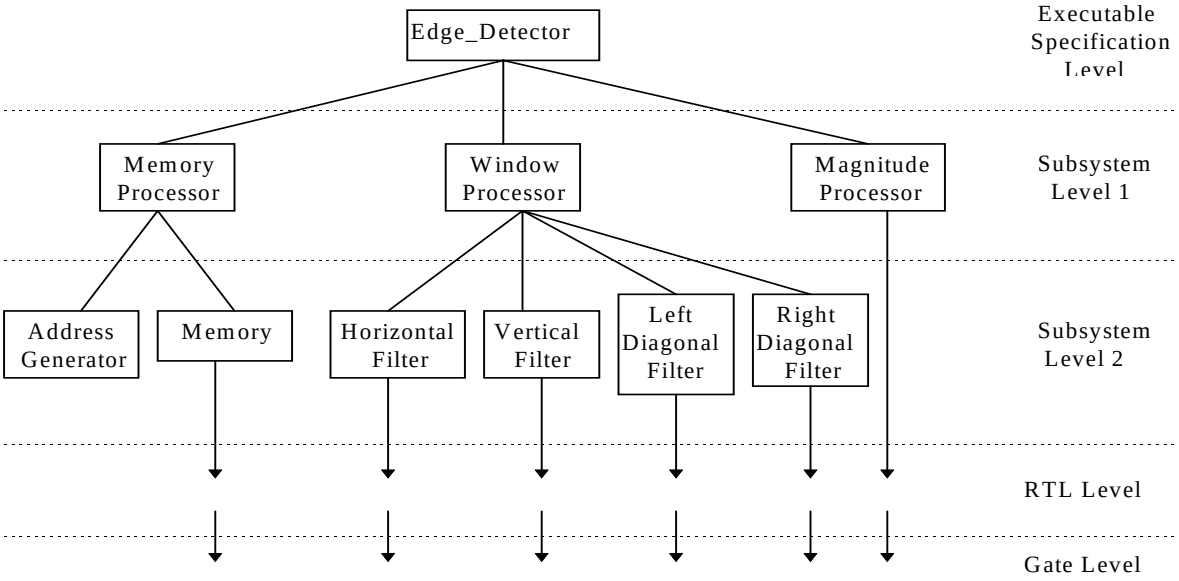



**Figure 3.4** System decomposition of the Sobel edge detection system

### 3.2 Top-Down Design and Bottom-Up Design

Top-down design and bottom-up design are the design techniques commonly used in industry. They refer to the sequence in which models at different levels of abstraction and different levels of hierarchical decomposition are developed. When a new model of a design is to be created, the designer can define a new level of hierarchy at the same level of abstraction, change the level of abstraction, or employ some combination of these two approaches.

Using top-down design, a design process evolves from the highest level of abstraction to the lowest level of abstraction. A word specification is first translated into an executable specification. This executable specification can be a behavioral model written in VHDL. This model is simulated to verify that the requirements are captured. The next step might be to create a structural model at the same level of abstraction that reflects a decomposition of the system into an interconnection of smaller components. These components might then be implemented as RTL models which could be synthesized into the gate level as an automated custom design or ASIC.

Alternatively, a bottom-up design can be created from an RTL model by interconnecting existing commercial parts. This is called a COTS (Commercial Off The Shelf) design. A module or sub-module is partitioned based on the available components. One benefit of this kind of design is that the components are standard and readily available. The system designed in this way may not be the best one, but the cost of overall design might be less because all the components are commercially available.

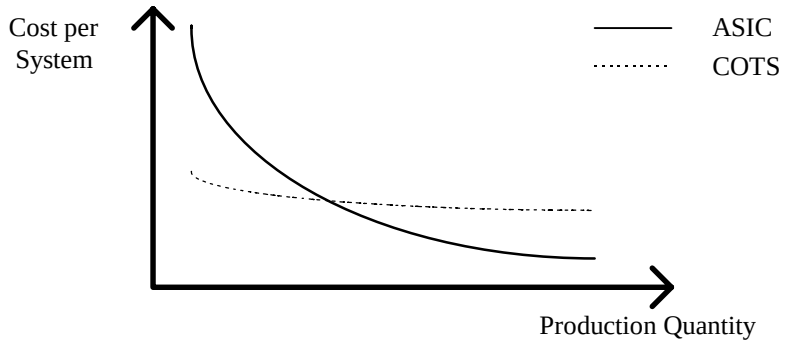


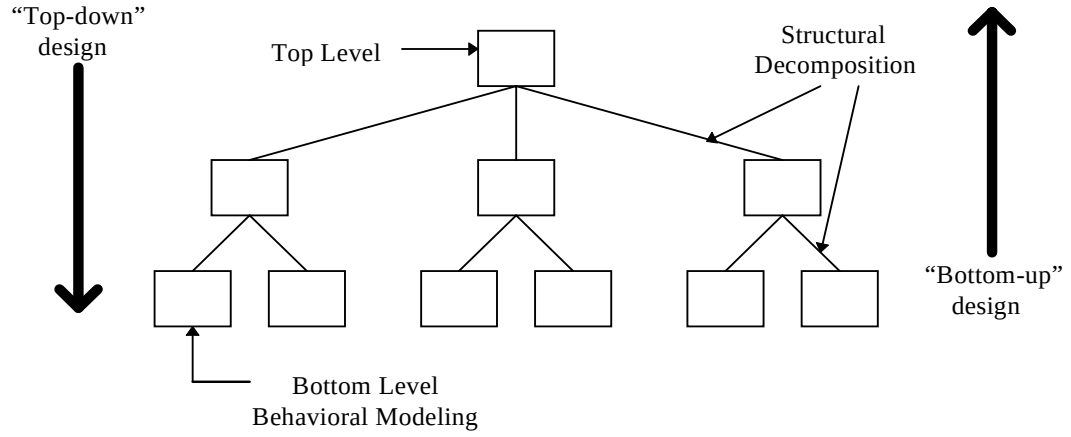

**Figure 3.5** Cost per system vs. production quantity

Comparing the two design approaches, the fabrication cost per device for the ASIC design is higher than that of the COTS design when the production quantity is small. This is because the fabrication mask is expensive. Thus, when the production quantity is small, the fabrication mask cost per chip is high. As the production quantity increases, the high mask cost is distributed over many chips and eventually the cost per chip for the ASIC design will be less than for the COTS design. Figure 3.5 presents the relationship between the chip cost and the production quantity.

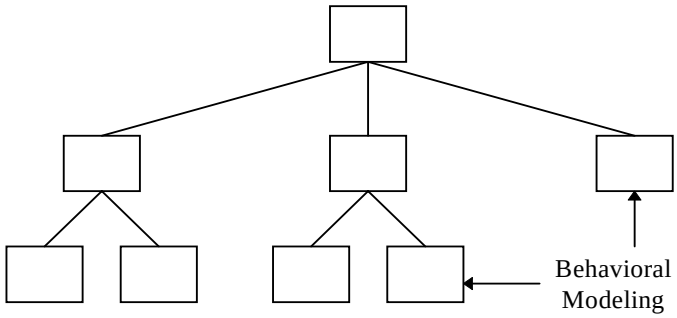
Figure 3.6 shows a structural decomposition of a system. The black bold arrows show the direction of top-down design and bottom-up design. As shown in the figure, the structure decomposition can be represented by a tree. Different levels of the tree correspond to the levels in the abstraction hierarchy. The models that are not decomposed are called leaf models and are represented by behavioral descriptions. The models other than leaf models are structural models. When the leaf models are at the same level, the design is called a *full tree design*; otherwise, *partial tree design* [4]. A partial tree design is a situation that happens very often. A designer frequently wants to evaluate the relationship between system components before all of them have been fully designed. After completing the design of a particular component, the designer wants to find out if the design will be compatible with other system components that are still under development. Thus one applies a partial tree simulation to the partial tree to check for compatibility. The method is sometimes called *multilevel simulation*.

After completing the detailed design of a component, the designer can get accurate and detailed timing information for it by performing individual component simulations. This information is used to update the timing of the behavioral description of the component developed at an earlier design stage. This updating process is referred to as *back annotation*. By applying back annotation, the timing for the higher level models become more accurate. When these models are used in multilevel simulations, we will obtain more accurate results.

In the Sobel edge detection system, top-down design is used.




(a) Full Tree Design




(b) Partial Tree Design

**Figure 3.6** Structural decomposition [4]

### 3.3 A Case Study of Top-Down Design

In a top-down design, a design moves from the top to the bottom in the abstraction hierarchy. Figure 3.7 shows a flow chart of the design process. The requirements repository of the system is first created using the Synopsys graphical environment (SGE) tool. Then a related behavioral model shell (with empty architecture and configuration) can be automatically generated by the SGE tool. A behavioral description of the architecture can be filled in to obtain an executable specification model. The written specification, the requirements repository and the executable specification are refined in an iterative fashion, as show in Figure 3.7. After the executable specification has met the design requirement, the system is decomposed into several smaller components which are at the same abstraction level or at the next lower abstraction level. This results in a structural model of the system. The components can themselves be further decomposed into smaller components. The decomposition process repeats until the components are at the RTL level. Then a synthesis tool is applied to the RTL level models to obtain gate level

models. The system decomposition of the Sobel edge detection system is shown in Figure 3.4. Detailed design will be illustrated in the following sections.

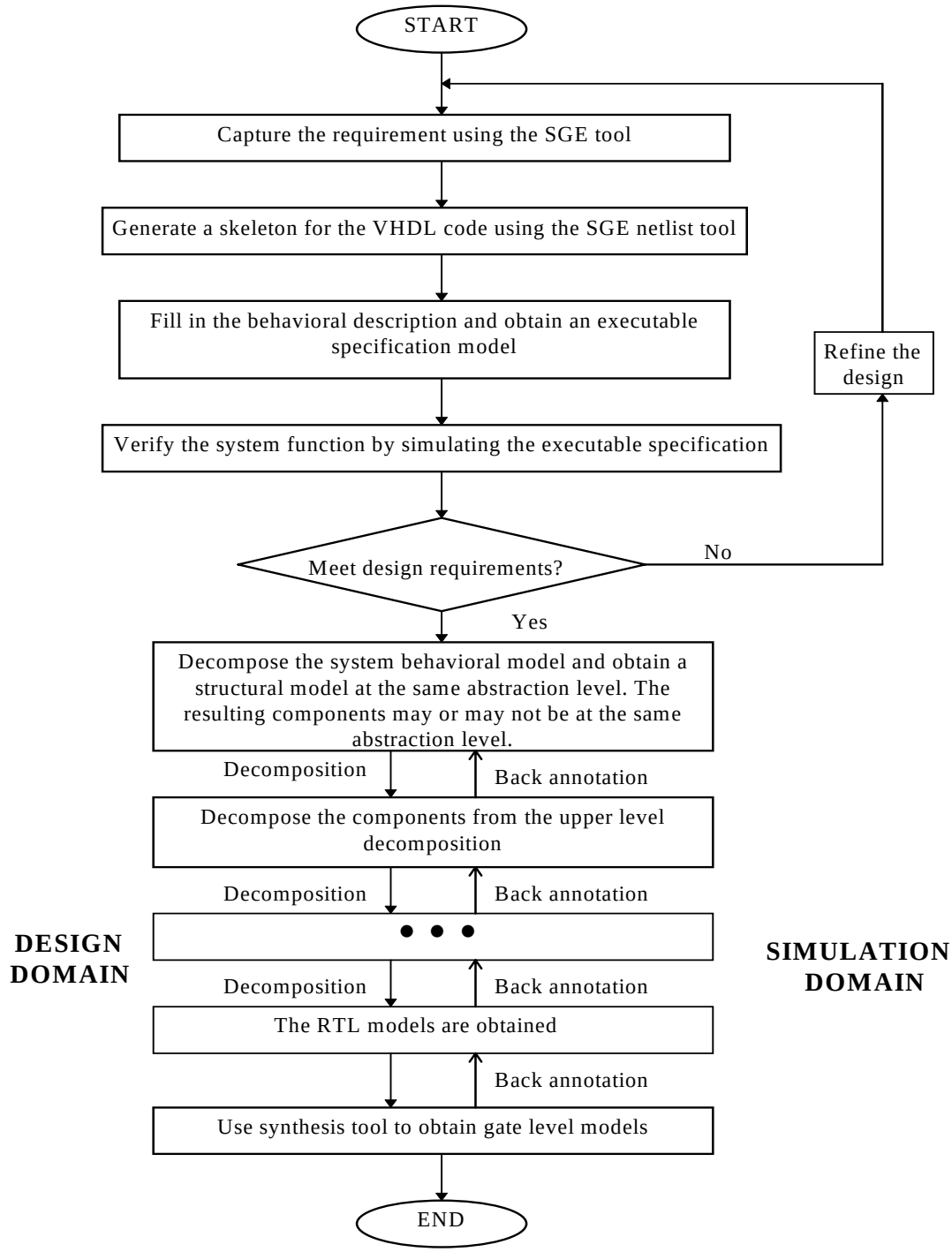



**Figure 3.7** Flow chart for the top-down design process

### 3.3.1 Written Specification

A written specification uses natural language, and possibly a block diagram and a timing diagram to describe a system to be designed. The input and output signals and the function of the system are specified. The basic knowledge about the system, such as what are the inputs and outputs, whether the system is synchronous or asynchronous, whether the signal ports are serial or parallel, are defined in the written specification. It is the place where the designer gets to know the system. However, the written specification might be ambiguous at times. As the design progresses, the designer will often refine the written specification to eliminate ambiguities.

The following is a written specification of the Sobel edge detection system:

*The Sobel edge detection system detects edges in images using the Sobel operator. The image pixels will be supplied in raster scan order (i.e. rows are scanned left to right and top to bottom.) The edge detector will serially output data which contains edge information. The system should be synchronized by a system clock. The system should be designed in such a way that images of various size can be processed.*

### 3.3.2 Requirements Repository

This is the first step in the design process. This is where interface of the system will be defined. The designer will decide the necessary ports for data and control signals which will communicate with the outside world to fulfill the system function. Also, the parameters of the system, such as the delays and size of the input data, will be declared. This design stage is usually related to the eventual executable specification level in the abstraction hierarchy. After the system interface has been designed, the designer can develop a behavioral architecture for the system to verify the functionality. The following section will discuss the requirements repository for the Sobel edge detection system. We will use the SGE tool to capture the requirements.

#### 3.3.2.1 About the SGE Design Tool

The SGE is an environment for creating and verifying designs [7]. In SGE, design entry is performed using a schematic capture editor. The port name, polarity, data type and initial value are defined using the *pin attributes* declaration tool. The parameters of the entity, if any, can be defined using the *symbol attribute* declaration tool. Then the related VHDL code can be generated automatically using the SGE *netlist* tool. The architecture and configuration parts are empty and ready for the designer to fill in. After the designer fills in the empty architecture, verification can be performed using the Synopsys VHDL System Simulator.

#### 3.3.2.2 Graphical Capture of the Sobel Edge Detection System

The ports and the generics of the system are declared by using the SGE tool. As mentioned before, the written specification, the requirements repository and the executable

specification are refined in a progressive and iterative fashion until the executable specification meets the design requirements. This process needs the creativity of the designer. Different results might be produced by different designers.

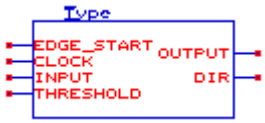



**Figure 3.8** Block diagram of the Sobel edge detection system

Figure 3.8 is an SGE diagram of the Sobel edge detection system at the executable specification level. The file name of this block diagram was chosen to be EDGE_DETECTOR. This name is also used in two other places.

1. As this block is used as a component by a higher level schematic, this name will replace the “type” shown in the above figure and will serve as the name of the component.

2. This name will also show up in the VHDL code automatically generated by SGE. The entity name will be declared as “EDGE_DETECTOR”.

The port name, port data type, in/out mode and the default value of the each port are declared as *Pin Attributes*, as shown in Figure 3.9.

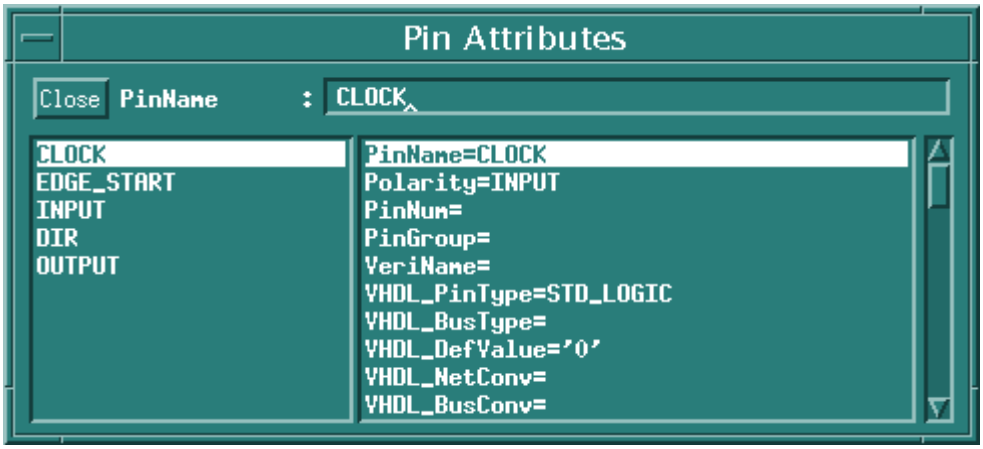



**Figure 3.9** Attributes for pin CLOCK

As shown in Figure 3.8, the EDGE_DETECTOR has four input ports, namely **CLOCK**, **EDGE_START**, **INPUT** and **THRESHOLD**. The data type of **EDGE_START** and **CLOCK** are STD_LOGIC. **EDGE_START** is the signal which starts the edge detection process. **CLOCK**

is the system clock. The edge detection system is synchronized by this signal. The data type of **INPUT** is *PIXEL*. The image pixels are input at pin **INPUT** in raster scan order. The first image row is scanned from left to right, then the second row, from left to right, etc., until all the image pixels are scanned in. At any point during image scanning, the last three rows are stored in an internal memory. The input signal **THRESHOLD** is of type *FILTER_OUT*. This value is compared with the intermediate value computed by the edge detection process to decide whether or not each pixel is an edge pixel. The system has two bidirectional ports, namely, **OUTPUT** and **DIR**. The output image pixels and the direction information for each pixel are output at these two pins, respectively. The data types for **OUTPUT** and **DIR** are PIXEL and *DIRECTION*, respectively. One of two values will be assigned to **OUTPUT**: if the current pixel is an edge pixel, output will be assigned the foreground value (255); if not, it will be assigned the background value (0). If a particular pixel is an edge pixel, **DIR** is given as one of the eight digitized directions which is perpendicular to the edge direction; if not, one of the eight directions determined by the algorithm will be given to **DIR**. The eight digitized directions are discussed in Chapter 2.

The data types, PIXEL, FILTER_OUT and DIRECTION, are user defined and used in the Sobel edge detection process. The PIXEL data type indicates the data is an image data which represents the gray level of the image. The FILTER_OUT data type indicates the data is the filtering output. The DIRECTION data type indicates the data is one of the eight digitized directions. The PIXEL and FILTER_OUT are constrained INTEGER data types, and the DIRECTION is STD_LOGIC data type for the executable specification. These data types are declared in the IMAGE_PROCESSING package which collects all useful data types, functions and procedures. We will discuss this package in detail later.

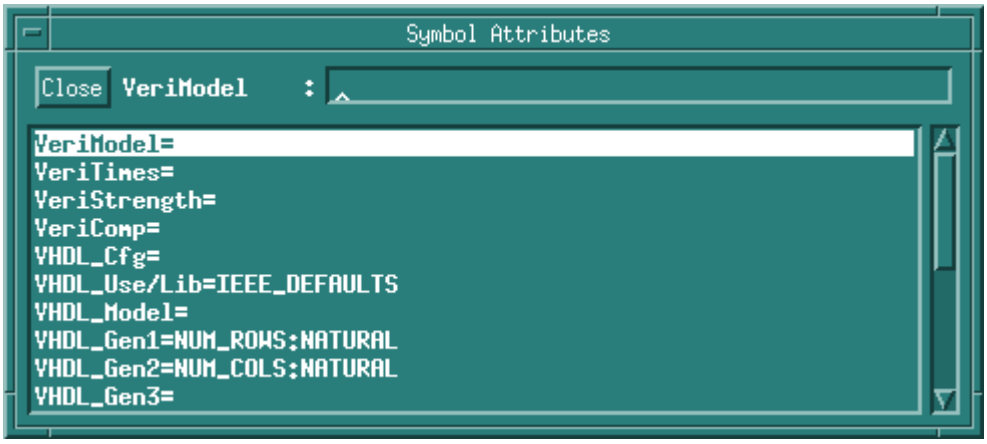



**Figure 3.10** Symbol attributes of the Sobel edge detection system

Figure 3.10 shows the declaration of the symbol attributes. The VHDL_Use/Lib defines the name of the section the netlister copies from the .synopsys_sge2vhdl.setup (provided by the

VHDL Synopsys Simulator) file to define the libraries and packages [7]. In our case, IEEE_DEFAULTS is given. This indicates the IEEE library and four packages (std_logic_1164, std_logic_misc, std_logic_arith and std_logic_components) will be used in the VHDL code automatically generated by the SGE netlist tool. Two generics, namely **NUM_ROWS** and **NUM_COLS**, are declared as NATURAL data type. They define the number of rows and columns of the input image, respectively. The actual values of the generics will be given later when the design entity is used as a component in a larger system. By assigning different values to the generics, this model can deal with images of different sizes. We will discuss the use of the generics later in Chapter 4.

VHDL code can be generated automatically using the SGE netlist tool, as shown in Figure 3.11. First, the IEEE library and the four packages (std_logic_1164, std_logic_misc, std_logic_arith and std_logic_components) are declared and made visible. Then the interface of the design entity is declared. The entity name is given as EDGE_DETECTOR as described before. The entity parameters (NUM_ROWS and NUM_COLS) are declared as generics. The port declarations are derived from the pin attributes in the SGE diagram. The architecture and configuration are empty and ready for a designer to fill in with a behavioral or structural description of the model and any necessary component bindings. Since this block diagram has no schematic related to it, it is treated as a leaf component and a behavioral model is assumed. Thus "BEHAVIORAL" and "CFG_EDGE_DETECTOR_ BEHAVIORAL" are assigned as the architecture and configuration name, respectively, by the SGE tool. The designer needs to make some modifications before this VHDL code can be analyzed and simulated, as discussed in the next section.

---

```
-- VHDL Model Created from SGE Symbol edge_detector.sym -- May 31 14:32:43 1997

library IEEE;
  use IEEE.std_logic_1164.all;
  use IEEE.std_logic_misc.all;
  use IEEE.std_logic_arith.all;
  use IEEE.std_logic_components.all;

entity EDGE_DETECTOR  is
    generic( NUM_ROWS: NATURAL;
             NUM_COLS: NATURAL);
    Port ( CLOCK : In    STD_LOGIC := '0';
           EDGE_ START : In  STD_LOGIC := '0';
           INPUT : In    PIXEL:=0;
           THRESHOLD: In FILTER_OUT:=0;
           DIR : InOut   DIRECTION :=0;
           OUTPUT : InOut   PIXEL:=0 );
end EDGE_DETECTOR;
```

```
architecture BEHAVIORAL of EDGE_DETECTOR  is
   begin
end BEHAVIORAL;

configuration CFG_EDGE_DETECTOR_BEHAVIORAL of EDGE_DETECTOR is
   for BEHAVIORAL
   end for;

end CFG_EDGE_DETECTOR_BEHAVIORAL;
```

**Figure 3.11** VHDL code of the edge detector (EDGE_DETECTOR) generated by the SGE tool

### 3.3.3 System Level Executable Specification

The initial function and requirements of a system are captured in VHDL as an executable requirement. This is the most abstract model in the design hierarchy that can be verified by a VHDL simulator. Any necessary refinements are applied to this model until a complete executable specification is obtained. Typically, this is a behavioral description of the system. This model can be inserted into a test bench to verify the function of the system. Also, this model provides the requirements for the next lower level design. The VHDL code generated by the SGE tool can be partially reused. Necessary modification is done to this code according to the application to develop the executable specification for the system.

The following sections explain the detailed steps to develop the system level executable specification of the Sobel edge detection system.

Figure 3.12 gives the VHDL code for the executable specification of the entity EDGE_DETECTOR. The code is explained in greater detail in Section 3.3.3.3 after we describe the IMAGE_PROCESSING package.

```
-- purpose: executable specification of the Sobel edge detector

----------------------------------------------------------------------------------------------------------
-----                    SECTION 1:  LIBRARY AND PACKAGE DECLARATION                    -----
----------------------------------------------------------------------------------------------------------
library IEEE;
   use IEEE.STD_LOGIC _1164.all;

---   deleted unnecessary packages
  -- use IEEE.std_logic_misc.all;
  -- use IEEE.std_logic_arith.all;
```

```
  --use IEEE.std_logic_components.all;

-- added a user defined library and package
library BEH_INT;
use BEH_INT.IMAGE_PROCESSING.all;

--------------------------------------------------------------------------------------------------------------
-----                    SECTION 2:   INTERFACE DECLARATION                    -----
--------------------------------------------------------------------------------------------------------------
entity EDGE_DETECTOR  is
  -- the generics were declared as symbol attributes in the SGE model
   generic( NUM_ROWS: NATURAL;                -- the number of rows of the image file
            NUM_COLS: NATURAL);               -- the number of columns of the image file
  -- the ports were declared as port attributes in the SGE model
   Port ( CLOCK : in    STD_LOGIC := '0';      -- the system CLOCK
          EDGE_ START : in  STD_LOGIC := '0';  -- the START signal for the image  processing
          INPUT : in    PIXEL:=0;            -- input image  pixel
          THRESHOLD: in FILTER_OUT:=0;       -- threshold for determine the edge pixel
          DIR : inout   DIRECTION :=0;         -- edge direction
          OUTPUT : inout   PIXEL:=0 );         -- output image  pixel
end EDGE_DETECTOR;

--------------------------------------------------------------------------------------------------------------
-----              SECTION 3:   ARCHITECTURE BODY DECLARATION              -----
--------------------------------------------------------------------------------------------------------------
architecture BEHAVIORAL  of EDGE_DETECTOR  is

---- added behavioral description
type MEMORY_ARRAY is array(1 to 3, 1 to NUM_COLS) of PIXEL;
---- begin architecture declaration
begin
  SOBEL: process
  variable BUSY1,BUSY2  : std_logic:='0';              --internal BUSY signal
  variable A     : PIXEL3_3:=((0,0,0),(0,0,0),(0,0,0));    -- temporary storage the input pixels
  variable TEMP  : PIXEL;              -- temporary storage for the output pixel
  variable E_H   :  FILTER_OUT;        -- temporary storage for results of horizontal filter
  variable E_V   :  FILTER_OUT;        -- temporary storage for results of  vertical filter
  variable E_DL  :  FILTER_OUT;        -- temporary storage for results of left_diagonal filter
  variable E_DR  :  FILTER_OUT;        -- temporary storage for results of right_diagonal filter
  variable M     :  FILTER_OUT;        -- temporary storage for results of maximum absolute
                                       -- value of the filters
  variable P     :  FILTER_OUT;        -- temporary storage for results of the perpendicular of
                                       --the maximum value of the filters
```

```
 variable MAG   :  FILTER_OUT;      -- output edge pixel
 variable PHASE: DIRECTION;         -- direction of the edge
-- row index of the internal memory for the WRITE operation
 variable X1:NATURAL:=1;
  -- column index of the internal memory for the WRITE operation
 variable Y1:NATURAL:=1;
 -- row index of the internal memory for the READ operation
 variable X2:NATURAL:=1;
 -- column index of the internal memory for the WRITE operation
 variable Y2:NATURAL:=1;
 variable COUNT_R1,COUNT_R2: INTEGER:=0;                    -- internal counters
 variable MEMORY: MEMORY_ARRAY;     -- memory array, size: 3 rows x num_cols column

----- begin the SOBEL process -----
 begin
   wait until rising_edge(CLOCK);
--------------  Set the internal busy variable ------------------------------------------------
          if EDGE_START =’1’ then
               BUSY1:=’1’;
          end if;

---------------- Store the input image data in a memory array   -------------------------
          if BUSY1 =’1’ then
                    MEMORY(X1,Y1):=INPUT;

---- set internal busy variable BUSY2 and start generate the address for the READ operation ----
                    if COUNT_R1=2 and Y1=1 then
                              BUSY2:=’1’;
                    end if;
---------------- calculate the memory location for the next WRITE operation -------
                    Y1:=Y1+1;
                   --- start a new row ---
                    if Y1=NUM_COLS+1 then
                              Y1:=1;
                              X1:=X1+1;
                    ------ count number of rows ------
                              COUNT_R1:=COUNT_R1+1;
                    end if;
                    ----- reuse first row of internal memory -----
                    if X1=4 then
                              X1:=1;
                    end if;
---------- Reset BUSY1  if all rows have been read in ----------------------------------------------
```

```
                if COUNT_R1=NUM_ROWS then
                        BUSY1:='0';
                end if;
        end if;

---------- Read out the image data from the memory  ----------------------------------
        if BUSY2='1' then
----------- Update the 3 x 3 buffer window A ------------------------------------------
                ---- move columns 2 and 3 left one position -----
                for J in 1 to 2 loop
                        for I in 1 to 3 loop
                                A(I, J):= A(I, J+1);
                        end loop;
                end loop;
                ---- insert new values for column 3 -----
                for I in 1 to 3 loop
                        X2:= (COUNT_R2 +I-1)mod 3+1;
                        A(I, 3):=memory(X2,Y2);
                end loop;
--------- calculate the memory location for the next memory READ operation  ----
                Y2:=Y2+1;
                ---- update column counter -----
                if Y2=NUM_COLS+1 then
                        COUNT_R2:=COUNT_R2+1;
                        ---- reset column index to start of next column ----
                        Y2:=1;
                end if;
----------- Reset the internal busy variable BUSY2  -----------------------------------
                if COUNT_R2 = NUM_ROWS-2 then
                        BUSY2 := '0';
                end if;

--------- apply the filtering function to the input image data -------------------------
                E_H  := HORIZONTAL_FILTER(A);
                E_V  := VERTICAL_FILTER(A);
                E_DL := DIAGONAL_L_FILTER(A);
                E_DR := DIAGONAL_R_FILTER(A);

--------- determine the output edge pixels and directions ----------------------------
                COMPARE(E_H,E_V,E_DL,E_DR,M,P,PHASE);
                MAG:= MAGNITUDE(M,P);
                if MAG >= THRESHOLD then
                        TEMP := FOREGROUND;
```

```
                else
                        TEMP := BACKGROUND;
                end if;
---------- Set out the outputs ----------------------------------------------------------
                OUTPUT <= TEMP;
                DIR <= PHASE;
        end if;
    end process SOBEL;
end BEHAVIORAL;

---- delete unnecessary configuration part for the behavioral model
-- configuration CFG_EDGE_DETECTOR_BEHAVIORAL of EDGE_DETECTOR is
--   for BEHAVIORAL
--   end for;

-- end CFG_EDGE_DETECTOR_BEHAVIORAL;
```

**Figure 3.12** VHDL code for the executable specification of the entity EDGE_DETECTOR

#### 3.3.3.1 Library Declaration

In order to manage the models, libraries are used to group the analysis results of the VHDL models. A library is declared by the *library* clause, followed by the library name [4]. The following statement declares the A library.

*Library A*;

Models inside a library can be made visible by the *use* clause. The following statement makes the model B, stored in library A, visible.

*Use A.B;*

The library and package declarations for the Sobel edge detection system are shown in section 1 of Figure 3.12. It is different from the same portion of the code in Figure 3.11 which is automatically generated by SGE. In our example, two libraries are used. One is the built-in IEEE library. In this library, only the std_logic_1164 package is made visible, because we do not need the other three packages which are for synthesis. The other library is a user defined library, namely BEH_INT. In this design library, we store the analysis results of the executable specification model and several other models, such as the IMAGE_PROCESSING package and the clock generator. These models function together to fulfill the executable specification simulation. In this library, the data type for the input, output (except DIR) and intermediate image data are INTEGER. The data type for DIR is STD_LOGIC. The use of user defined

libraries aids the management of the design and the reuse of the code. We will discuss library organization further in Chapter 4.

### 3.3.3.2 Packages Used in the Entity EDGE_DETECTOR in BEH_INT Library

It is tedious and unnecessary to repeat declarations, such as data type declarations, function declarations, and procedure declarations, each time they are used. A package is a place where the frequently used declarations are stored. Then each time they are needed, they are made visible by a *use* statement.

Figure 3.13 shows the image processing package declaration used for the executable specification of the EDGE_DECTOR.

---

```
-- file name: imageprc.vhd
-- purpose: IMAGE_PROCESSING package declaration (for BEH_INT and BEH_STRUC libraries)

------------------ library and package declaration ------------
library IEEE;
use IEEE.STD_LOGIC_1164.all;

---------------- package declaration -----------------------------
package IMAGE_PROCESSING is
---------------- declare two constants --------------------------
constant FOREGROUND:INTEGER:=255;    -- define the value for the foreground pixel
constant BACKGROUND:INTEGER:=0;      -- define the value for the background pixel

---------------- declare types and subtypes ----------------------
subtype PIXEL is INTEGER;
type PIX3 is array (1 to 3) of PIXEL;
type PIXEL3_3 is array(1 to 3, 1 to 3) of PIXEL;
subtype FILTER_OUT is INTEGER;
subtype DIRECTION is STD_LOGIC_VECTOR(2 downto 0);

-------------- declare 8 direction constants --------------------
constant EAST:DIRECTION:="000";
constant NORTHEAST:DIRECTION:="001";
constant NORTH:DIRECTION:="010";
constant NORTHWEST:DIRECTION:="011";
constant WEST:DIRECTION:="100";
constant SOUTHWEST:DIRECTION:="101";
constant SOUTH:DIRECTION:="110";
```

```
constant SOUTHEAST:DIRECTION:="111";

---------- declare the names and parameters of  functions and a procedure ----------
function WEIGHT ( X1,X2,X3: PIXEL)
   return FILTER_OUT;
function SHIFT_LEFT ( A: PIX3; B: PIXEL)
   return PIX3;
function HORIZONTAL_FILTER ( A: PIXEL3_3)
   return FILTER_OUT;
function VERTICAL_FILTER ( A: PIXEL3_3)
   return FILTER_OUT;
function DIAGONAL_L_FILTER ( A: PIXEL3_3)
   return FILTER_OUT;
function DIAGONAL_R_FILTER ( A: PIXEL3_3)
   return FILTER_OUT;
procedure COMPARE ( H,V,LD,RD :in FILTER_OUT;
                              X,Y: out FILTER_OUT; DIR:out DIRECTION);
function MAGNITUDE ( A,B:  FILTER_OUT)
   return FILTER_OUT;
function INT_TO_STDLOGIC8 ( A: INTEGER)
   return STD_LOGIC_VECTOR;
function STDLOGIC_TO_INT ( S:STD_LOGIC_VECTOR)
   return INTEGER;
function INT_TO_STDLOGIC12 (A: INTEGER)
   return STD_LOGIC_VECTOR;
function STDLOGIC_TO_BIT (A: STD_LOGIC_VECTOR)
    return BIT_VECTOR;
end IMAGE_PROCESSING;
```

**Figure 3.13** IMAGE_PROCESSING package declaration

In the package declaration of the image processing package shown in Figure 3.13, constants, data types, a procedure and some functions are declared. FOREGROUND and BACKGROUND are the two constants which define the value of a foreground pixel and a background pixel, respectively. Data types, such as PIXEL (the data type of the image data), FILTER_OUT (the data type of the output of the filters), etc., are declared in order to reduce repeat declaration of them in other VHDL models. In this particular example, most of the data types are constrained INTEGER. INTEGER data types are frequently used at the highest abstraction level because the written specification frequently are stated in terms of integer quantities and because the simulation time is less than using BIT or STD_LOGIC data types.

Functions and procedures are used to make the design modular and allow reuse of the code. These subprograms are called when used. The interfaces to them are declared in the package declaration.

The code for the functions and procedures are given in the package body. If a package contains no such subprograms, a package body is not required. Figure 3.14 gives the package body of the package shown in Figure 3.13.

---

```
-- purpose: IMAGE_PROCESSING package body description (for BEH_INT and BEH_STRUC
-- libraries

library IEEE;
use IEEE.STD_LOGIC_1164.all;

---------------- package body description ----------------------
package body IMAGE_PROCESSING is

---------------- WEIGHT function --------------------------------
function WEIGHT ( X1,X2,X3: PIXEL)
   return FILTER_OUT is
begin
   return X1+ 2*X2 + X3;
end WEIGHT;

---------------- SHIFT LEFT function ----------------------------------
function SHIFT_LEFT ( A: PIX3;  B: PIXEL)
   return PIX3 is
begin
   return A(2) & A(3) & B;
end SHIFT_LEFT;

---------------- HORIZONTAL_FILTER function ---------------------
function HORIZONTAL_FILTER ( A: PIXEL3_3)
   return FILTER_OUT is
begin
  return  WEIGHT( A(1,1), A(1,2), A(1,3) ) - WEIGHT( A(3,1), A(3,2), A(3,3) );
end HORIZONTAL_FILTER;

---------------- VERTICAL_FILTER function ----------------------
function VERTICAL_FILTER ( A: PIXEL3_3)
   return FILTER_OUT is
begin
```

```
    return WEIGHT( A(1,3), A(2,3), A(3,3)) -WEIGHT(A(1,1), A(2,1), A(3,1));
end VERTICAL_FILTER;

---------------- DIAGONAL_L_FILTER function ----------------------
function DIAGONAL_L_FILTER ( A:PIXEL3_3)
   return FILTER_OUT is
begin
   return WEIGHT( A(1,2), A(1,3), A(2,3)) -WEIGHT(A(2,1), A(3,1), A(3,2));
end DIAGONAL_L_FILTER;

---------------- DIAGONAL_R_FILTER function ----------------------
function DIAGONAL_R_FILTER ( A:PIXEL3_3)
   return FILTER_OUT is
begin
   return  WEIGHT( A(1,2), A(1,1), A(2,1)) -WEIGHT( A(2,3), A(3,3), A(3,2) );
end DIAGONAL_R_FILTER;

---------------- COMPARE procedure ------------------------------
procedure COMPARE( H,V,LD,RD :in FILTER_OUT;
                           X,Y: out FILTER_OUT;
                                   -- X is the maximum absolute value of the four inputs
                                  -- Y is the absolute value that is perpendicular to X
                          DIR: out DIRECTION) is  -- the direction of the edge pixel
variable AH: FILTER_OUT:=0;
variable AV: FILTER_OUT:=0;
variable ALD: FILTER_OUT:=0;
variable ARD: FILTER_OUT:=0;
variable C: FILTER_OUT:=0;       -- temperate storage for the current maximum absolute value
variable D: FILTER_OUT:=0;      -- temperate storage for the value which is perpendicular to C
variable DIREC: DIRECTION;
begin

-- get absolute values
  if H <0 then AH := -H;
  elsif H>=2049 then AH:=4096-H;
  else AH := H;
  end if;

  if V <0 then AV := -V;
  elsif V>=2049 then AV:=4096-V;
  else AV := V;
  end if;
```

```
 if LD <0 then ALD := -LD;
 elsif LD>=2049 then ALD:=4096-LD;
 else ALD := LD;
 end if;

if RD <0 then ARD := -RD;
 elsif RD>=2049 then ARD:=4096-RD;
 else ARD := RD;
 end if;

--------- get the maximum value and the value that is perpendicular to the maximum ---------

 -- First, assume AH is the maximum
 C:=AH;
 D:=AV;
        if H>0 then DIREC:="010";
        else DIREC:="110";
        end if;

-- if AV>AH, then update C and D to indicate AV is the maximum so far
if AV>C then
   C:=AV;
   D:=AH;
        if V>0 then DIREC:="000";
        else DIREC:="100";
        end if;
end if;

 -- Check to see if ALD is larger than AH and AV. If so, the update C and D to indicate ALD is
-- maximum so far
if ALD>C then
    C:=ALD;
    D:=ARD;
        if LD > 0 then DIREC:="001";
        else DIREC:="101";
        end if;
end if;

-- If ARD is the maximum, then update C and D to indicate ARD is the maximum
 if ARD > C then
      C:=ARD;
      D:=ALD;
        if RD >0 then DIREC := "011";
```

```
        else DIREC := "111";
        end if;
 end if;
 X:=C;
 Y:=D;
 DIR:= DIREC;
end COMPARE;

----- calculate the magnitude according to the Sobel algorithm discussed in Chapter 2 ----------
function MAGNITUDE (A,B:  FILTER_OUT)
   return FILTER_OUT is
begin
        return (A + (B/8));
end MAGNITUDE;

---------- STD_LOGIC_VECTOR TO INTEGER function ----------------------
function STDLOGIC_TO_INT(S:STD_LOGIC_VECTOR) return INTEGER is
  variable RES : INTEGER:=0;
begin
  for I in S'low to S'high loop
        if S(I)='1' then RES:=RES + (2 ** I);
        end if;
  end loop;
return RES;
end STDLOGIC_TO_INT;

---------- INTEGER TO STD_LOGIC_VECTOR( 7 downto 0) function ----------
 function INT_TO_STDLOGIC8(A: INTEGER) return STD_LOGIC_VECTOR is
  variable RESULT : STD_LOGIC_VECTOR(0 to 7):="00000000";
  variable TEMP_A : INTEGER:=0;
  variable TEMP_B : INTEGER:=0;
 begin
  TEMP_A := A;
  for I in 7 downto 0 loop
   TEMP_B:=TEMP_A/(2**I);
   TEMP_A:=TEMP_A rem(2**I);
   if(TEMP_B = 1) then
        RESULT(7-I):='1';
   else
        RESULT(7-I):='0';
   end if;
  end loop;
 return RESULT;
```

```
end INT_TO_STDLOGIC8;

---------- INTEGER TO STD_ULOGIC_VECTOR(11 downto 0) function ----------
function INT_TO_STDLOGIC12(A: integer) return STD_LOGIC_VECTOR is
  variable RESULT : STD_LOGIC_VECTOR(0 to 11):="000000000000";
  variable TEMP_A : integer:=0;
  variable TEMP_B : integer:=0;
 begin
  TEMP_A := A;
  for I in 11 downto 0 loop
   TEMP_B:=TEMP_A/(2**I);
   TEMP_A:=TEMP_A rem(2**I);
   if(TEMP_B = 1) then
        RESULT(11-I):='1';
   else
        RESULT(11-I):='0';
   end if;
  end loop;
 return RESULT;
end INT_TO_STDLOGIC12;

------------ STD_LOGIC_VECTOR to BIT_VECTOR function ----------------------
------- This function is used for WAVES -----------------------------------------------
function STDLOGIC_TO_BIT (A: STD_LOGIC_VECTOR) return BIT_VECTOR is
variable B: BIT_VECTOR(A'length-1 downto 0);
begin
    for I in A'low to A'high loop
      if A(I)='1' then
                B(I):='1';
        else B(I) :='0';
        end if;
    end loop;
    return B;
end STDLOGIC_TO_BIT;
end IMAGE_PROCESSING;
```

**Figure 3.14** IMAGE_PROCESSING package body declaration

### 3.3.3.3 The Behavioral Description of the Sobel Edge Detection System

The behavior of the EDGE_DETECTOR is described in this section. This model is the most abstract model in the design hierarchy and is used to capture the function of the system and generate requirements for the next lower level design. The VHDL code is shown in Figure 3.12.

In each clock cycle, one memory address is calculated for the memory WRITE operation. Then an image pixel from the outside is written to that memory location. When the location of the third row, the first column of the memory has been written, three addresses for the memory READ operation are generated. Then the memory READ operation is implemented and three data are read out from memory and written to a 3 x 3 window buffer. The memory READ operation follows the WRITE operation, and they are implemented in the same clock period. Figure 3.15 illustrates how the image data values are written to the window buffer.

As shown in Figure 3.15, the window buffer is updated each cycle. Assume Window[i] is the current window. In the next clock cycle, first the three data values in Column 2 of Window[i] are used to update the three window cells in Column 1 of the same row. Next, Column 2 is used to updated by Column 3 which is then updated by three new coming data values. As a result, a new window, namely Window[i+1], is obtained.

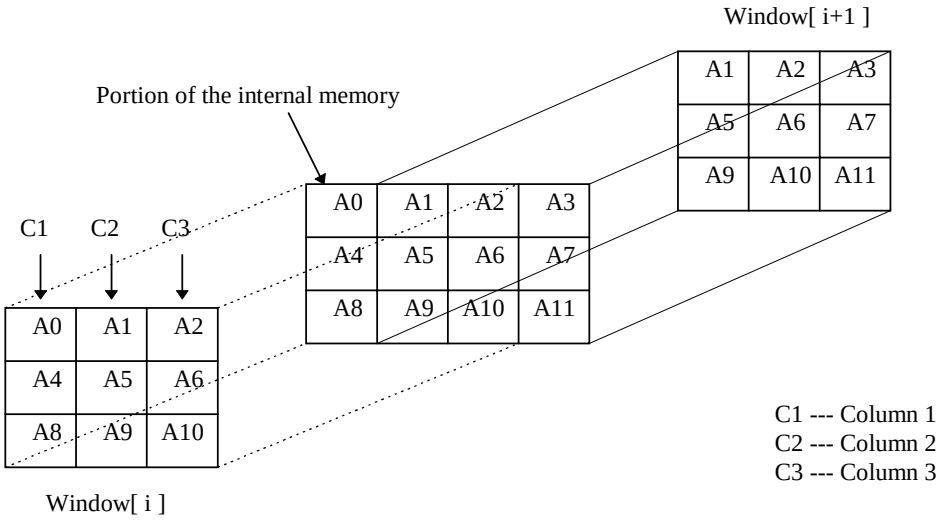



**Figure 3.15** Updating the window buffer

Four Sobel operators (Horizontal Filter, Vertical Filter, Left Diagonal Filter and Right Diagonal Filter) are then performed on the data values in the current window buffer (Window[i+1]) according to the Sobel algorithm discussed in Chapter 2. The filtering outputs are used to calculate the magnitude which is then compared with the threshold to determine whether the pixel location is an edge pixel or not. Both the pixel value and the pixel direction are send out. These actions are performed in one process, named “SOBEL”. Every clock cycle, the process is executed once and one image pixel is checked to see if it is an edge pixel.

Several variables are declared and used as temporary storage in the process. **BUSY1** is set to logic ‘1’ during the memory WRITE operation. At the instant when **BUSY1** is set, the internal counter **COUNT_R1** begins to count how many rows of the image have been written into the memory. It stops counting when all the image data has been written to the memory. Then **BUSY1** is reset to logic ‘0’ and the WRITE operation ends. **BUSY2** is another internal variable which is related to the memory READ operation. It works in a similar sense to **BUSY1**. Another internal counter **COUNT_R2** counts how many rows of image have been read from the memory. It stops counting when all the image pixels have been read out from the memory. The internal variable **BUSY2** is reset by that time. (**X1**, **Y1**), (**X2**, **Y2**) store the two-dimensional addresses for the memory WRITE and READ operation, respectively. **X1**, **Y1** and **Y2** are updated once each clock cycle; **X2** is updated three times in one clock cycle for the three memory READ locations.

The following paragraphs describe the edge detection process in detail.

At each rising edge of **CLOCK**, the value of **EDGE_START** is checked. If it is logic ‘1’, **BUSY1** is set to ‘1’ and the “SOBEL” process starts. **EDGE_START** needs to remain at logic ‘1’ for at least one clock period.

As long as **BUSY1** is logic ‘1’, one pair of memory addresses (**X1**, **Y1**) is calculated for the memory WRITE operation. Then one image data comes in from **INPUT** port and is written to that memory location. Since the size of the Sobel operator is 3 x 3, the number of rows of the image in use is 3. Thus the minimum size of the internal memory is 3 x *NUMBER_OF_COLUMNS*. For the memory WRITE operation, the input image data is written to the memory from the first column to the last column in the first row. After the first row is filled up, the data is written to the second row, then the third row. In order to reduce the size of the memory, these three rows are reused. The WRITE process repeats from the first row to the third row until all the image data has been written to the memory once. The internal counter, **COUNT_R1**, is updated when one row of image pixels have been written into the memory. This signal counts for the end of the WRITE operation and resets **BUSY1** to logic ‘0’ at that time.

As the first column, the third row of the memory is written for the first time, another internal variable **BUSY2** is set to logic ‘1’. This indicates the beginning of the memory ***READ*** operation. The memory address pair (**X2**, **Y2**) is calculated and the data value of that location is read out. The value of **Y2** is equal to that of **Y1**; this means that the WRITE and READ operations are applied to the same column of the memory. In order to send three data in the same column to the filters simultaneously, **X2** is updated three times in one clock cycle, thus three data values are read out from the memory every clock cycle. Then, the edge detection function is applied to these data values read from the memory. The internal counter, **COUNT_R2,** detects the end of the memory READ and edge detection operation, and resets **BUSY2** to logic ‘0’.

For the memory READ operation, the data is read out from the memory and written to a 3 x 3 buffer array **A**. **A** is convoluted with the Sobel operator, i.e., **A**(I, J) x **S**(I, J) for I, J = 1 to 3. This procedure is called filtering, and the sum of these products is called the filtering output.

In our edge detection algorithm, the image is filtered by four filters, namely, *Horizontal Filter*, *Vertical Filter*, *Left Diagonal Filter* and *Right Diagonal Filter*. Since these are four independent functions, they can be executed simultaneously. Then, a comparison function is applied to the filtering outputs. The maximum absolute value of the four filtering outputs and the absolute value perpendicular to it are found. An addition function adds these two values together, and the sum is compared to a given threshold. If the sum is equal to or greater than the threshold, the center pixel of **A** is determined to be an edge pixel. The value of a **FOREGROUND** pixel is assigned to this pixel location, and the direction information is calculated according to the Sobel Algorithm discussed in Chapter 2. Otherwise, the value of a **BACKGROUND** pixel is given to this pixel location and the direction is calculated in the same way as that for the edge pixel. **OUTPUT** and **DIR** are the signals for the output image data and direction, respectively. The **BACKROUND** and **FOREGROUND** pixel value can be defined by the designer. The defaults are 0 for **BACKROUND** pixel and 255 for the **FOREGROUND** pixel.

Notice that several generics are used, such as **NUM_ROWS** (number of rows of the image), **NUM_COLS** (number of columns of the image). Since the edge detection system should work well for any image of any size, these values are not declared until later. The actual values of these generics are declared in the test bench where the model is tested**.** Figure 3.16 and Figure 3.17 shows the image of a tank and its edges as detected by the executable specification model shown in Figure 3.12.

**Figure 3.16** Input image of the M1A1 tank

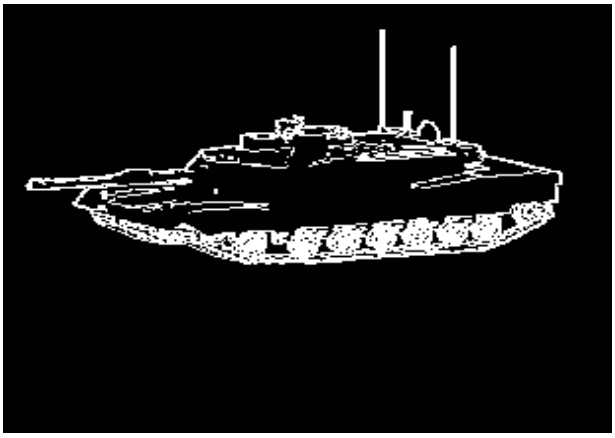


**Figure 3.17** Edge detected output image for the M1A1 tank

For the development of the test bench and the generation of the test vectors, please refer to [8].

### 3.3.4 System Decomposition

Following verification of the executable specification, the next design step is to decompose the system into subsystems. At this time, a designer knows what the requirements of the system are and how the system functions. Based on this information, the designer can separate the single process in the executable specification into several smaller processes by the major functions. The system is decomposed into smaller modules called subsystems, based on functionality. These subsystems can be the components at the next lower abstraction level or they can be at the same abstraction level. The interface and the function of each subsystem must be specified. A structural model is then developed that reflects the system decomposition. It is clear that the overall interface and the function of the system remain the same as the executable specification. The new architecture is a structural description of the system.

In our Sobel edge detection system, the process of edge detection contains the following three major steps which are executed in one clock cycle.

Step1. During each clock cycle, store the value on the **INPUT** pin in the array **MEMORY** whose size is 3 x *number_of_columns*. Starting when new data is stored in the third row, first column of the array, read out the data stored in the same column into which the WRITE operation occurs;

Step 2. Apply the four Sobel filter functions (horizontal filtering, vertical filtering, left diagonal filtering and right diagonal filtering) to the data from step 1;

Step3. Find the maximum absolute value of the four filtering outputs. Also, find the absolute value of the filtering output which is perpendicular to it. Compare the given threshold with the weighted sum of these two values. Declare a pixel to be an edge pixel if the sum is equal to or greater than the threshold.

According to these three steps, the system is naturally decomposed into three components, namely, a memory processor component that executes Step1, a window processor component that executes Step 2 and a magnitude processor component that executes Step3. The block diagram of the system decomposition is shown in Figure 3.18. Note in Figure 3.18 there are six intermediate signals. **MEM_OUT1**, **MEM_OUT2**, and **MEM_OUT3** are the signals that carry the 3-bit column of data read out in Step 1 from the memory processor component to the inputs of the window processor component. The outputs of the window processor component are connected to the inputs of the magnitude processor component by the four signals **E_H**, **E_V**, **E_DL** and **E_DR** that represent the results of the four filtering operations performed in step 2.

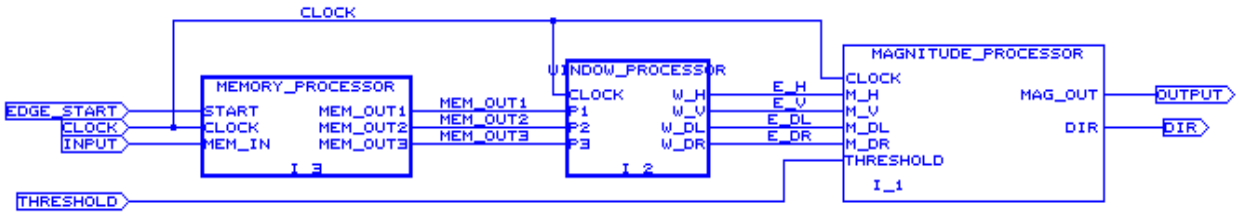



**Figure 3.18** Decomposition of the Sobel edge detection system

The corresponding VHDL skeleton code (structural model) can be generated by the SGE netlist tool. The designer can make necessary modification to it according to the application as described in Section 3.3.3.3. Figure 3.19 shows the final structural model for the edge detector.

```
 -- file name: edge_s_complete.vhd (STRUC_INT library)
-- purpose:  the structure model for the edge detector

library IEEE;
use ieee.std_logic_1164.all;
library BEH_INT;
use BEH_INT.IMAGE_PROCESSING.all;

---------------- interface declaration -------------------------------------
entity EDGE_DETECTOR is
  generic( NUM_ROWS: NATURAL;                -- the number of rows of the image file
           NUM_COLS: NATURAL);               -- the number of columns of the image file
  Port ( CLOCK : in    STD_LOGIC := '0';      -- the system CLOCK
```

```
        EDGE_ START : in  STD_LOGIC := '0';   -- the START signal for the image  processing
        INPUT : in    PIXEL:=0;               -- input image  pixel
        THRESHOLD: in FILTER_OUT:=0;          -- threshold for determine the edge pixel
        DIR : inout   DIRECTION :=0;          -- edge direction
        OUTPUT : inout   PIXEL:=0 );          -- output image  pixel
end EDGE_DETECTOR;

---------- structural description of edge detector ----------
architecture STRUCTURE of EDGE_DETECTOR is
---------- memory processor component declaration ----------
component MEMORY_PROCESS1
  generic (NUM_ROWS, NUM_COLS: NATURAL;
           MEM_OUT_DELAY:TIME)
  port(CLOCK: in STD_LOGIC:='0';                          -- system CLOCK
       START: in STD_LOGIC:='0';                                -- start signal
       MEM_IN: in PIXEL:=0;                               -- input pixel
       MEM_OUT1, MEM_OUT2, MEM_OUT3: out PIXEL:=0);   -- memory unit output
end component;
---------- window processor component declaration ----------
component WINDOW_PROCESSOR1
   generic(HORIZ_DELAY, VERT_DELAY, LEFT_DIAG_DELAY,
           RIGHT_DIAG_DELAY, WAIT_TIME:TIME)
    port(CLOCK: in STD_LOGIC:='0';        -- system CLOCK
         P1,P2,P3: in  PIXEL:=0;          -- input pixels
         W_H: inout FILTER_OUT:=0;        -- horizontal filter output
         W_V: inout FILTER_OUT:=0;        -- vertical filter output
         W_DL: inout FILTER_OUT:=0;       -- left diagonal filter output
         W_DR: inout FILTER_OUT:=0 );     -- right diagonal filter output
 end component;
---------- magnitude processor component declaration ----------
component MAG_PROCESSOR1
   generic(MAG_DELAY: TIME);                      -- magnitude processor output delay
   port(CLOCK: in STD_LOGIC:='0';                  -- system CLOCK
        M_H,M_V,M_DL,M_DR: in FILTER_OUT:=0;  -- inputs from the window processor
        THRESHOLD: in FILTER_OUT:=0;         -- the value is used to determine the edge pixel
        MAG_OUT: inout PIXEL:= 0;            -- determine edge pexel
        DIR: inout DIRECTION :="000");        -- determine direction
end component;
---------- intermediate signals between the components ----------
signal E_H,E_V,E_DL,E_DR: FILTER_OUT:=0;
signal MEM_OUT1, MEM_OUT2, MEM_OUT3: PIXEL:=0;
  begin
    ---------- component instantiation ------------------------------
```

```
   MEMP1: MEMORY_PROCESSOR1
          use entity STRUC_INT.MEMORY_PROCESSOR(BEHAVIOR);
          generic map(NUM_ROWS => NUM_ROWS, NUM_COLS => NUM_COLS,
                      MEM_OUT_DELAY => 2 ns)
          port map(CLOCK, EDGE_START, INPUT, MEM_OUT1, MEM_OUT2, MEM_OUT3);
   WINP: WINDOW_PROCESSOR1
          use entity STRUC_INT.WINDOW_PROCESS(BEHAVIOR);
          generic map(HORIZ_DELAY => 3 ns, VERT_DELAY=> 3 ns,
                      LEFT_DIAG_DELAY => 3 ns, RIGHT_DIAG_DELAY => 3 ns,
                      WAIT_TIME => 0 ns)
         port map(CLOCK, MEM_OUT1, MEM_OUT2, MEM_OUT3, E_H, E_V, E_DL, E_DR);
   MAGP: MAG_PROCESSOR1
          use entity STRUC_INT.MAG_PROCESS(BEHAVIOR);
          generic map( MAG_DELAY => 3 ns)
          port map(CLOCK, E_H, E_V, E_DL, E_DR,THRESHOLD, OUTPUT, DIR)
end STRUCTURE;
```

**Figure 3.19** VHDL code for the structural model for the Sobel edge detection system

In the architecture body, the components and the signals that connect these components are declared. Then the components are *instantiated* after the key word *begin*. Thus a specific instance of a general component entity is created. The port map creates an association between the inputs and outputs of the component declaration and the instantiated components. In this case the association is "by position": the signals in the port map are listed in the same order as the port signals in the entity declaration. One problem with *positional association* is that it is not immediately clear which signals are being connected to which ports. Someone must refer to the entity declaration to check the order of the ports in the entity interface declaration. In another approach, *named association*, each port is explicitly named along with the signal to which it is connected. The association is position independent, as shown in the following example:

```
port map(CLOCK=>CLOCK, M_H=>E_H, M_V=>E_V, M_DL=>E_DL,
         M_DR=>E_DR, THRESHOLD=>THRESHOLD, MAG_OUT=>OUTPUT,
         DIR=>DIR);
```

The subsystems are decomposed further until they reach the RTL, where the leaf behaviors correspond to VHDL data flow descriptions. As shown in Figure 3.4, the memory processor is decomposed into an address generator and a memory component, while the window processor is decomposed into a horizontal filter, a vertical filter, a left diagonal filter and a right diagonal filter. These models are at subsystem level 2.

### 3.3.5 RTL (Register Transfer Level)

At the RTL (Register Transfer Level), behavior is described by VHDL data flow descriptions [4]. Arithmetic and logical operations, such as add, subtract, and compare, are applied at this level. These operations access the data in registers and store the results back to registers. The registers are clocked memory elements.

RTL design for a particular model begins from the behavioral description of the model. Instead of using functions, hardware adders, subtractors, etc., are used. Registers are used for data storage. Synthesis tools are applied to the models at this level or to the behavioral models to obtain gate level models automatically. As an example, the following paragraph explains how the RTL horizontal filter is designed.

The design begins from the behavioral model of the horizontal filter. Figure 3.20 gives the VHDL code. The analysis result of this model is stored in the **STRUC_INT** library.

---

```
-- file name: horiz_b.vhd (STURC_INT library)
-- purpose:   horizontal filtering

library IEEE;
use ieee.std_logic_1164.all;
library STRUC_INT;
use STRUC_RTL.IMAGE_PROCESSING.all;
------------- interface declaration -----------------------------------------
entity HORIZONTAL_FILTER is
        generic(HORIZ_DELAY, :TIME);
        port ( CLOCK: in STD_LOGIC:='0';
               P1,P3: in PIXEL:=0;
               H: inout FILTER_OUT:=0);
end HORIZONTAL_FILTER;
----------- behavioral description ---------------------------------------
architecture BEHAVIOR of HORIZONTAL_FILTER is
signal TEMP1: FILTER_OUT:=0;  -- intermediate signal
begin
  H_FILTER: process
  variable TEMP_H: FILTER_OUT:=0;       -- temporary storage
  variable FIRST_LINE: PIX3:=(0,0,0);     -- a 3-stage buffer for the first scan line
  variable THIRD_LINE: PIX3:=(0,0,0);   -- a 3-stage buffer for the third scan line
  begin
    wait until rising_edge(CLOCK);
 ------------------- store the input pixels in the first 3-stage buffer ----------------------------
    FIRST_LINE :=SHIFT_LEFT(FIRST_LINE,P1);
```

```
------------------- store the input pixels in the third 3-stage buffer -----------------------------
    THIRD_LINE:=SHIFT(THIRD_LINE,P3);
    TEMP_H:= WEIGHT(FIRST_LINE(1), FIRST_LINE(2), FIRST_LINE(3))
                -WEIGHT(THIRD_LINE(1), THIRD_LINE(2), THIRD_LINE(3));
    TEMP1 <= TEMP_H after HORIZ_DELAY;
 end process H_FILTER;
---------------- make H as a delay version of temp1 --------------------------
  DELAY:  process(TEMP1)
     begin
                H <= TEMP1 after WAIT_TIME;  -- Horizontal filter output
   end process DELAY;
end behavior;
```

**Figure 3.20** VHDL code for the behavioral model of the horizontal filter

As shown in Figure 3.20, two functions, SHIFT and WEIGHT, are used in this model. The shift function stores the input data in an array. The weight function applies the horizontal Sobel operator to the input data. In the RTL model, several registers are used which replace the SHIFT_LEFT function and the corresponding arrays. It is clear from the VHDL code for the WEIGHT function (shown in Figure 3.21) that it can be implemented using adders and registers. We apply the WEIGHT functions to the data values in the first and third scan lines. The result of the first data line is then subtracted from that of the third line, and sent out as the horizontal filter output. A subtractor is used for the subtraction function.

```
---------------- Weight function ---------------------------------
function WEIGHT
  ( X1,X2,X3: PIXEL)
   return FILTER_OUT is
begin
   return X1+ 2*X2 + X3;
end WEIGHT;
```

**Figure 3.21** VHDL code for the WEIGHT function

The process DELAY is used in the behavioral model. It gives a delayed version of TEMP1 as the horizontal filter output. The reason to do this is to support *multilevel simulation* in which structural models are used for some components and behavioral models are used for others. Since the timing of the behavioral model of the filter is different from that of the structural model, delay has to be added to the behavioral model.

Based on the above analysis, we conclude that the components needed for the horizontal filter are registers, adders and subtractors. The functional block diagram of Figure 3.22 aids in

producing the SGE schematic (shown in Figure 3.23) and the VHDL model (shown in Figure 3.24 and Figure 3.25). Note that in Figure 3.24, empty component declarations and empty component instantiations are made. The generic maps, port maps and the component binding are moved to a separate configuration body shown in Figure 3.25. This coding style allows reuse of the code. We will explain it in more detail in Chapter 4. In this chapter, we are concentrating on the design path through the abstraction hierarchy.

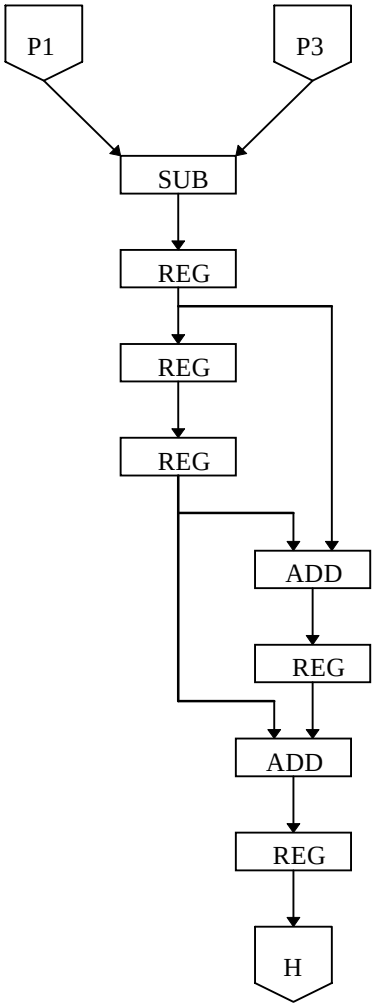



**Figure 3.22** Functional block diagram of the horizontal filter

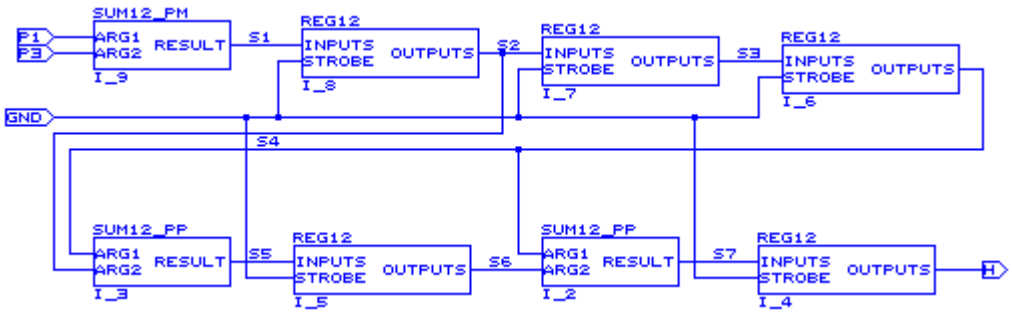



**Figure 3.23** SGE schematic for the RTL horizontal filter

```
-- file name: horiz_rtl.vhd
-- purpose:   horizontal filtering
-- library:   STRUC_RTL library

library IEEE;
use IEEE.STD_LOGIC_1164.all;

library STRUC_RTL;
use STRUC_RTL.IMAGE_PROCESSING.all;
use STRUC_RTL.all;

------------- interface declaration ----------------------------------------
entity HORIZONTAL_FILTER is
        generic(HORIZ_DELAY:TIME);
        port ( CLOCK: in STD_LOGIC:='0';
               P1,P3: in PIXEL:="00000000";
               H: inout STD_LOGIC_VECTOR(11 downto 0));
end HORIZONTAL_FILTER;

----------- structural description ---------------------------------
architecture STRUCTURE of HORIZONTAL_FILTER is

  ------ an empty 12-bit Full_Subtractor component declaration ---
  component SUM12_PM
  end component;

  ------ an empty 12-bit Register component declaration ----------
  component REG12
  end component;
```

```
   ------ an empty 12-bit Full_Adder component declaration ----------
   component SUM12_PP
   end component;

   ----- an empty 8-bit to 12 bit extendor ----------------------
   component EXT8_12
   end component;

   ----- an empty 12-bit to 12 bit multiplier -----------
   component MULT_2_12_12
   end component;

   ------ intermediate signal declarations ---------------------------
   signal S1, S2, S3, S4, S5, S6, S7 : STD_LOGIC_VECTOR(11 downto 0);
   signal A, B, C : STD_LOGIC_VECTOR(11 downto 0);

begin

   ----- empty component instantiations -------------------
   EXT8_12A: EXT8_12;
   EXT8_12B: EXT8_12;
   DIFF1  : SUM12_PM;
   DELAY1 : REG12;
   DELAY2 : REG12;
   DELAY3 : REG12;
   SUM1   : SUM12_PP;
   DELAY4 : REG12;
   MULT_2_12_12A: MULT_2_12_12;
   SUM2   : SUM12_PP;
   DELAY5 : REG12;
end STRUCTURE;
```

**Figure 3.24** RTL model of the horizontal filter

```
-- file name: config_h_rtl
--purpose: configuration body for the horizontal filter
-- library: STRUC_RTL

library STRUC_RTL;
use STRUC_RTL.all;
```

```
configuration H_RTL of HORIZONTAL_FILTER is
        for STRUCTURE
                 ------------------ A <= "0000" & P1 ----------------
                for EXT8_12A: EXT8_12 use entity STRUC_RTL.EXT8_12(BEHAVIOR)
                        port map(P1,A);
                end for;

                ------------------ B <= "0000" & P3 ----------------
                for EXT8_12B: EXT8_12 use entity STRUC_RTL.EXT8_12(BEHAVIOR)
                        port map(P3,B);
                end for;

                ------------------ S1 <= A - B ----------------
                for Diff1:  SUM12_PM use entity STRUC_RTL.SUM12_PM(PARTS)
                        port map(A,B,S1);
                end for;

                 ------------------ S2 <= S1 ----------------
                for DELAY1: REG12 use entity STRUC_RTL.REG12(PARTS)
                        generic map(HORIZ_DELAY)
                        port map( S1, S2, CLOCK );
                end for;

                ------------------ S3 <= S2 ----------------
                for DELAY2: REG12 use entity STRUC_RTL.REG12(PARTS)
                        generic map(HORIZ_DELAY)
                        port map( S2, S3, CLOCK );
                end for;

                ------------------ S4 <= S3 ----------------
                for DELAY3: REG12 use entity STRUC_RTL.REG12(PARTS)
                        generic map(HORIZ_DELAY)
                        port map( S3, S4, CLOCK );
                end for;

                ------------------ S5 <= S4 + S2 ----------------
                for SUM1:   SUM12_PP use entity STRUC_RTL.SUM12_PP(PARTS)
                        port map( S4, S2, S5 );
                end for;

                ------------------ S6 <= S5 ----------------
                for DELAY4: REG12 use entity STRUC_RTL.REG12(PARTS)
                        generic map(HORIZ_DELAY)
```

```
                port map( S5, S6, CLOCK );
        end for;

        ------------------ C <= S4(10 downto 0) & '0' ----------------
        for MULT_2_12_12A: MULT_2_12_12
                use entity STRUC_RTL.MULT_2_12_12(BEHAVIOR)
                port map(S4,C);
        end for;

        ------------------ S7 <= C + S6 ----------------
        for SUM2:   SUM12_PP use entity STRUC_RTL.SUM12_PP(PARTS)
                port map(C, S6, S7);
        end for;

        ------------------ H <= S7 ----------------
        for DELAY5: REG12 use entity STRUC_RTL.REG12(PARTS)
                generic map(HORIZ_DELAY)
                port map( S7, H, CLOCK );
        end for;
    end for;
end;
```

**Figure 3.25** Configuration body for the RTL horizontal filter

### 3.3.6 Gate Level Synthesis

The gate level is the lowest abstraction level at which a model is typically represented in VHDL. The models at the RTL can be transformed into gate level circuits individually which are then wired together to complete a design that performs the function of the system. This translation can be done manually or by use of synthesis tools. One motivation for VHDL modeling is to allow automatic synthesis of circuits. This section discusses the Synopsys synthesis tool. The horizontal filter is used as a case study.

At the RTL, a design is represented by a circuit consisting of components, such as adders, subtractors and registers. Packages, if any, and VHDL code for the RTL components are analyzed and then read by the synthesis tool. Next, the structural model of the system is read. The synthesis tool generates a block first which is then mapped to a particular library to generate a gate level circuit. Figure 3.26 shows the gate level circuit generated by the Synopsys synthesis tool *design_analyzer*. The VHDL files read in are shown in Figure 3.24 and Figure 3.25.

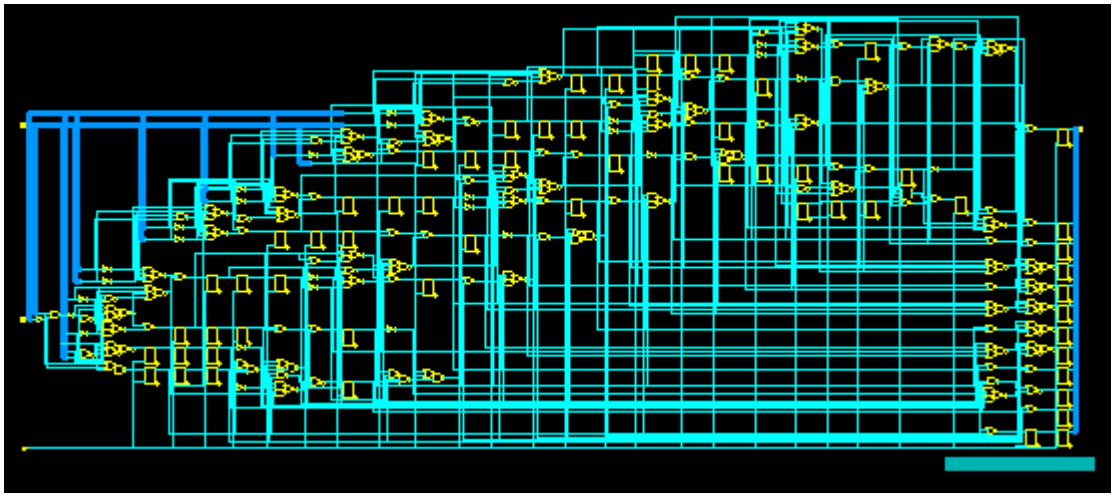


**Figure 3.26** Gate level circuit of the horizontal filter

Once the RTL model has been synthesized, the next step is to optimize the circuit. The typical optimization methodology is to optimize for area first and then to optimize for timing. Hierarchical blocks in a large design are normally optimized starting from the lower level blocks in a bottom up process. The synthesis tools carry out the design optimization according to user defined constraints. Current synthesis tools typically allow constraints to be set for minimal area and minimal timing delay.

The optimization processes use different algorithms on a trial and error basis to find a circuit implementation that best fits the constraints. A circuit is optimized for minimum area based on what the optimizer can find. This may not always be the absolute minimum circuit that could be produced if the design were carefully designed by hand.

Constraints represent desired circuit characteristics that are specified as part of the design goals. Different constraints cause different optimized circuits to be generated, but with the same functionality. The most common constraints used today are area and timing.

***AREA***: An area constraint is a number corresponding to the desired maximum area of a specified design module. The area number will have units corresponding to the units defined in the cells technology library, e.g. equivalent gates, grids or transistors.

***TIMING***: Timing constraints put limits on the maximum delay from any input to any output. The static timing analyzer in the synthesis tool will extract timing information in order to compute actual paths delays. This includes setup time and hold time of registered elements and signal delays through combinational logic given specific global constraints. Signal path delays in the model are computed and compared with desired timing constraints, whereby automatic optimization is performed as necessary in order to improve timing characteristics. Typically any

designer will progressively increase timing optimization effort levels to progressively trade off area for improved timing.

To illustrate, apply area and timing optimization to the gate level horizontal filter shown in Figure 3.26. Initially, the maximum area is specified as 0, and the clock period is specified as 50 ns with 1 ns skew. By attempting to reach 0 area, the system will obtain the minimum area possible for this design. Then we gradually reduce the clock period from 50 ns to 9 ns and repeat the optimization. The optimization can be repeated until a design violation occurs.

A report can be generated. Table 3.1 gives the optimization results. As one decreases the clock period, one may get a faster circuit with more circuit area and more power consumption. There always exists a trade-off among the different designs. One can decide which circuit to choose according to ones needs. An environment for performing trade-off studies is developed in Chapter 4.

**Table 3.1** Optimization results of the horizontal filter

| Circuit No. | Max Area Specified | Clock Period Specified (ns) | Clock Skew (ns) | Area Consumed | Slack (clock_period - critical path delay ) (ns) | Power consumed ($\mu W$) |
|---|---|---|---|---|---|---|
| 1 | not specified | not specified | not specified | 706 | not specified | 276.24 |
| 2 | 0 | 50 | 1 | 706 | 29.02 | 13.86 |
| 3 | 0 | 31 | 1 | 712 | 10.53 | 21.87 |
| 4 | 0 | 21 | 1 | 709 | 0.81 | 31.94 |
| 5 | 0 | 18 | 1 | 712 | 0.3 | 38.07 |
| 6 | 0 | 14 | 1 | 740 | 0.05 | 52.44 |
| 7 | 0 | 11 | 1 | 751 | 0.02 | 66.69 |
| 8 | 0 | 9 | 1 | 823 | 0.01 | 98.01 |

Note that the *Max Area* is always specified as 0. Hence the synthesis tool will always generate a circuit with the minimum area for the given timing constraints.

# Chapter 4. Design Techniques

## 4.1 Introduction

Making a design functionally correct is the basic design goal. But this is not enough. It is also important to do it in an efficient way. Good design styles and techniques make a design easy to maintain and save design time and design cost.

Some design issues include the following: how to easily make changes to the design; how to design a system that can be simulated efficiently; and how to develop models that can be easily maintained and reused in future systems. Sometimes, there are several people in a design team. It is important to be able to share the design information efficiently. This chapter discusses some efficient design techniques and related concepts.

## 4.2 Modular Design

A design unit can be represented by a block, as shown in Figure 3.8. It can be treated as a black box whose interfaces are visible to the outside world and whose design detail is hidden. A higher level user can make use of its function without knowing the design detail. As design progresses, the design unit is decomposed into sub-blocks, as shown in Figure 3.4. Each block/sub-block is actually a separate module that can be reused in other systems.

One benefit of modular design is that it allows concurrent design. When a module is decomposed into several sub-modules, the interfaces and the specifications for the sub-modules are defined. A high level structural model of the system can be produced and simulated to verify that the structural model meets the system specification. Different designers can work on those sub-modules separately at the same time. They can work on their own piece without waiting until someone else's design is completed. As the detailed design for each sub-module is completed, a new detailed model can be substituted for the original higher level model. A new simulation will verify that the system still performs correctly with the detailed component in place. Again, it is not necessary to wait for other sub-modules to be completely designed and a partial tree simulation (discussed in Chapter 3) will be applied to the system. After all the sub-modules are

completed, they can be inserted into a system to check if they are compatible and can work together to meet the system functional requirements.

Another benefit is that each module is easily replaceable. Often, the system requirements are changed a little bit after detailed design is finished. Modular design allows redesigning a sub-module related to that change instead of redoing the whole system design. The only thing that a designer needs to consider is to keep the interface of the new design the same as the original one. After the new design is done, it can be inserted into the system and tested without changing any other sub-modules or interconnections between them.

The importance and benefits of modular design are obvious. Now we use the Sobel edge detection system to illustrate some tips on how to do modular design in a VHDL based system.

1. Make use of top-down design

As illustrated in Chapter 3, we can gradually decompose the system to multiple abstraction levels. As shown in Figure 3.4, the Sobel edge detector system is first decomposed into three components, namely, the memory processor, the window processor and the magnitude processor. The interface and the specifications for each component is generated as part of the decomposition process. A behavioral model for each component is developed and tested separately. Then the components are connected to form a structural system which is tested against the original executable specification. If differences exist, the interface and function of one or more components are modified until they work. Then the three processors are decomposed further. The memory processor is partitioned into an address generator and a memory chip, while the window processor is partitioned into four filters. The magnitude processor is not decomposed further at this level. The design and decomposition cycle repeats until the circuit has enough detail to be fabricated. By using top-down design, one system is decomposed into smaller independent components. These components can be designed, tested and replaced separately. Thus we achieve modular design.

2. Good Documentation and Coding Style

Good documentation and coding style make the VHDL code readable and self-explainable, and support modular design, reuse, debugging and modification as well.

There may be large numbers of variables, signals and functions used in the same design. Giving them meaningful names makes the program easy to follow and understand.

Figure 4.1 shows the entity interface declaration of a memory chip. Instead of naming the entity, ports and generics with A, B, C,..., we give them meaningful names. For example, the entity name is given as MEM_CHIP. Thus we have a clue that this is a memory chip. Port name DATA indicates that it is an I/O port.

```
entity MEM_CHIP is
  generic(ENABLE_DELAY: TIME;          -- ENABLE to DATA  delay
          RWB_DELAY: TIME;             -- RWB to write completion delay
          ADDR_DELAY:TIME;             -- ADDRESS to DATA delay
          WRITE_SETUP: TIME;           -- set up time for the write operation
          NUM_COLS: NATURAL);          -- number of columns of the input image file
  port (RWB: in STD_LOGIC:='0';        -- READ/WRITE signal
        Y_ADDR : in STD_LOGIC_VECTOR(7 downto 0):="00000000";  -- Y address
        ENABLE: in STD_LOGIC:='0';                             -- memory chip enable
        DATA : inout PIXEL);                                   -- data in/out port
end MEM_CHIP;
```

**Figure 4.1** Interface declaration of the memory chip

Some code segments are used in more than one place. They can be written as a function or procedure and stored in a package. By making the package visible, other programs can simply call them and make use of them. Thus, these code segments can be shared by many programs. For example, the weight function is declared once in the IMAGE PROCESSING package, as shown in Figure 3.13. It is used by all four of the filters.

VHDL is a strongly typed language. Any signal, variable, constant, etc., must be assigned a type before it can be used. Once declared, a name is generally visible only from the point of declaration until the end of the declarative region within which it is declared. If a data type is to be used by more than one entity within a design, it is tedious to do the declaration in every entity. A package can be used to collect commonly used declarations which are then made visible to every design entity. In the IMAGE PROCESSING package of the Sobel edge detection system shown in Figure 3.13, we declared several data types, such as *PIXEL*, *FILTER_OUT*, etc. These data types are used by many components in our design.

### 4.3 Maintaining, Sharing and Reuse of a Design

The complexity of hardware designs has greatly increased over the past number of years. With the progress of VLSI technology, it is not uncommon for millions of gates to be placed on one single chip. Usually, a work team instead of one single person works on a design project. An efficient way should be found to maintain and share the design information among the team members.

Also, it is rare that a particular design is unmodified in its life time. As the design progresses, it is continually tested to verify that its function meets the design requirements. As the design is being verified and used, designers and users may have some idea to improve the system, such as adding more functionality to the system, applying new technology to some

components, reducing the circuit area and/or power consumption, etc. Thus, a design often progresses in a growing and iterative fashion.

Nowadays, people care about not only the design quality, but also about the *time to market*, and the *life cycle cost*. Modifying and upgrading a design does not mean that a design has to be redone from the beginning. It is based on the previous designs with some improvements. Even for a new design, we can make use of some existing components or some individual parts of the previous design. Reuse of design data keeps us from reinventing when a new design problem comes up. If the previous designs can be reused as much as possible, precious design time and design cost are saved.

Two questions arise: 1) how to manage the design data so that it can be easily reused and 2) how to apply the concept of reuse to reduce the cost of the new simulations. The following sections use the Sobel edge detection system to illustrate approaches to answering these questions.

### 4.3.1 Manage the VHDL code and Make Use of Package

As illustrated in Section 4.2, a package collects frequently used data types, functions, procedures, etc. Other models can use these declarations by making the package visible to them. Thus information is shared by one or more designs. For example, the data types PIXEL and FILTER_OUT, and the function WEIGHT, once declared, can be reused by many components.

The design of any component is possibly changed over its life time. Sometimes, a component would not even exist because of the change of the system architecture. But some code segments might be reusable for other designs. Although a simple "*copy and paste*" helps to reuse the code segments, there is a better way to do it. If we write frequently used code segments as functions or procedures and store them in a package, other design can use these code segments by simply calling the functions or procedures.

### 4.3.2 Sharing and Reusing an Existing Design

Sometimes, changes and improvements need to be made to one or more existing models/subsystem in a system. Reasons for changes might be to apply new technology, to improve the new algorithm or architecture, to expand the functionality, to improve performance and form factors (size, weight, area and power consumption) of a system, or for other reasons. It is not cost effective to redo all the designs from the beginning. The advantage of reusing an existing design is that it can save design time and cost without reducing the quality of the design. By reusing, changes are made only when necessary. Most or at least a portion of the original design is kept unchanged. In this way, we can reduce the degree of redesign to a minimum. Sometimes, reuse can be applied to a brand new design. Some components are frequently used by different designs. Once designed, they can be reused by other designs.

The following sections explain some reuse techniques.

#### 4.3.2.1 Library

When a VHDL model is successfully analyzed, the result is stored in a *library*. We can have various design libraries which store various stages or various versions of a design. In the Sobel edge detection system, as the design evolves from high abstraction levels to low abstraction levels, the data types used for image data and intermediate data are changed from INTEGER to STD_LOGIC_VECTOR. Basically, we have five design phases: executable specification phase (INTEGER data type), architecture design phase (INTEGER data type), detailed architecture design phase (STD_LOGIC_VECTOR data type), RTL design phase (STD_LOGIC_VECTOR data type) and gate level design phase (STD_LOGIC_VECTOR data type). Five resource libraries are created: **BEH_INT**, **STRUC_INT**, **STRUC_STD**, **STRUC_RTL** and **STRUC_GATE**. The analysis results of the models are stored in one of the five libraries according to the design phase they belong to.

Usually, a VHDL analyzer will allow the use of any number of libraries and will allow selection of any model in a particular library. The existence of these libraries allows the VHDL models to be reused in the future. For example, a clock generator is analyzed and the result is stored in the **BEH_INT** library. The analysis result can be reused by the models in other designs. The following statements make the clock generator visible by other libraries:

```
library BEH_INT;
use BEH_INT.CLOCK_GENERATOR;
```

Each VHDL design library has three names: a *logical* name, a *library* name, and a *physical* name. These names are used in different contexts, as follows:

- The logical name is the VHDL identifier referring to the library in VHDL code (with *library* and *use* statements). This name is virtual and “portable”.
- The library name is used internally by VSS (VHDL Synopsys Simulator) Family programs. The logical name can be mapped to a host system directory via the library name.
- The physical name is the path name of the host directory that actually contains the analyzed design files [9].

Figure 4.2 diagrams the design library naming levels, from VHDL source code references to an operating system directory.

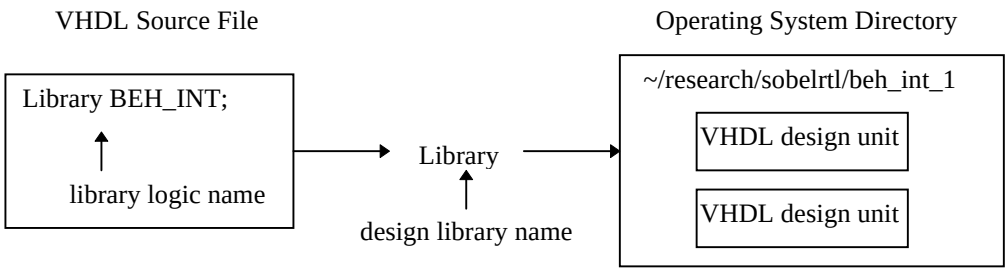



**Figure 4.2** VHDL design library concept [9]

Each VHDL implementation is required to support the use of two library logical names: **WORK** and **STD**. The **WORK** library stores all the current analysis results that can be used for future analysis and simulation purposes. The **STD** library is defined by the VHDL language. It contains two packages, **STANDARD** and **TEXTIO**. The **STANDARD** packages contains some fundamental data types, such as *BIT*, *BIT_VECTOR*, *SEVERITY_LEVEL* type, etc. The **TEXTIO** package contains declarations for types and subprograms that support ASCII I/O operations and the file subsystem of VHDL [10].

Besides these two logic libraries, we can have some other logic libraries in which analysis results are stored. These libraries and the **STD** library are called *resource* libraries. The resource libraries can be referenced during analysis and simulation, but cannot be written into [4]. However, we can temporarily designate a resource library as the **WORK** library. It is in the setup file that we can define the logic libraries and host libraries, the mapping between them, as well as the definition of the **WORK** library.

The project directory for the Sobel edge detection system is *~/research/sobelrtl*. In order to support the Synosys simulator, a setup file is stored in this directory. Figure 4.3 shows a portion of this file. **WORK** and **BEH_INT** are logic libraries, whose design library names are **DEFAULT** and **BEH_INT_HOST**, respectively. Two host directories, *./vhdlwork* and *~/research/sobelrtl/beh_int_1* are created to which the logic libraries are mapped. The statements in Figure 4.3 shows the mapping between the libraries and host directories. For example, the logic library **WORK** is mapped to design library **DEFAULT** which is then mapped to a host directory *./vhdlwork*.

```
WORK          > DEFAULT
DEFAULT       : ./vhdlwork
BEH_INT        > BEH_INT_HOST
BEH_INT_HOST  : ~/research/sobelrtl/beh_int_1
```

**Figure 4.3** A portion of the *.synopsys_vss.setup* file

Sometimes we may want to declare **BEH_INT** as the **WORK** library. One can either change the above statements as:

*WORK* *> BEH_INT_HOST*
*BEH_INT_HOST* *: ~/research/sobelrtl/beh_int_1*

or temporarily make **BEH_INT** the "**WORK**" library by invoking the analyzer as:

vhdlan -w BEH_INT horiz_b.vhd.

The analysis result for file horiz_b.vhd is stored in the **BEH_INT** library. The second method is better than the first one because it dynamically assigns the **WORK** library without changing the setup file.

The design library can be changed by changing the setup file. For example, if we want to change the design library BEH_INT_HOST in Figure 4.3 to BEH_INT_ CHANGED, we would change the setup file as shown in the following figure:

---

*WORK* *> DEFAULT*
*DEFAULT* *: ./vhdlwork*

*BEH_INT* *> **BEH_INT_CHANGED***
***BEH_INT_CHANGED*** *: ~/research/sobelrtl/**beh_int_changed***

---

**Figure 4.4** Changing the design library

Figure 4.4 shows an easy way to change the mapping among the logic library, design library and the host directory. The method enables a designer to easily change a design library without changing all the references in the VHDL code.

Storing the models for different design phases in different libraries is a good way to manage the design information during the life time of the design. A designer can easily reuse an existing design by making use of the library.

#### 4.3.2.2 Generics

Generics are frequently used in design of the VHDL models to specify constants. We can write a generic entity by including a generic interface list in the interface declaration of the entity. The actual values of the generics are not given in the architecture body. They are given later when the entities are used. The following Figure 4.5 and Figure 4.6 are the VHDL code of a 2-input OR gate with and without using the generic for delay time, respectively.

```
entity OR2 is
   port(I0,I1: in STD_LOGIC:='0';
        O   : out STD_LOGIC:='0');
end OR2;

architecture BEHAVIOR of OR2 is
begin
       O <= I0 or I1 after 2 ns;      -- actual value is used to specify the gate delay
end BEHAVIOR;
```

**Figure 4.5** VHDL code for a 2-input OR gate using actual value for the delay time

```
entity OR2 is
   generic(OR_DELAY: TIME);        -- the generic or_delay is used to define the gate delay
   port(I0,I1: in STD_LOGIC:='0';
        O   : out STD_LOGIC:='0');
end Or2;

architecture BEHAVIOR of OR2 is
begin
       O <= I0 or I1 after OR_DELAY;
end BEHAVIOR;
```

**Figure 4.6** VHDL code for a 2-input OR gate using a generic for the delay time

If one chooses to use an actual value to specify the constants as in Figure 4.5, one has to change the architecture code to change the value of the constants. Thus, the model developed in this way is a specific model, but not a general one. One has to specify the constants in every component used and also must re-analyze the code in order to change the value of the constants. The method is not practical, error-prone and not good for design reuse. Figure 4.6 shows a general way to design a VHDL model. The actual value of the constants are not given when it is being designed. We can specify the values by configuration when the entities are used. The benefit of using generics is that it allows parametizing the design entities and allows reusing the existing designs without having to re-analyze the design every time a value is changed.

#### 4.3.2.3 Coding Style

Coding style is important in designing VHDL models. Good style not only improves the readability and manageability, but also facilitates reuse of the code.

One design entity can be implemented as either a behavioral model or as a structural model. It is not uncommon for the generic list to be different for the two models with the same port declarations. In the behavioral model, we describe the algorithm or function of the design entity. As we do detailed design, we obtain the structural model which functions the same as the behavioral model. In most cases, the generic list is longer for the structural model than for the behavioral model. The example given in Figure 4.7 is the interface declaration of the structural model of the memory processor. This model consists of several components, such as an address generator, three memory chips, three buffers, etc. It re-declares every generic from the lower level models. Figure 4.8 gives the interface declaration of the behavioral model. It is obvious that these two models have the same port declarations, but different generic lists.

```
entity MEMORY_PROCESSOR  is
generic(ADDR_A_DELAY, RWB_A_DELAY, ENABLE_DELAY, RWB_M_DELAY,
         ADDR_M_DELAY, WRITE_SETUP,  DECODER_DELAY, BUFF_DELAY,
         OR_DELAY,MUX_DELAY, REG_DELAY,NOT_DELAY:TIME;
        NUM_ROWS,NUM_COLS:NATURAL;
        N:INTEGER);
port(CLOCK: in STD_LOGIC:='0';  -- system CLOCK
      START: in STD_LOGIC:='0';  -- start signal
      MEM_IN: in PIXEL:=0;       -- input pixel
      MEM_OUT1, MEM_OUT2, MEM_OUT3: out PIXEL:=0);
                                          --memory unit output
end MEMORY_PROCESSOR;
```

**Figure 4.7** Interface declaration of the structural model of the memory processor

```
entity MEMORY_PROCESSOR is
generic(NUM_ROWS, NUM_COLS:NATURAL; MEM_OUT_DELAY:TIME);
port(CLOCK: in STD_LOGIC:='0'; -- system CLOCK
      START: in STD_LOGIC:='0';  -- start signal
      MEM_IN: in PIXEL:=0;       -- input pixel
     MEM_OUT1, MEM_OUT2, MEM_OUT3: out PIXEL:=0); -- memory unit output
end MEMORY_PROCESSOR;
```

**Figure 4.8** Interface declaration of the behavioral model of the memory processor

Both the behavioral model and the structural model can be treated as components and inserted into a higher level structural model. The component declaration in the higher level structural model can be complete in which case the generic list and the port list are declared.

Figure 4.9 gives the component declaration corresponding to the interface declaration in Figure 4.8, with a complete generic list and port list.

```
component MEMORY_PROCESSOR1
generic(NUM_ROWS, NUM_COLS:NATURAL; MEM_OUT_DELAY:TIME);
port(CLOCK: in STD_LOGIC:='0'; -- system CLOCK
     START: in STD_LOGIC:='0';  -- start signal
     MEM_IN: in PIXEL:=0;       -- input pixel
    MEM_OUT1, MEM_OUT2, MEM_OUT3: out PIXEL:=0); -- memory unit output
end component;
```

**Figure 4.9** Component declaration of the memory processor corresponding to Figure 4.8

There is however one shortcoming of this method. The component declared is constrained by its own definition. Models that are instantiated based on the above declaration must have the same interface declaration as the component. The component declaration part of the structure model would have to be changed if we choose design units with different interface declarations. In order to achieve code reuse, we need to develop a structural model that is suited for both the behavioral and the structural declaration.

There is another reason why we need a general component declaration. The data types of the interface signals for a design entity are different at different levels of abstraction. For example, we use INTEGER data types for the interface signals at the executable specification level and use STD_LOGIC data types for the interface signals at the RTL level. To do multiple level simulation or mixed data type simulation (discussed in section 4.4), we might choose components of various data types according to the application.

Figure 4.10 gives a general component declaration of the memory processor.

```
component MEMORY_PROCESSOR1
end component;
```

**Figure 4.10** A general component declaration for the memory processor corresponding to Figure 4.8

We use this method to develop a general structural architecture for the Sobel edge detection system as shown in Figure 4.11. This structural model is similar to the one shown in Figure 3.19, but has some statements deleted. The missing parts are: 1) the generic list and the port list in the component declaration regions, 2) the component binding, 3) the mapping for the generics and the ports. Thus, the model shown in Figure 4.11 is actually a skeleton of the model shown in Figure 3.19. The actual value of the generics and the port mappings are provided in a

separate *configuration body* shown in Figure 4.12. Also, the component bindings are completed in the configuration body. The next section will explain how to develop a configuration body. This general model allows us to define a variety of mixed level architectures by declaring a separate configuration body for each architecture without changing or re-analyzing the original model. And, the design time and design cost are reduced because a single general model is used for different architectures.

---

```
 entity EDGE_DETECTOR is
    generic(NUM_ROWS,NUM_COLS:NATURAL);  -- parameters for the image size
    port ( CLOCK: in STD_ULOGIC;                    -- system CLOCK
           EDGE_START : in STD_LOGIC:='0';          -- start signal for edge detection
           INPUT: in PIXEL:=0;                      -- input pixel
           THRESHOLD: in FILTER_OUT:=0;             -- threshold for the edge pixel
           DIR  : inout DIRECTION;                  -- output direction
           OUTPUT: inout PIXEL:=0);                         -- output edge pixel
end EDGE_DETECTOR;

architecture STRUCTURE of EDGE_DETECTOR is

component MEM_SYS1
----------------------------------delete the generic list and  port list ------------------------------------
 --generic (NUM_ROWS, NUM_ROWS, NUM_COLS, MEM_OUT_DELAY:TIME)
 -- port(CLOCK: in STD_LOGIC:='0';                                  -- system CLOCK
 --       START: in STD_LOGIC:='0';                                 -- start signal
 --       MEM_IN: in PIXEL:=0;                                      -- input pixel
 --       MEM_OUT1, MEM_OUT2, MEM_OUT3: out PIXEL:=0); -- memory unit output
-------------------------------------------------------------------------------------------------------------
end component;

------------ window processor component declaration -------------------
component WINDOW_PROCESSOR1
----------------------------------delete the generic list and  port list ------------------------------------
-- generic(HORIZ_DELAY, VERT_DELAY, LEFT_DIAG_DELAY,
--           RIGHT_DIAG_DELAY, WAIT_TIME:TIME)
-- port(CLOCK: in STD_LOGIC:='0';          -- system CLOCK
--       P1,P2,P3: in  PIXEL:=0;           -- input pixels
--          W_H: inout FILTER_OUT:=0;      -- horizontal filter output
--          W_V: inout FILTER_OUT:=0;      -- vertical filter output
--          W_DL: inout FILTER_OUT:=0;      -- left diagnal filter output
--          W_DR: inout FILTER_OUT:=0 );    -- right diagnal filter output
-------------------------------------------------------------------------------------------------------------
 end component;
```

```
------------ magnitude processor component declaration ----------------
component MAG_PROCESSOR1
---------------------------------- delete the generic list and  port list ------------------------------------
--   generic(MAG_DELAY: TIME);
--   port(CLOCK: in STD_LOGIC:='0';                          -- system CLOCK
--         M_H,M_V,M_DL,M_DR: in FILTER_OUT:=0;  -- inputs from the window processor
--         THRESHOLD: in FILTER_OUT:=0;         -- the value is used to determine the edge pixel
--         MAG_OUT: inout PIXEL:= 0;                -- determine edge pexel
--         DIR: inout DIRECTION :=EAST);            -- determine direction
-------------------------------------------------------------------------------------------------------------------
end component;

------------ intermediate signal between the components ---------------------
signal E_H,E_V,E_DL,E_DR: FILTER_OUT:=0;
signal MEM_OUT1, MEM_OUT2, MEM_OUT3: PIXEL:=0;

  begin

    ---------- component instantiation ------------------------------------------------------------------
    MEMP1: MEM_SYS1;
---------------------------------- delete configuration statements  ----------------------------------------
--           use entity STRUC_INT.MEM_SYS(BEHAVIOR);
--           generic map(NUM_ROWS => NUM_ROWS, NUM_COLS => NUM_COLS,
--                          MEM_OUT_DELAY => 2 ns)
--           port map(CLOCK, EDGE_START, INPUT, MEM_OUT1, MEM_OUT2, MEM_OUT3);
-------------------------------------------------------------------------------------------------------------

    WINP: WINDOW_PROCESSOR1;
---------------------------------- delete configuration statements  ----------------------------------------
--         use entity STRUC_INT.WINDOW_PROCESS(BEHAVIOR);
--           generic map(HORIZ_DELAY => 3 ns, VERT_DELAY=> 3 ns,
--                          LEFT_DIAG_DELAY => 3 ns, RIGHT_DIAG_DELAY => 3 ns;
--                          WAIT_TIME => 0 ns)
--         port map(CLOCK, MEM_OUT1, MEM_OUT2, MEM_OUT3, E_H, E_V, E_DL, E_DR);
-------------------------------------------------------------------------------------------------------------

MAGP: MAG_PROCESSOR1;
---------------------------------- delete configuration statements  ----------------------------------------
--          use entity STRUC_INT.MAG_PROCESS(BEHAVIOR);
--           generic map( MAG_DELAY => 3 ns)
--           port map(CLOCK, E_H, E_V, E_DL, E_DR,THRESHOLD, OUTPUT, DIR)
-------------------------------------------------------------------------------------------------------------
```

```
end STRUCTURE;
```

**Figure 4.11** The structural model of the Sobel Edge Detector

```
configuration EDGE_STRUC of EDGE_DETECTOR is
   generic map(NUM_ROWS=>10,NUM_COLS=>10);
      for STRUCTURE
         for MEMP1:MEMORY_PROCESSOR1
                use entity STRUC_INT.MEMORY_PROCESSOR(BEHAVIOR)
                 generic map(NUM_ROWS=>NUM_ROWS, NUM_COLS=>NUM_COLS,
                             MEM_OUT_DELAY=> 2 ns)
                  port map (CLOCK, EDGE_START, INPUT, MEM_OUT1, MEM_OUT2,
                            MEM_OUT3);
         end for;
         for WINP: WINDOW_PROCESSOR1
                use entity STRUC_INT.WINDOW_PROCESSOR(BEHAVIOR)
                generic map(HORIZ_DELAY => 3 ns, VERT_DELAY=> 3 ns,
                            LEFT_DIAG_DELAY => 3 ns, RIGHT_DIAG_DELAY => 3 ns,
                            WAIT_TIME => 0 ns)
                port map(CLOCK, MEM_OUT1, MEM_OUT2, MEM_OUT3, E_H, E_V, E_DL,
                         E_DR);
         end for;
         for MagP: MAG_PROCESSOR1
                use entity STRUC_INT.MAG_PROCESSOR(BEHAVIOR)
                generic map( MAG_DELAY => 3 ns)
                port map (CLOCK, E_H, E_V, E_DL, E_DR, THRESHOLD, OUTPUT, DIR);
         end for;
      end for;
   end;
```

**Figure 4.12** The configuration body for the structural model shown in Figure 4.11

#### 4.3.2.4 Configuration Body

The structural architecture of one level of a design is developed in such a way that the component models are first declared and then instantiated. Sometimes, the interconnections between the components and the actual generic values are also specified in the structural architecture. To support simulation or synthesis, we also need to give the implementation for each component instance; that is, a real entity interface and corresponding architecture body should be *bound* for each of the component instances. This can be done by writing configuration specification statements for the design.

The configuration specification statements can be written in two places: the structural architecture body or a separate configuration body. For example, the Sobel edge detection system shown in Figure 3.19 includes the configuration specification in its architecture body. Figure 4.11 and Figure 4.12 show another way to describe the Sobel edge detection system where a separate configuration body is used.

The simulation results are the same no matter where we write the configuration specification statements as long as we give the same design entity to bind the component. However, the separate configuration body allows a later component binding without re-analyzing any part of the design itself. Thus the design entities are of a general form and the specific parameters are given in the configuration body. Component binding can be done in an easy and convenient way.

In the example shown in Figure 3.19, the generic maps, the port maps and the configurations are written in the architecture body. If one decides to choose another design unit, one has to change the architecture code. Besides, this changed model must be re-analyzed before any simulation can be done. Further, any model that uses this model as a component must also be re-analyzed.

Figure 4.11 and Figure 4.12 shows a more efficient and powerful way to map the real design unit to the component. In this example, the generic maps, port maps and configuration statements are written in a separate configuration body. The dependence between the models are less so that the number of models that need to be re-analyzed is reduced to a minimum. In other words, only the models which are changed and the configuration body must be re-analyzed prior to simulation.

Making use of configuration bodys actually addresses a fundamental design issue: how to reconstruct the full design once the various parts defined by the system decomposition have been completely designed. This reconstruction is necessary for various purposes, such as simulation, testing, delivery, etc. The configuration body provides a powerful way to reconstruct a system without changing any model design or re-analyzing any part of the design. The existing design is fully reused, and both the design time and simulation time are saved.

In the configuration body, generic maps, port maps and the component binding can be changed freely. This advantage allows us to freely choose different parameters for a component, components in different design libraries or a different architecture for the system. A single general design can be easily configured differently to get different implementations of the system.

## 4.4 Multiple Level and Mixed Data Type Simulation

### 4.4.1 Multiple Level Simulation

It is a good engineering practice to verify each component separately first, then verify the system. From the simulation point of view, we want to verify a design with minimum simulation time. Simulating a system with all the components at the lowest abstraction level is both time consuming and often not practical. We may have a situation that the simulator can not work well because the circuit size is too big.

What we usually should do is: verify the components one by one and apply *multiple level simulation*. That is, we choose the lowest abstraction level for the component under test and the highest level for the other components. In this way, we can achieve minimum time simulation. This method is also suited for the case when we want to verify a component when the other components are still under design.

Again, the configuration body plays an important role here. By changing the configuration body, we can freely choose different designs, different components or system parameters or different architectures. Obviously, we can also choose the components at different abstraction levels by choosing the design library and the design entity.

Figure 4.13 shows a structural model for the window processor. The window processor has four components: a horizontal filter, a vertical filter, a left diagonal filter and a right diagonal filter. Assume that the RTL horizontal filter is to be verified in our case. So, we use the RTL model (with STD LOGIC data type) for the horizontal filter, and use behavioral models (with INTEGER data type) for the other three filters. Note that the model shown in Figure 4.13 is a general structure model for the window processor. A separate configuration body is developed to implement this architecture, as shown in Figure 4.14.

---

```
-- file name: window_s2.vhd
-- purpose: structural model of window processor

library IEEE;
use ieee.std_logic_1164.all;
library STRUC_INT;
use STRUC_INT.IMAGE_PROCESSING.all;

---------- interface declaration ----------------------------------------
-- entity WINDOW_PROCESSOR is
 --  generic(HORIZ_DELAY,VERT_DELAY,LEFT_DIAG_DELAY,RIGHT_DIAG_DELAY,
--              WAIT_TIME: TIME);
 --   port( CLOCK: in STD_LOGIC:='0';          -- system CLOCK
```

```
--          P1,P2,P3: in  PIXEL:=0;                -- input pixels
 --         W_H: inout FILTER_OUT:=0;          -- HORIZONTAL FILTER output
 --        W_V: inout FILTER_OUT:=0;           -- VERTICAL FILTER output
 --        W_DL: inout FILTER_OUT:=0;          -- LEFT DIAGNAL FILTER output
 --       W_DR: inout FILTER_OUT:=0 );        -- RIGHT DIAGNAL FILTER output
-- end WINDOW_PROCESSOR;

---------- structural description ----------------------------------------
architecture structure of WINDOW_PROCESSOR is
---------- empty HORIZONTAL FILTER component -----------------------------
  component HORIZONTAL_FILTER1
  end component;
---------- empty VERTICAL FILTER component -------------------------------
  component VERTICAL_FILTER1
  end component;
---------- empty LEFT DIAGNAL FILTER component ---------------------------
  component LEFT_DIAG_FILTER1
  end component;
---------- empty RIGHT DIAGNAL FILTER component --------------------------
  component RIGHT_DIAG_FILTER1
  end component;
begin
------------- empty components instantiation ----------------------------
  HP: HORIZONTAL_FILTER1;
  VP: VERTICAL_FILTER1;
  LDP: LEFT_DIAG_FILTER1;
  RDP: RIGHT_DIAG_FILTER1;
end structure;
```

**Figure 4.13** Structural model for the window processor

```
 configuration WINDOW_STRUC of WINDOW_PROCESSOR is
        generic map(HORIZ_DELAY=>3 ns, VERT_DELAY=>3 ns,
                    LEFT_DIAG_DELAY=>3 ns,  RIGHT_DIAG_DELAY=>3 ns,
                   WAIT_TIME=>0 ns)
        port map(CLOCK, MEM_OUT1, MEM_OUT2, MEM_OUT3, E_H, E_V, E_DL, E_DR);
        for structure
              for HP: HORIZONTAL_FILTER1 use configuration STRUC_RTL.H_RTL
                    generic map(HORIZ_DELAY=>HORIZ_DELAY)
                    port map (CLOCKH=>CLOCK,  P1H=>INT_TO_STDLOGIC8(P1),
                                 P3H=>INT_TO_STDLOGIC8(P3),
                                 STDLOGIC_TO_INT(H)=>INT_TO_STDLOGIC12(W_H));
```

```
            end for;
            for VP: VERTICAL_FILTER1
                  use entity STRUC_INT.VERTICAL_FILTER(BEHAVIOR)
                  generic map(VERT_DELAY=>VERT_DELAY,
                              WAIT_TIME=>WAIT_TIME)
                  port map (CLOCK, P1, P2, P3, W_V);
            end for;
            for LDP: LEFT_DIAG_FILTER1
                  use entity STRUC_INT.LEFT_DIAG_FILTER(BEHAVIOR)
                  generic map(LEFT_DIAG_DELAY=>LEFT_DIAG_DELAY,
                              WAIT_TIME=>WAIT_TIME)
                  port map (CLOCK, P1, P2, P3, W_DL);
            end for;
            for RDP: RIGHT_DIAG_FILTER1
                  use entity STRUC_INT.RIGHT_DIAG_FILTER(BEHAVIOR)
                  generic map(RIGHT_DIAG_DELAY=>RIGHT_DIAG_DELAY,
                              WAIT_TIME=>WAIT_TIME)
                  port map (CLOCK, P1, P2, P3, W_DR);
            end for;
 end for;
end;
```

**Figure 4.14** An application of multiple level simulation: the horizontal filter is a RTL level structure model using STD_LOGIC data type, the other filters are behavioral models using integer data type for image data.

The VHDL codes for the RTL level horizontal filter and the configuration body are shown in Figure 3.24 and Figure 3.25. As shown in Figure 4.14, we use the statement

***for HP:HORIZONTAL_FILTER1 use configuration STRUC_RTL.H_RTL;***

to choose this RTL model for the horizontal filter. For the other three filters, we choose the behavioral models. For example, we use the following statement to choose the behavioral model of the vertical filter:

*for VP: VERTICAL_FILTER1 use entity STRUC_INT.VERTICAL_FILTER(BEHAVIOR);*

### 4.4.2 Mixed Data Type Simulation

When do large scale designs, we may want to use some sets of models acquired from outside the organization. The data types used by those models might be different from those of our own design. When we develop a system using the components from both resources, there exists a communication problem between the components with different data types. One way to

solve this problem is that we can change the data types in our own design. Although this method can solve the problem, it is tedious. And, if we have hundreds of components with different data types, it is impractical and error-prone to change the data types for all of them. In this case, we need to find a simple way to make those two sets of data types "talk" to one another without redesigning any component. Since the simulation is applied to a system which consists of components of different data types, we called it *mixed data type simulation.*

VHDL functions play important roles in the mixed data type simulation. They serve as the basis for this method. Figure 4.15 gives a data conversion function which converts INTEGER data type to 8- bit STD_LOGIC_VECTOR data type.

```
function INT_TO_STDLOGIC8(A: integer) return STD_LOGIC_VECTOR is
  variable RESULT : STD_LOGIC_VECTOR(0 to 7):="00000000";
  variable TEMP_A, TEMP_B : INTEGER:=0;
  begin
         TEMP_A := A;
        for I  in 7 downto 0 loop
                TEMP_B:=TEMP_A/(2**I);
                TEMP_A:=TEMP_A REM(2**I);
                 if (TEMP_B = 1) then
                        RESULT(7-I):='1';
                 else
                        RESULT(7-I):='0';
                end if;
        end loop;
  return RESULT;
end INT_TO_STDLOGIC8;
```

**Figure 4.15** The data conversion function: INTEGER to 8-bit STD_LOGIC_VECTOR

After the data conversion functions have been developed, they can be store in a package for future use. To achieve mixed data type simulation, we need to associate these data conversion functions with particular ports in the component instantiation. This is done by using the data type conversion function names as part of the port association. The following paragraphs illustrate the mixed data type simulation using the Sobel edge detection system.

Figure 4.13 and Figure 4.14 show a general structural model of the window processor and a separate configuration associated to it. Assume that we wish to insert this structural model for the window processor into the edge detection system and apply simulation to it. In this example, only the horizontal filter fully uses the STD_LOGIC data type. Other models partially use INTEGER data type. In other words, the horizontal filter should accept INTEGER data type signals as its inputs and send out INTEGER data type signals as its outputs. The VHDL code

shown in Figure 4.14 presents an efficient way to solve the communication problem between the horizontal filter and the other components.

Since the horizontal filter is the only component that has data type conflicts with the other components, the data conversion functions are used as part of port maps for this filter. Three data conversion functions are used: INT_TO_STDLOGIC8 (INTEGER to 8-bit STD_LOGIC_VECTOR), STDLOGIC_TO_INT (STD_LOGIC_VECTOR to INTEGER) and INT_TO_STDLOGIC12 (INTEGER to 12-bit STD_LOGIC_VECTOR). The named associations are used for the port associations in this example. The signals (**P1H**, **P3H** and **H)** which are declared in the components are of data type STD_LOGIC_VECTOR. **P1**, **P3**, **W_H** are type INTEGER and are declared in the window processor as intermediate signals which connect the horizontal filter to the memory processor and the magnitude processor. Data conversion functions are used to change the signals of type INTEGER from the outside (**P1** and **P3**) into type STD_LOGIC_VECTOR which are then mapped to the input ports of the horizontal filter (**P1H** and **P3H**, respectively). And, the output of the horizontal filter (**H**, data type STD_LOGIC) is converted into INTEGER data type and then mapped to the outside signal (**W_H**) whose data type is INTEGER. Thus, the horizontal filter component can communicate with other components and the mixed data type simulation can be implemented.

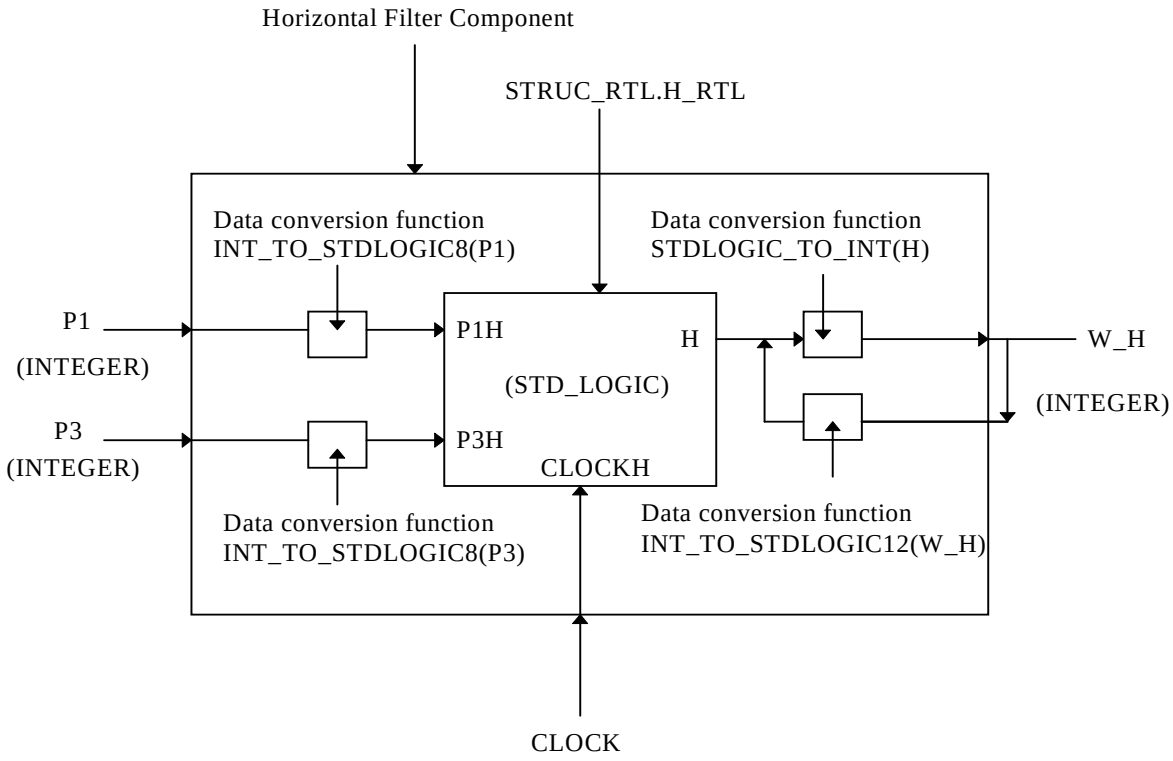



**Figure 4.16** Associate the data conversion functions with the port maps

Figure 4.16 diagrams how the STD_LOGIC data type horizontal filter communicates with other INTEGER data type components.

The following statement gives the port map for the horizontal filter according to Figure 4.16. The signals on the left side of the arrow (type STD_LOGIC) are the ones that are declared in the horizontal filter design entity. The signals on the right side of the arrow (type INTEGER) are the ones declared in the structural model of the window processor.

*port map (CLOCKH=>CLOCK,* ***P1H=>INT_TO_STDLOGIC8(P1)****,*
***P3H=>INT_TO_STDLOGIC8(P3)****,*
***STDLOGIC_TO_INT(H)=>INT_TO_STDLOGIC12(W_H))****;*

P1 and P3 are converted from INTEGER data type to STD_LOGIC data type by the statement ***INT_TO_STDLOGIC8(P1)*** and ***INT_TO_STDLOGIC8(P3)***, respectively. Since port **H** is declared as inout, we must have conversion functions in both directions. Signal **W_H** is type INTEGER and is converted to STD_LOGIC data type by the data conversion function ***INT_TO_STDLOGIC12(W_H)***. Port **H** is converted from STD_LOGIC to INTEGER by the data conversion function ***STDLOGIC_TO_INT(H)***.

Another benefit of mixed data type simulation is that it helps to save simulation time. For example, the simulation time is less for the INTEGER/NATURAL data type component than that for the STD LOGIC data type component. The saving is particularly significant when we simulate large systems.

## 4.5 Design Trade-off

### 4.5.1 Introduction

It is possible that a design can have several implementations. In other words, a design entity can have a common interface but several architectures. Since the functionality of the design is unchanged, the different implementations actually refer to different structural architectures. When we decompose a component into several sub-components, different ways of partitioning result in different structural architectures. Trade-offs exist between these architectures, such as design time, design effort, design cost, fabrication cost, fastest working frequency, minimum area, minimum power used, etc. There is often no absolutely best architecture. The designer should evaluate design alternatives according to the application. For example, a trade-off can be made between the architectures which use customer designed components (ASIC) and COTS (commercial off-the-shelf) components. Also, trade-offs exist when the gate level circuits are synthesized to achieve different design goals: minimum area, minimum working clock frequency or minimum power consumption.

### 4.5.2 Trade-Offs Among ASIC Designs and COTS Designs

In a top-down design process, the hardware is decomposed into a collection of interconnected modules. The decomposition is repeated until the design reaches an abstraction level where the components can be fabricated. Usually, the manual design work ends with the customer designed components at the RTL. Then the synthesis tools are applied to these RTL components to automatically generate the gate level models to be fabricated. The components produced in this way are seldom commercially standard. They are designed for a specific application, thus are called ASICs.

On the other hand, a designer sometimes wants to use COTS components for a system. The designer should be clear about what components are available. The system should be constructed only from those components. This kind of design process infers a bottom-up design, because the design begins with the components that are available.

When comparing ASIC designs with COTS designs, the following facts should be considered.

- A circuit consisting of several COTS components is behaviorally similar to a single ASIC component. But sometimes, there are timing differences between the two.
- Total power consumption is usually more for COTS implementations than that of the ASIC implementations.
- Design time and design effort are more for the COTS system so that the design cost is higher because the ASIC design is done by synthesis tools.
- The fabrication cycle is shorter for COTS designs because the components are commercially available; a circuit can be constructed by inserting the COTS chips on a circuit board.
- The fabrication cost for an ASIC design is very high because the fabrication mask is very expensive.
- Assembly costs are higher for COTS designs because parts must be inserted into a PC board and the boards must be tested. Also, more parts must be purchased.
- The total cost (including design cost, fabrication cost, and assembly cost) depends on the production quantity.

The fabrication cost includes the cost of the fabrication masks, the circuit itself and the circuit board. For a COTS design, since the fabrication masks are available, there is no fabrication mask cost. The cost for a COTS design is the sum of the cost of the chips. However, new fabrication masks are needed for an ASIC design. The total cost for an ASIC product includes the mask cost per chip and the cost for fabricating the circuit. It might be very expensive to make the fabrication mask for an ASIC design. Assume the cost for fabricating one circuit remains the same regardless of the production quantity. Thus, the fabrication cost for each ASIC product is high when the production quantity is small, because the mask cost per chip is very high. As the production quantity increases, the mask cost per chip reduces. When the production

quantity is large, the cost of the fabrication masks for each product becomes very low. Thus, the total fabrication cost of the ASIC design might be less than that of the COTS circuit.

### 4.5.3 Trade-offs Among Optimized Gate-Level Circuits

As mentioned in Chapter 3, gate level circuits are first synthesized from the RTL models by commercial tools. Then optimizations are performed to achieve different design goals: minimum area, minimum clock period or minimum power consumption. The typical optimization methodology is to optimize for area first and then to optimize for timing. The synthesis tools carry out the design optimization according to user defined constraints. Current synthesis tools typically allow constraints to be set for minimal area and minimal timing delay. Trade-offs exist among these gate-level optimized circuits. The following section will use the window processor to illustrate the methodology of making the trade-offs among these circuits.

### 4.5.4 Case Study: Making Trade-Offs for the Window Processor

In the Sobel edge detection system, the window processor is used to apply the Sobel operators on the image data. The RTL window processor consists of three components, namely, the horizontal filter, the vertical filter and the diagonal filter (which combines the left diagonal filter and the right diagonal filter), as shown in Figure 4.17.

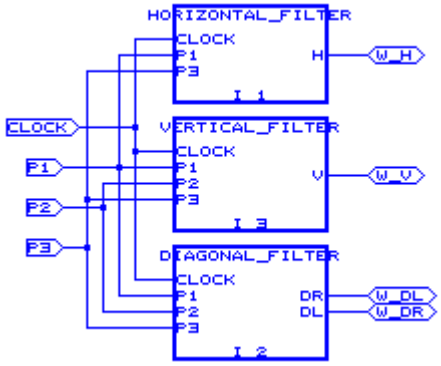



**Figure 4.17** SGE diagram of the window processor

After obtaining the gate level circuit for each filter, optimizations have been performed on them to optimize for area first and then to optimize for timing. Table 4.1, Table 4.2 and Table 4.3 show the optimization results for the horizontal filter, the vertical filter and the diagonal filter, respectively.

**Table 4.1** Optimization results of the horizontal filter

| Circuit No. | Max Area Specified | Clock Period Specified (ns) | Clock Skew (ns) | Area Consumed | Slack (clock_period - critical path delay ) (ns) | Power consumed ($\mu W$) |
|---|---|---|---|---|---|---|
| 1 | not specified | not specified | not specified | 706 | not specified | 276.24 |
| 2 | 0 | 50 | 1 | 706 | 29.02 | 13.86 |
| 3 | 0 | 31 | 1 | 712 | 10.53 | 21.87 |
| 4 | 0 | 21 | 1 | 709 | 0.81 | 31.94 |
| 5 | 0 | 18 | 1 | 712 | 0.3 | 38.07 |
| 6 | 0 | 14 | 1 | 740 | 0.05 | 52.44 |
| 7 | 0 | 11 | 1 | 751 | 0.02 | 66.69 |
| 8 | 0 | 9 | 1 | 823 | 0.01 | 98.01 |

**Table 4.2** Optimization results of the vertical filter

| Circuit No. | Max Area Specified | Clock Period Specified (ns) | Clock Skew (ns) | Area Consumed | Slack (clock_period - critical path delay ) (ns) | Power consumed ($\mu W$) |
|---|---|---|---|---|---|---|
| 1 | not specified | not specified | not specified | 749 | not specified | 251.20 |
| 2 | 0 | 50 | 1 | 749 | 24.88 | 13.32 |
| 3 | 0 | 25 | 1 | 745 | 5.94 | 25.40 |
| 4 | 0 | 15 | 1 | 766 | 0.04 | 45.81 |
| 5 | 0 | 12 | 1 | 788 | 0.01 | 59.05 |

**Table 4.3** Optimization results of the diagonal filter

| Circuit No. | Max Area Specified | Clock Period Specified (ns) | Clock Skew (ns) | Area Consumed | Slack (clock_period - critical path delay ) (ns) | Power consumed ($\mu W$) |
|---|---|---|---|---|---|---|
| 1 | not specified | not specified | not specified | 1744 | | 767.91 |
| 2 | 0 | 50 | 1 | 1755 | 23.79 | 35.27 |
| 3 | 0 | 25 | 1 | 1748 | 2.33 | 68.48 |
| 4 | 0 | 21 | 1 | 1765 | 0.04 | 84.56 |
| 5 | 0 | 19 | 1 | 1763 | 0.01 | 92.32 |

The optimization results for the minimum area, minimum clock period and minimum power for each filter are summarized in Table 4.4. As we can see, there are trade-offs among the optimized circuits which achieve different design goals. A minimum area circuit might have a longer working clock period and/or might use more power. A faster circuit might have more area and/or consume more power. A circuit that consumes minimum power might not be the one with the minimum area or might not be the fastest circuit.

**Table 4.4** Circuits with different constraints

| Filter Name | Optimization | Area | Clock Period (ns) | Power Consumption ($\mu W$) |
|---|---|---|---|---|
| Horizontal (2) | min. Area | 706 | 50 | 13.86 |
| Horizontal (8) | min. Clock period | 823 | 9 | 98.01 |
| Horizontal (2) | min. Power consumption | 706 | 50 | 13.86 |
| Vertical (3) | min. Area | 745 | 25 | 25.40 |
| Vertical (5) | min. Clock period | 788 | 12 | 59.05 |
| Vertical (2) | min. Power consumption | 749 | 50 | 13.32 |
| Diagonal (3) | min. Area | 1748 | 25 | 68.48 |
| Diagonal (5) | min. Clock period | 1763 | 19 | 92.32 |
| Diagonal (2) | min. Power consumption | 1755 | 50 | 35.27 |

Simply choosing a particular optimal component is meaningless, because the components should work together to perform the system function. The designer should make a balance among the optimized circuits to obtain an optimal system. For example, suppose that the designer wants to obtain the fastest circuit for the window processor. Since the window processor consists of three filters, we should make a balance among these filters. As shown in Figure 4.17,

the three filters operate in parallel to process the input image data, and each filter output is synchronized by a common system clock. Thus the designer should consider that the minimum working clock period should be the maximum of the minimum periods for the three filters. Considering the corresponding minimum clock period for each filter, we obtain that the minimum working clock period for the window processor is 19 ns which is determined by the diagonal filter. Then under this condition, we can separately choose circuits with less area and less power consumption for the horizontal filter and the vertical filter as long as the clock period does not exceed 19 ns. For example, we can choose the No. 5 circuit from Table 4.1 for the horizontal filter, and the No. 4 circuit from Table 4.2 for the vertical filter whose area and power consumption are less than the other choices.

# Chapter 5. Distance Teaching over the Internet

## 5.1 Introduction

Nothing before has captured more imagination and interest of educators simultaneously around the globe as has the World Wide Web. The Web is the way of linking text, images, sound, and video resources on computers that are connected to the Internet, the world-wide network of computer networks. The power of hyperlink is that in a hyperlinked document if one wants more information about a particular subject mentioned, one can usually "just click on it" to obtain more detail [11].

The Web changes the traditional way of teaching and learning and provides convenient access to the resources.

- Learning resources can be augmented by learning resources around the world via the Web [11]. Learning material can be written using a hyperlinked document such as an HTML file. Each file has a unique Internet address that identifies it. The educator can choose and give accesses to other material by making links to their addresses. By simply clicking on the mouse button, one can navigate seamlessly from one resource to the other without regard to whether the resource is on the same computer or on another computer located elsewhere on the Internet. One commonly used navigation tool is Netscape.
- With web, teaching and learning can be freed from the boundaries of classrooms and class schedules [12]. Traditional lectures and teacher presentations can become multimedia learning experiences for students. The educator can link material in different formats, such as text, video, and audio, to make the teaching material more interesting and animated. In addition, students are provided with a flexible learning environment. They are not limited by the time, or classroom any more, but can access multimedia educational material at their convenience
- The Web can help us re-focus our institutions from teaching to learning, from teacher to student [11]. With the web, we can create educational modules specific for students

with different backgrounds that allow them to learn more actively. The students can follow their own interests to explore the depth of the topics.

As for the educational module, it is important that we have not only good teaching material, but also a good organization of the material. Usually, a person learns by doing. One easily becomes bored when reading a "page by page" material. Hence educational modules should be interactive and interesting, and encourage the student to search for more information.

## 5.2 Goals for the Computer Based Educational (CBE) Module

The main goal of the VITAMINS project is to create a low cost modular learning experiences over the Internet on the use of VHDL and its related standards for the Air force acquisition and maintenance specialists.

The major requirements of the CBE module are:

1. The CBE module should be an effective source of information about the RASSP design process. The teaching material should be carefully chosen, based on the interests of the reader.
2. The CBE module should provide an efficient way for the reader to get information. The teaching material should be organized properly so that the reader can easily find the information.
3. The CBE module should be suited for various users. Links to the topics with different detail and other related material should be provided by the designer. It is the reader who decides how much detail to learn.
4. Access to the other material about RASSP tools, VHDL models and data files should be provided.
5. The courseware should be structured in such a way that changes can be easily made.
6. The CBE module should be interesting and interactive so that the users are encouraged to interact with the module.

All the material discussed in Chapters 2, 3, and 4 of this thesis have been carefully re-organized and integrated into the CBE module. In addition, links are made to the material of the RASSP design home pages, the VHDL home pages and the MIL-HDBK-62 (Department of Defense handbook).

## 5.3 Developing the CBE Module Over the Internet

As we mentioned before, a CBE module can not be developed by simply collecting all the teaching material or linking them from the first page to the last page. Well organizing the teaching material is as important as preparing the material. The related materials should be linked to each other so that a reader can choose the topics to learn. Both text files and figures should be used to explain the problem. Also, the CBE should be interactive so that a reader learns by doing.

### 5.3.1 Organizing the Teaching Materials

In order to help the student follow and understand the tutorial, the teaching materials are separated into several chapters. The major chapters are *Introduction, Background, Abstraction Levels, Design Techniques, Testing Strategies, Related Links, RASSP home page* and *MIL-HDBK-62 (Department of Defense handbook).*

A brief introduction to the VITAMINS project is made in the Introduction chapter. Concepts such as VHDL modeling, Sobel edge detection algorithm, WAVES and the RASSP design process are provided in the Background chapter that helps the reader to understand the later chapters.

Since the goal of this CBE is to teach the Air force acquisition and maintenance specialists how to efficiently and effectively model and test a system, focuses are on the abstraction levels in the design process, the design techniques and the testing strategies chapters. In these chapters, detailed explanations are provided for the top-down design process, reuse and model year concept, integrated design environment, back annotation, diagnostic testing, regression testing, etc. A complete VHDL simulation based on the top-down design process, from the written specification to the gate level circuit synthesis, is presented. The tutorial is organized in such a way that it follows the natural top-down design process and has a sufficient but clear explanation on each design step. In order to teach the student how to prepare models in various design stages for testing, maintenance and reuse, design techniques such as the coding style, the usage of libraries, generics, packages, multiple level simulation, and mixed data type simulation are explained. Testing strategies such as behavioral test bench, structural test bench, diagnostic testing, regression testing are explained along with the design process.

Links are made to the MIL-HDBK-62 (Department of Defense handbook), several VHDL home pages and the RASSP home page. The reader can navigate to those web pages to get more information about VHDL and the RASSP design process.

### 5.3.2 Developing the HTML files

Since the educational module should be accessed over the Internet, all the materials in our CBE are written in HTML which can be read by many popular web browsers.

In practical forms, HTML files define the various components of a WWW document. HTML documents are plain-text (also known as ASCII) files that can be created using any text editor (e.g., *emacs*, *vi* or *textedit* on UNIX machines) [13]. Markup tags are used in the HTML file to make the paragraphs, tables, lists, etc., and to make links to other material. For example, the following statement makes a link from the current web page to the web page welcome.htm. By clicking on the sentence “Welcome!”, that web page will be displayed.

<a href=“welcome.htm”>Welcome!</a>

In our project, the teaching materials of other format are first converted to HTML files. The figures are converted to JPG files or GIF files by commercial tools such as Paint Shop, and then links are made to the corresponding HTML file. The teaching materials are reorganized and an individual HTML file is made for each teaching topic.

### 5.3.3 Web Page Design Techniques

Some web page design techniques are discussed in this section. As mentioned before, the CBE module should contain sufficient teaching materials. Also, it should be a comfortable and interesting environment for the learning activity.

#### 5.3.3.1 Making a Comfortable Learning Environment

Making a comfortable reading environment is very important. Our CBE should make the reader feel comfortable and assist one to easily search for detailed information. The background colors of our browser windows are carefully chosen so that the contrast is suited for reading. In order to help the reader to capture the information, different fonts are used to differentiate the sub-topics and the textual explanation. The reader can quickly go through the topics before reading the detailed contents.

#### 5.3.3.2 Making an Interesting Learning Environment

Effort has been made to make the web page interesting and assist the reader to understand the teaching material. For example, images are frequently provided along with the textual explanation. Figure 5.1 shows a partial web page for the requirement repository lesson. This page uses several figures to illustrate the design process step by step. Images make the teaching material animate and helps the reader to understand the textual explanation.

#### 5.3.3.3 Providing Step by Step Explanations

In order to improve the readability of the tutorial, long textual explanation has been replaced by a step by step learning process. In the example shown in Figure 5.1, we explained how to capture the requirements of the system. Each design step is explained first then figures are used to show the results. The reader can easily adapt the tutorial to do the design for another system. Note detailed explanations for particular items are hidden by links. For example, the explanations for the system inputs and outputs are provided by another web page. The reader can click on the item “input and output” to obtain the information.

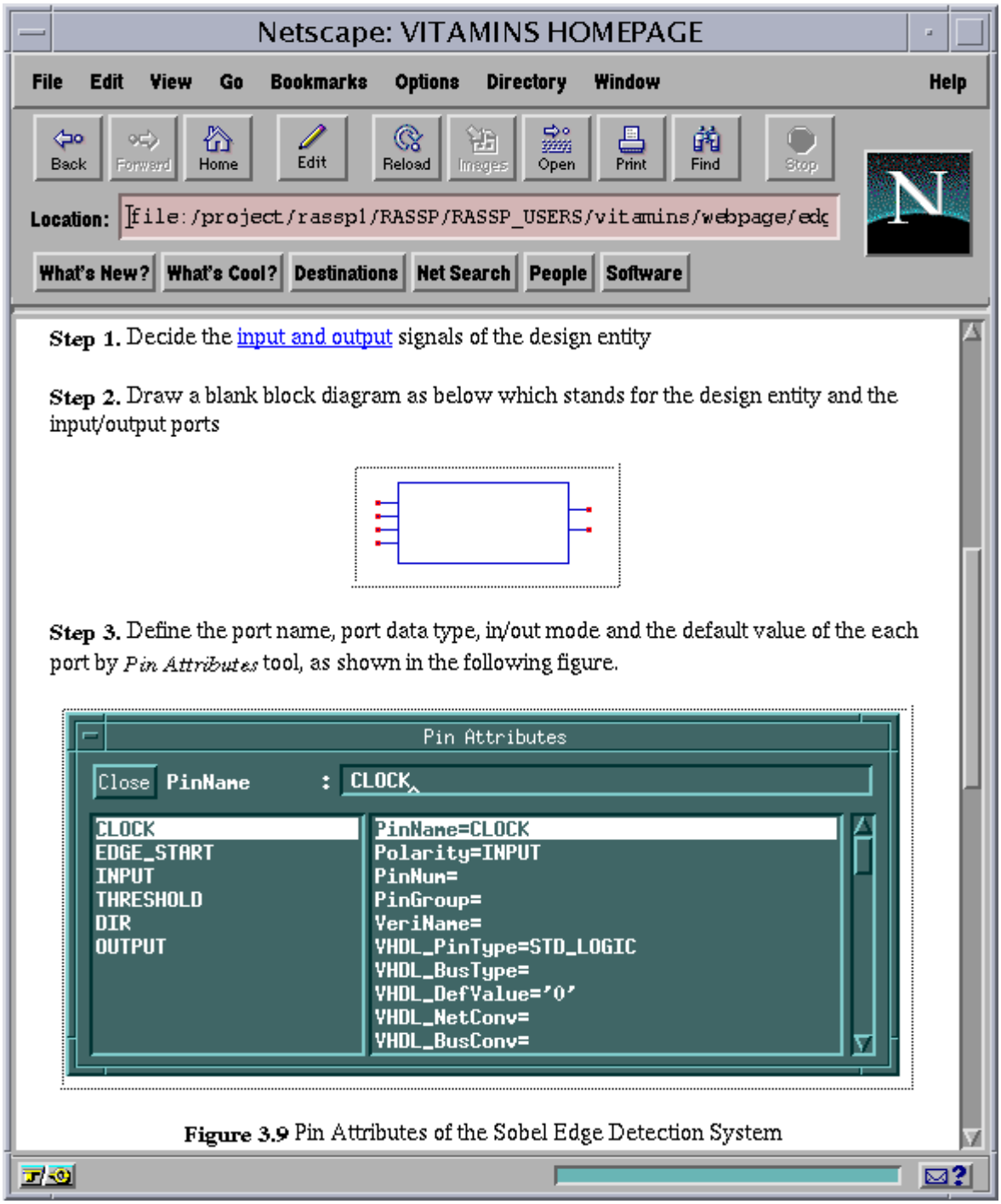



**Figure 5.1** Partial web page of the requirement repository page

#### 5.3.3.4 Providing a Complete Tutorial for Design, Testing, and Results

A complete tutorial including the step by step design process, developing the test vectors, the test results, and the source codes are provided for a model development. These materials are hyperlinked to each other. The student can selectively read these materials. For example, a complete explanation of how to develop and test the executable specification is provided. A link is made to the web where the model is explained. The reader can selectively read the code.

#### 5.3.3.5 Making Use of Frame Sets

The frame is sometimes divided into sub-frames which help a reader to compare two or more documents. Or, textual explanations and the figures are given in a separate frame on one web page. Figure 5.2 shows a web page in which the step by step design process for the system decomposition is explained. The upper left frame explains the steps in the executable specification. The upper right frame explains how the components are developed according to those steps. The bottom frame gives the SGE diagram of the system decomposition. The resulting components, the interconnections between those components and the intermediate signals are shown in this figure.

#### 5.3.3.6 Making a Short HTML for Each Page

In order to keep the interest of the reader and fit the readers of different backgrounds, one topic might be explained by several hyperlinked small sub-topics with various details. Each sub-topic is written in a separate HTML file. When a reader clicks on a topic, an introduction page comes out first. The reader can get a general idea about a particular topic from this page. Links are given to the examples and the other related topics for further reading. An example will be given later in the *Library* lesson in section 5.3.4.2.

Another benefit of making smaller HTML files is that they are easier to read. Long files might not fit in one browser window. Although the reader can click on the scroll bar to read the whole file, it is inconvenient for a reader to keep scrolling the window vertically when reading. Keeping each HTML short allows one to concentrate on the material. A complete browser window is used for a particular design issue as often as possible. As shown later in Figure 5.11, a complete window size is used for the design issue: how to change a design library. The reader can use the links to go to other topics. Frame sets are used only if comparisons are needed between teaching topics.

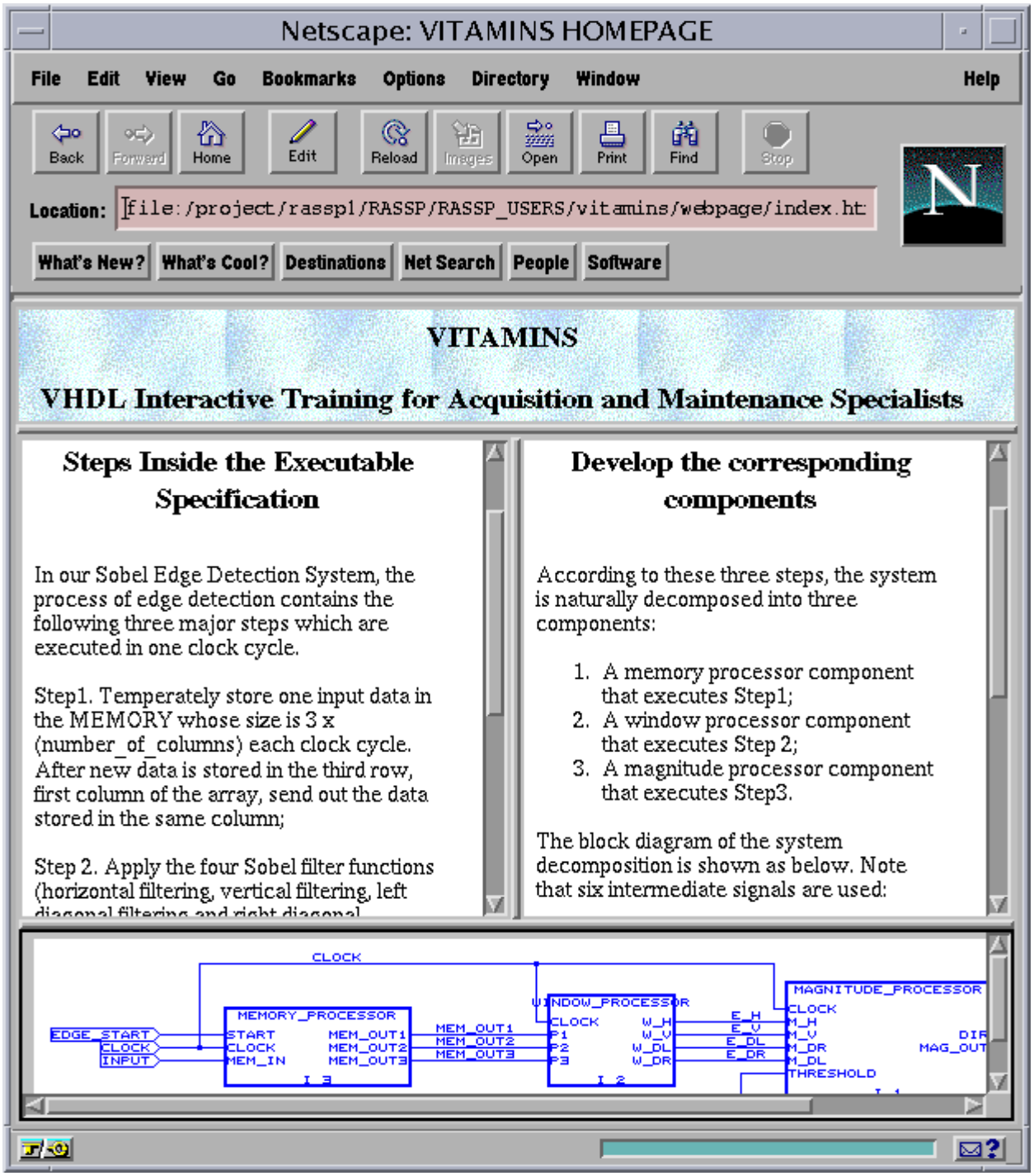



**Figure 5.2** Using frame sets to improve readability: system decomposition web page

#### 5.3.3.7 Making Links

It is common that a reader will go back or forth when reading a book. Appropriate links are given so that readers can easily either follow the tutorial or choose the topics from the “Table of Contents”.

Although readers can search backward and forward by buttons provided by the browser windows, they tend to be led in the learning activity. We have provided links to the previous page, the next page, and the TOC page on each introduction page. The previous page and the next page link allows readers to go back to the previous topic or go forth to the next topic. The link to the TOC page allows readers to go to the table of contents of the tutorial. Also, links to the detailed explanations and other related topics are provided. These links help to make the CBE module convenient for readers.

#### 5.3.3.8 Showing Testing Data in Image

The testing vectors are written in ASCII format which can be read by the test bench, but is hard for a human being to read. Thus, all the test vectors are represented in image format in our CBE, which make the web page more animated as well. Several testing results are provided, and the reader can select by simply clicking on the various links. This method encourages the engagement of the reader. In Figure 5.3, frame sets are used to show the input image, output image and the choices for the output images under different conditions. A reader can choose a condition and see the result.

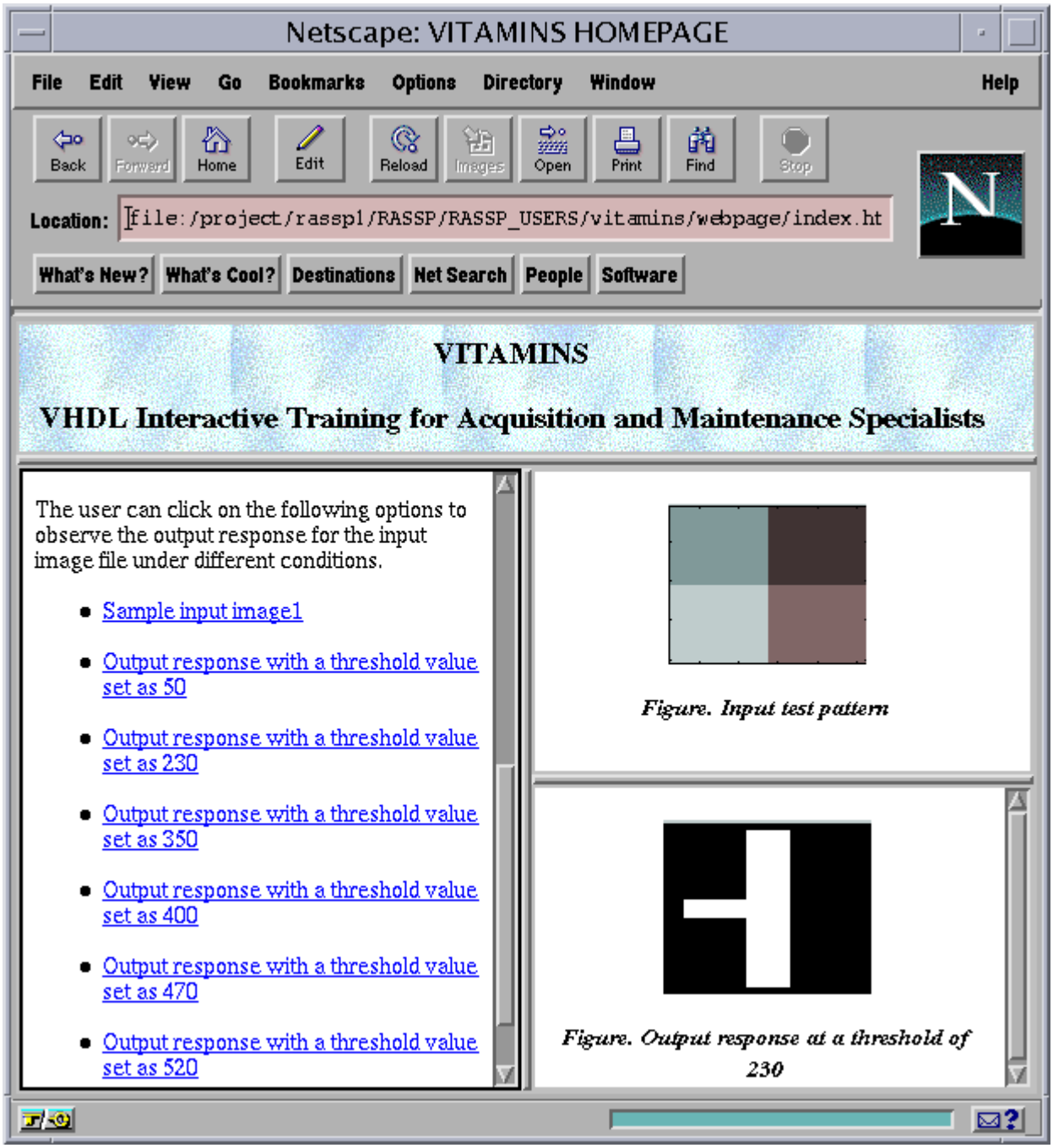



**Figure 5.3** Making use of images: testing result page

### 5.3.4 Developing a Web Site for the CBE Module

A web site for the CBE module has been developed based on the web page design techniques discussed in section 5.3.3. A reader can access the CBE module by opening the files using a Netscape browser. The entry file is index.htm which is a welcome page to the reader. This page leads the reader to the TOC page where the reader can start the tutorial.

Having decided the teaching topics and the organization of the tutorial, the HTML file for the TOC page can be developed. Individual HTML files are designed separately for each teaching topic. Those files are hyperlinked to each other so that the reader can follow the tutorial to learn or they can select topics to learn.

The following sections will discuss the implementation of the CBE module. The TOC page, the executable specification lesson and the library lesson are explained in detail. These two lessons are from the *abstraction levels* chapter and the *design techniques* chapter, respectively.

#### 5.3.4.1 Making a "Table of Contents" (TOC) Page

Before developing the tutorials for each individual teaching topic, a TOC page is developed to structure the teaching topics. This page is developed according to the organization of the teaching material mentioned in section 5.3.1.

The major chapters in the TOC page of our CBE are*: Introduction, Background, Abstraction Levels, Design Techniques, Testing Strategies, Related Links, RASSP home page* and *MIL-HDBK-62 (Department of Defense handbook)*. These chapters are divided further into several sections which give the reader an idea about what is inside each chapter. Figure 5.4 shows part of the TOC window.

Both the designer and the reader get benefits from this page. It helps the designer to organize the teaching topics. It can be easily updated when new topics are added or changes are made. The reader obtains a general idea about the topics of the CBE from this page. The link to this page is made in each individual HTML file so that the reader can always go back to this page to select additional topics to study.

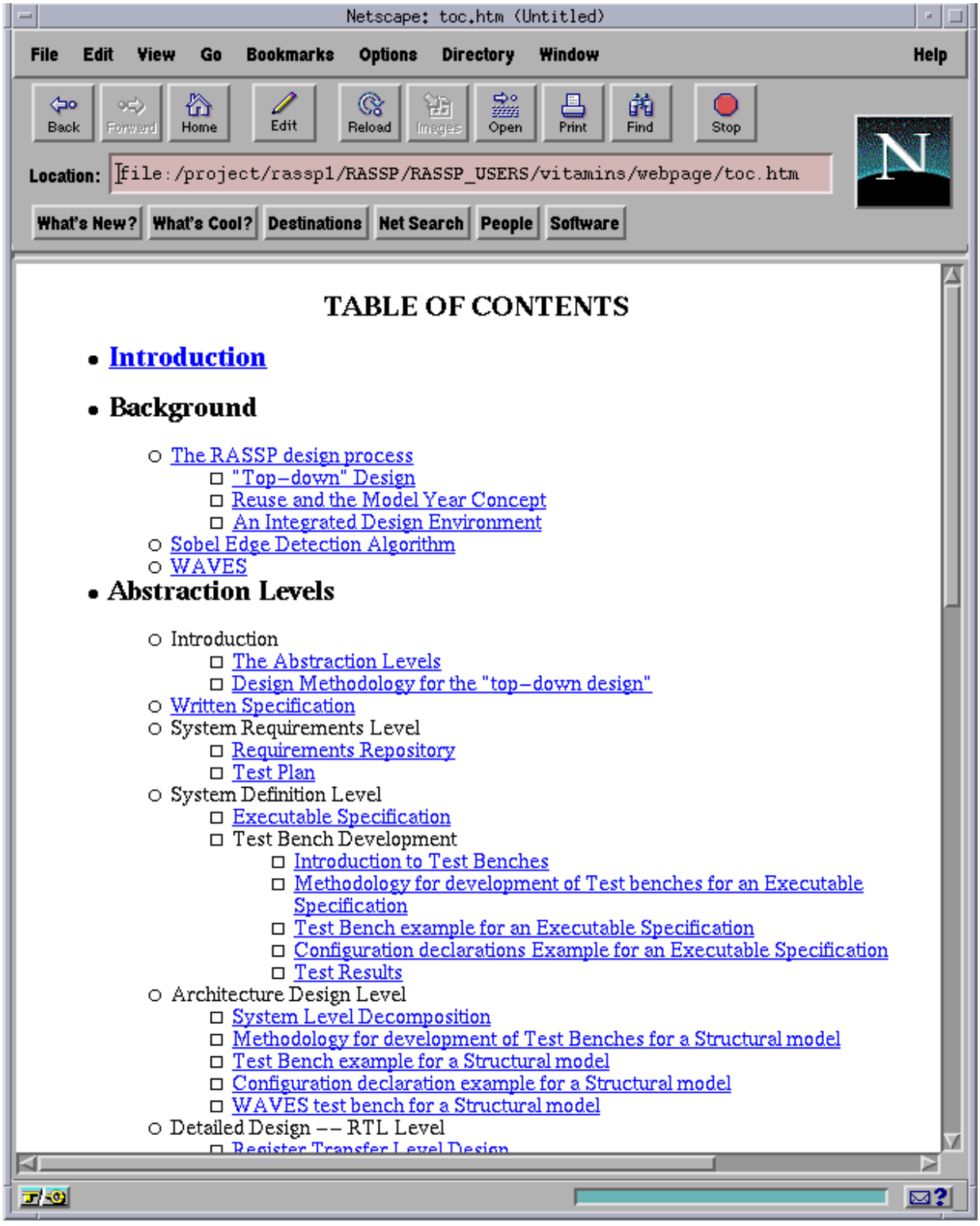



**Figure 5.4** "Table of Content" page

#### 5.3.4.2 Implementation of the *Executable Specification* Lesson

Figure 5.5 shows the introduction page of the *executable specification* lesson. The concept of the executable specification is explained first. The Sobel edge detection system is used as a case study in the lesson. A link to the case study is given which leads the reader to the second page, as shown in Figure 5.6 and Figure 5.7. The behavior of the Sobel edge detection process is described in a step by step sense. A link is made to the written specification page in case the reader wants to review it. The VHDL code is provided by clicking on the corresponding link. As shown in Figure 5.8, the comparison between the VHDL code generated by the SGE tool and the actual executable specification is provided. A reader might want to know the edge detection results for the executable specification. So, the links to two examples are given, as shown in Figure 5.7. By clicking on the item “An artificial testing image”, both the input image and the output image are shown in the browser window, as shown in Figure 5.9. Backward links are given in the pages other than the introduction page, which allow the reader to go back. In the introduction page, links are provided to the previous page, the next page and the TOC page.

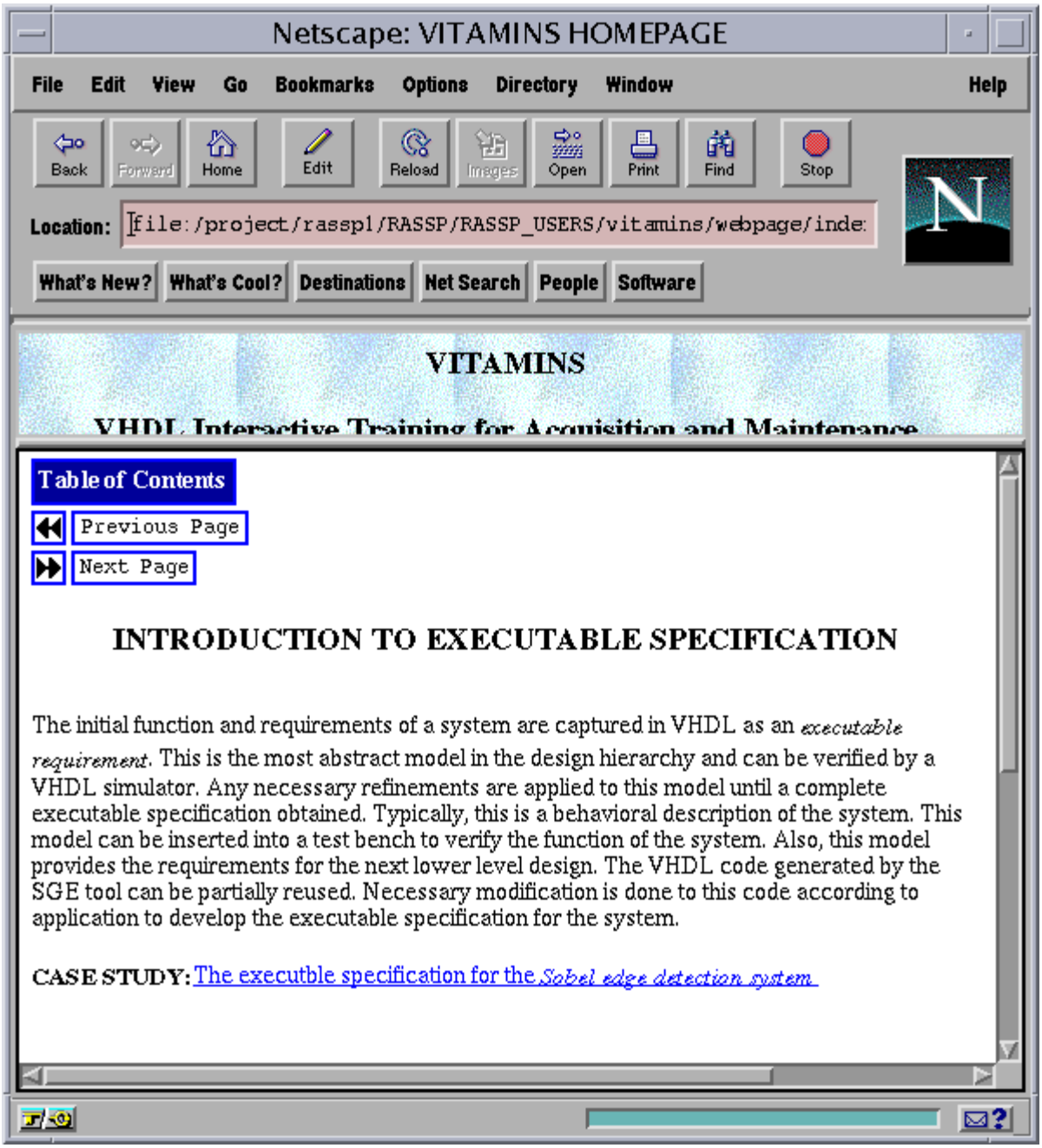



**Figure 5.5** The executable specification lesson: introduction (page 1)

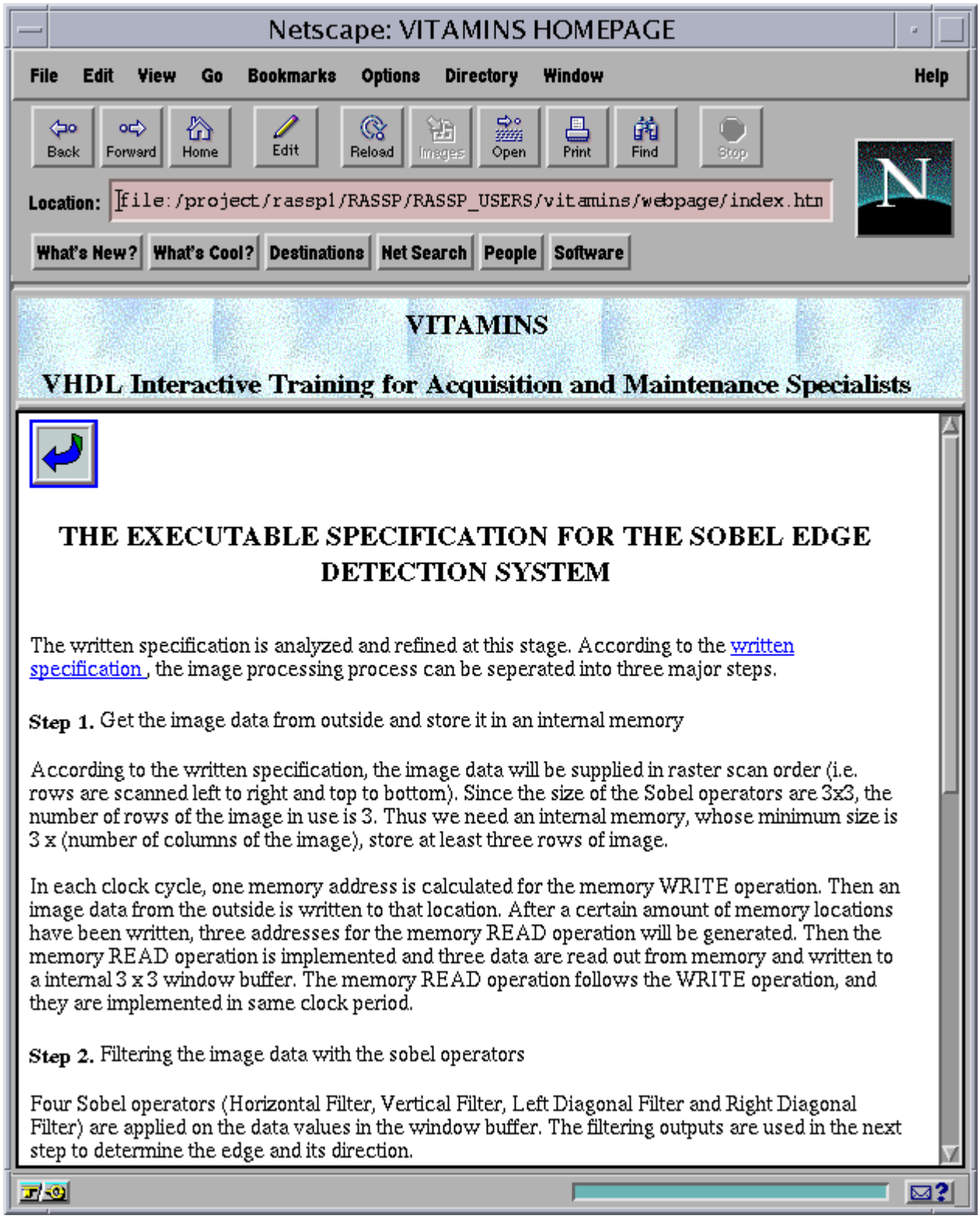



**Figure 5.6** The executable specification lesson: case study (page2)

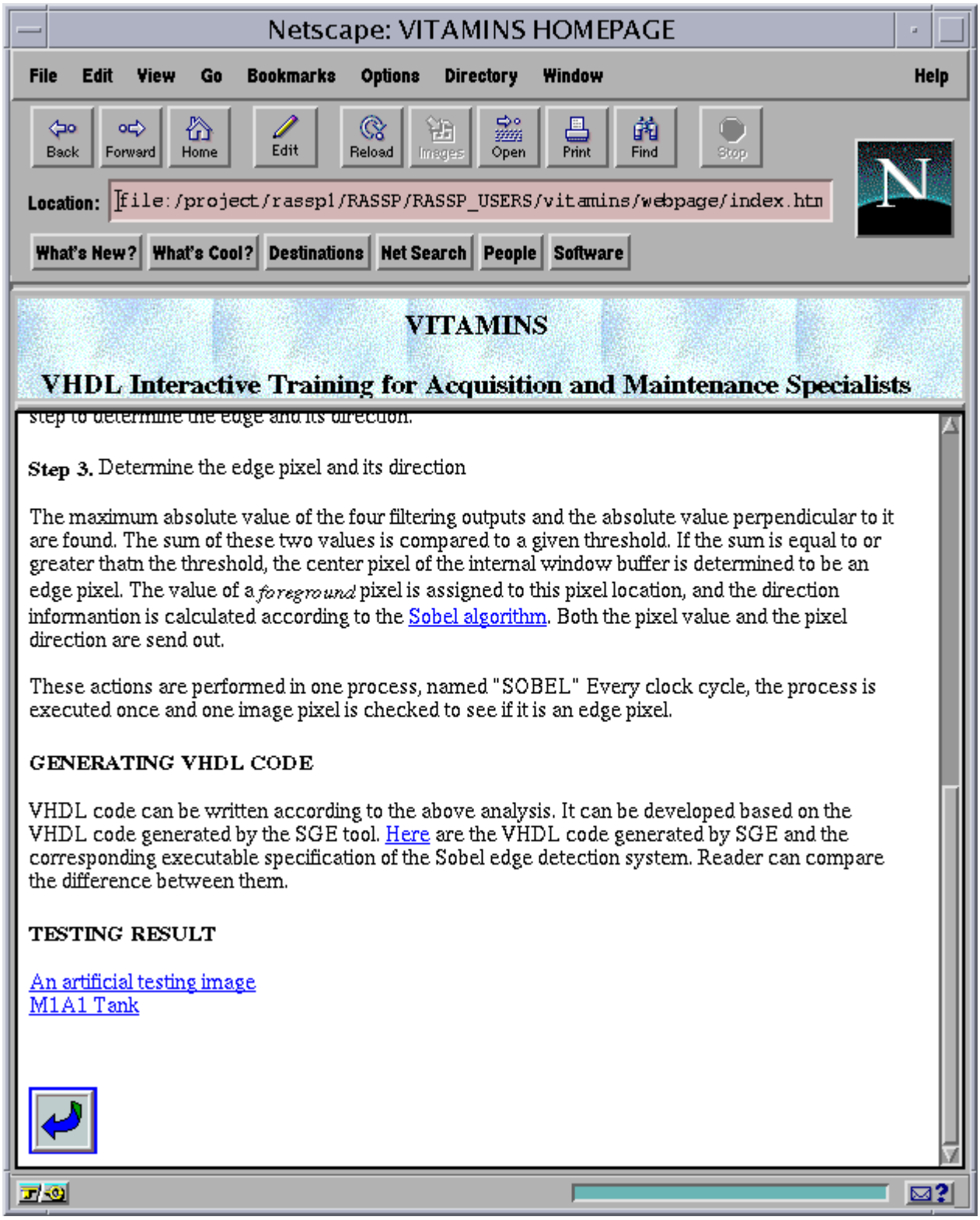



**Figure 5.7** The executable specification lesson: case study (page3)

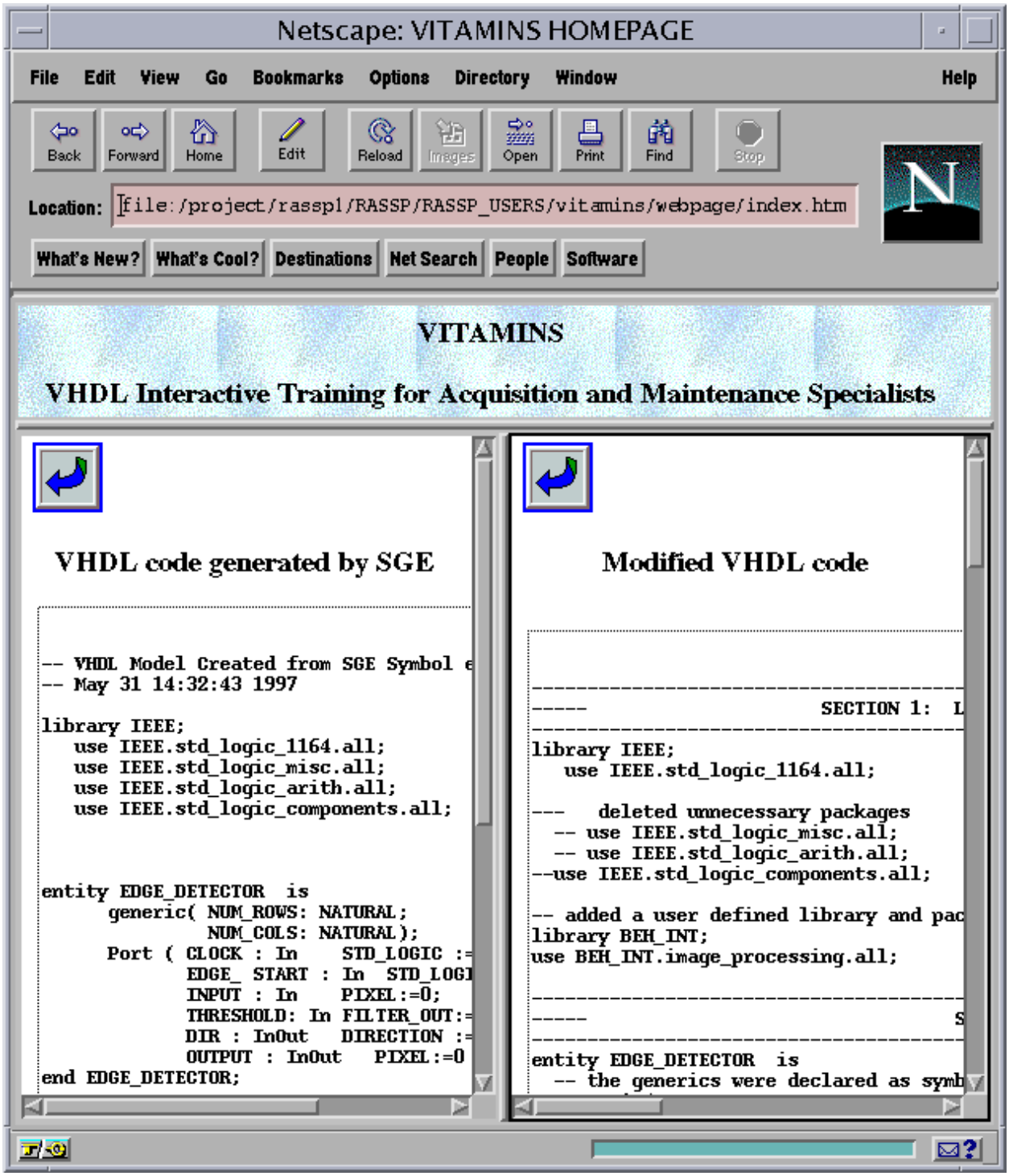



**Figure 5.8** The executable specification lesson: testing image (page 4)

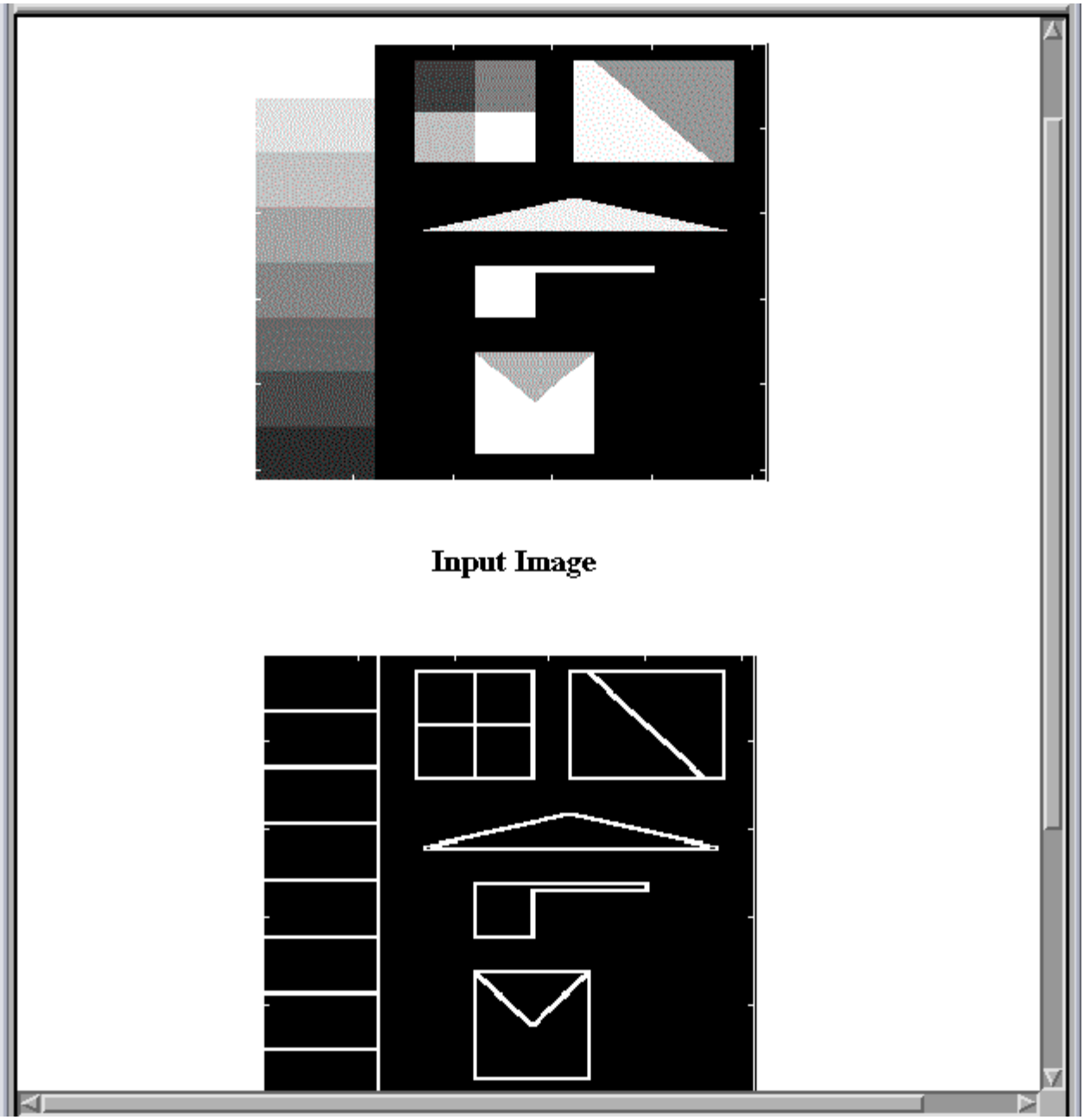



**Figure 5.9** The executable specification lesson: testing image (page 5)

#### 5.3.4.3 Implementation of the *Library* Lesson

Figure 5.10 shows a portion of the *Library* lesson explained in section 4.3.2.1 of this thesis.

The first page of this lesson is the introduction page. The concept and the usage of the *Library* concept are explained first, then links of several design issues, such *as how many libraries can we use, how many names can a particular library have, how to change a design library,* are provided. Also, the links are made to the previous lesson, the next lesson and the table of contents. By clicking on one link of the design issues, the reader is led to another browser window which corresponds to that particular issue. For example, by clicking on the item, *How to change a design library,* another browser window, shown in Figure 5.11, comes out. Also, links to other related issues and the introduction page are provided. These links allow the reader either to go back to the introduction page or to one of the other related issues.

As shown in Figure 5.10, links to the previous page, next page and the TOC page are provided in the introduction page. These links allow the reader to go back to the TOC page, or go back to the previous topic, or go further to the next topic.

The above sections are only three examples of the development of the CBE module. The design methodology discussed in Section 5.3.3 has been selectively adopted in the design of the teaching courses. Efforts have been made to provide students an interactive, interesting and comfortable learning environment over the Internet.

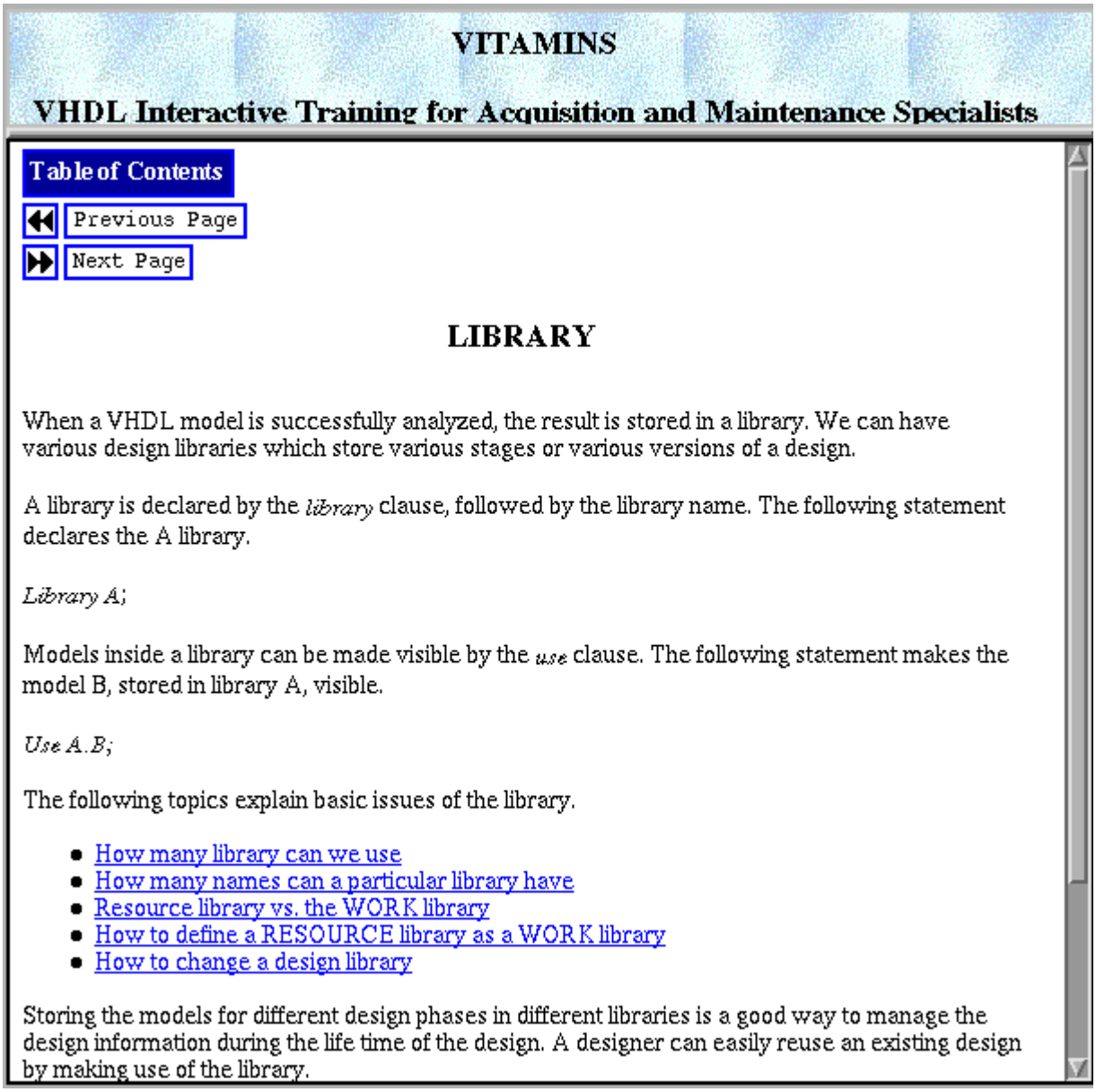



**Figure 5.10** The web page for the *library* lesson

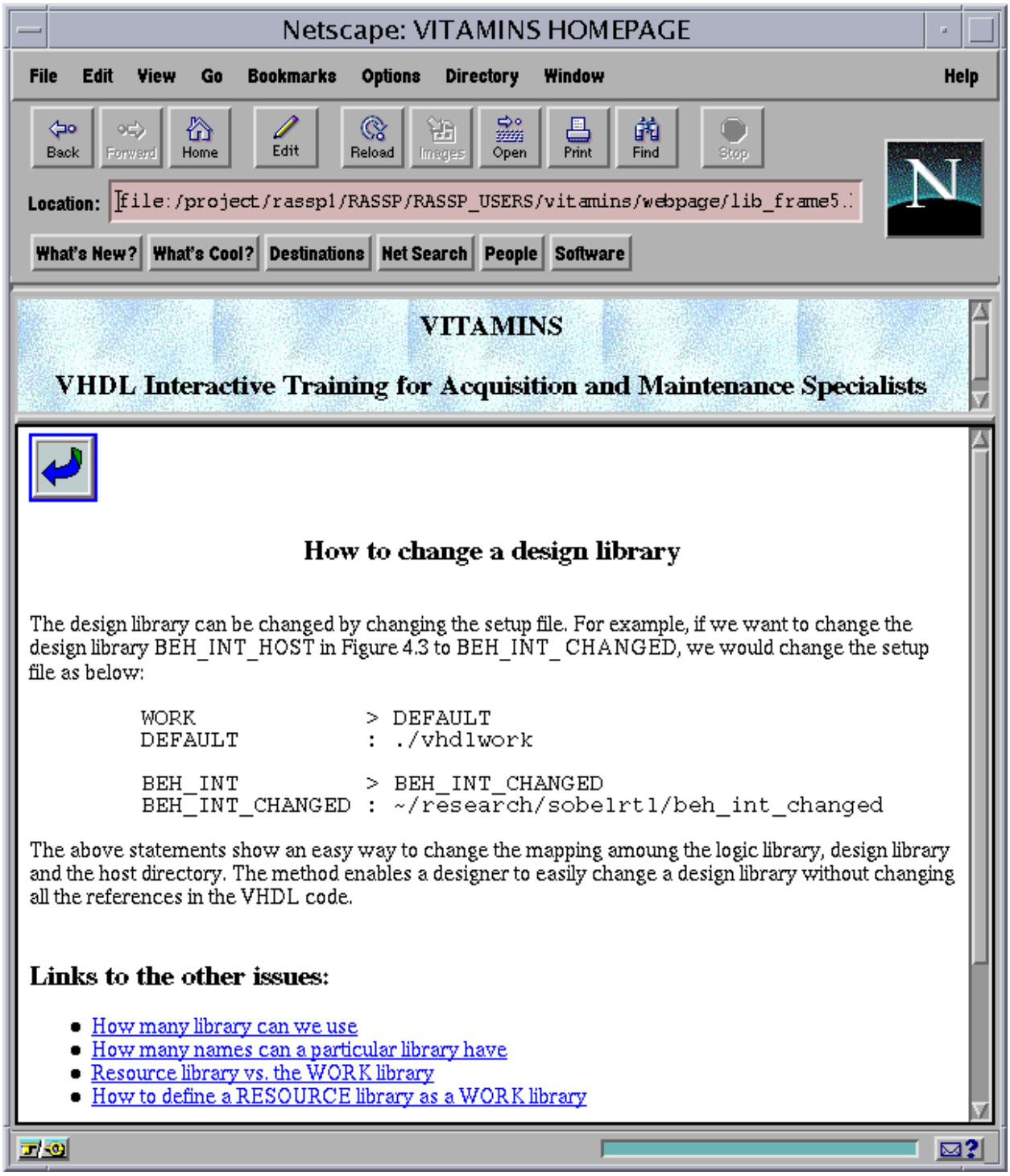



**Figure 5.11** The web page for the issue of changing a design library

# Chapter 6. Conclusions and Future Work

This thesis has presented a methodology to develop a VHDL educational module for the Air force acquisition personnel. The RASSP design concepts are integrated into the teaching material. Effort has been made to explain the many facets and advantages of VHDL and the use of VHDL in various design stages. Design techniques, such as modular design, reuse and sharing of design, multiple abstraction level simulation, mixed data type simulation, and design trade-off are illustrated. The Sobel edge detection system has been used as a case study to make things easy to understand. An interactive training module has been developed based on these materials. The user can easily use a navigational tool to access information on the VHDL models, the RASSP design process, and other useful resources.

Some of the areas in which future work are possible is as follows:

1. Develop an interface for automatically capturing and changing the generics

In this project, we explained the generics and their usage. We discussed that there are two places in the VHDL codes where the actual values of the generics are given: the architecture body or the configuration body. Defining the generics in the configuration body allows design reuse. It is a better solution.

The VHDL models can be developed automatically by the SGE tool (discussed in Chapter 3). When developing a design entity using the SGE tool, the generics can be declared and assigned value by using SGE *Symbol Attribute* tool.

Another research team, TPALS, at Virginia Tech is working on how to automatically generate the test benches. A user interface could be developed for automatically capturing and changing the generics of the design entity in the SGE design environment and added to our educational module.

2. Develop a Performance Model

A performance model could be developed for the Sobel edge detection system to model system level behavior. Different performance statistics such as bandwidth of a bus, percentage

utilization of different blocks in the system, etc. can be studied. Traffic congestion and communication bottlenecks can be estimated. The designer can make a comparison of different architectures and can help decide the best architecture for a particular application [13].

3. Develop Built-in Self-test

Currently, all the test vectors are generated manually and stored in the ASCII files. These test data are read into a frame first and then written to the memory processor. Built-in Self-test could be inserted into the circuit for each component to provide on-line testing.

## References


[1] P. J. Ashenden, *The designer's guide to VHDL*, Morgan Kaufmann Publishers, Inc., 1996.

[2] A. Rushton, *VHDL for Logic Synthesis*, McGraw-Hill Book Company,1997.

[3] *IEEE Standard VHDL Language Reference Manual, Standard 1076-1987*, IEEE Inc., 1988.

[4] J. R. Armstrong and F. G. Gray, *Structured Logic Design with VHDL*, Prentice-Hall, 1993.

[5] N. Kanopoulos, N. Vasanthavada and R. L. Baker, *"Design of an Image Edge Detection Filter Using the Sobel Operator,"* IEEE Journal of solid-state circuits, vol. 23, No. 2, pp1-2, April 1988.

[6] R. Jain, R. Kasturi, B. G. Schunck, *Machine Vision*, McGraw-Hill, Inc, 1995.

[7] *Synopsys Graphical Environment Users Guide*, Version 3.0, December 1992.

[8] S. Gopalakrishnan, *Development of Web-Based Educational Modules for Testing VHDL Models of Digital Systems*, Master of Science thesis, Electrical and Computer Engineering Department, Virginia Tech, August 1997.

[9] *Synopsys Online Documentation*, v3.4b.

[10] R. Lipsett, C. Schaefer, C. Ussery, *VHDL: Hardware Description and Design*, Kluwer Academic Publishers, 1989.

[11] R. Owston, "*The Teaching Web: A Guide to the World Wide Web for All Teachers"*, http://www.edu.yorku.ca/~rowston/chapter.html.

[12] S. R. Hiltz, "*Design a Virtual Classroom"*, http://www.njit.edu/njIT/Department/CCCC/VC/Papers/Teacing.html, 1995.

[13] T. Boutell, *World Wide Web FAQ*, http://www.boutell.com/faq/htedit.htm, 1996.

[14] S. Vuppala, *Methodology for VHDL Performance Model Construction and Validation*, Master of Science thesis, Electrical and Computer Engineering Department, Virginia Tech, May 1997.

[15] W. Hood, M. Hoffman, J. Malley, C. Myers, R.Ong, E. Rundquist, L. Scanlan, F. Shirley, D. Uyemura, " Sander RASSP Program Overview", The RASSP Digrest, Vol.2, 3nd. Qtr. 1995.

# Vita

Weihong Song was born on December 31, 1967 in Shanghai, P. R. China. She got her Bachelor's degree in Precision Instruments in July, 1990 from Shanghai Jiao Tong University, Shanghai, P. R. China. She joined graduate school at the Virginia Polytechnic Institute and State University and received her Master's Degree in Electrical Engineering in August, 1997. After graduating from Virginia Tech, Weihong began employment with COMSAT Laboratory, Maryland.