INFORMATION TO USERS

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

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

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

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

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

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI®

A Methodology for Analyzing Hardware Description Language Specifications of Legacy Designs

by

Claudio Costi
B.Eng, Politecnico di Milano, Italy, 1991
M.Sc., University of Victoria, Canada, 1994

A Dissertation Submitted in Partial Fulfillment of the Requirements for the Degree of

DOCTOR OF PHILOSOPHY

in the Department of Computer Science

We accept this dissertation as conforming to the required standard

---

Dr. D.M. Miller, Supervisor (Department of Computer science)

Dr. Micaela Serra, Departmental Member (Department of Computer Science)

Dr. Margaret-Anne Storey, Departmental Member (Department of Computer Science)

Dr. Nikitas J. Dimopoulos, Outside Member (Department of Electrical & Computer Engineering)

Dr. Ronald J.Bolton , External Examiner (Department of Electrical Engineering, University of Saskatchewan)

© Claudio Costi, 2001
University of Victoria

All rights reserved This dissertation may not be reproduced in whole or in part, by photocopying or other means, without the permission of the author.

Supervisor: Dr. D.Michael Miller

ABSTRACT

In order to increase productivity, methodologies based on reuse of previously designed components are exploited by the Integrated Circuit (IC) design community. However, designers are often faced with the problem of reusing a legacy design for which the behavior is unclear due to missing documentation and the complexity of the design. In this dissertation a methodology to assist designers in retrieving the original intent of a design from its Hardware Description Language (HDL) specification is described. The methodology is based on code analysis and techniques which produce different views of HDL code. These views represent the behavior of a design in more abstract terms than the HDL code.

Examiners:

Dr. D.M. Miller, Supervisor (Department of Computer science)

Dr. Micaela Serra, Departmental Member (Department of Computer Science)

Dr. Margaret-Anne Storey, Departmental Member (Department of Computer Science)

Dr. Nikitas J. Dimopoulos, Outside Member (Department of Electrical & Computer Engineering)

Dr. Ronald J.Bolton , External Examiner (Department of Electrical Engineering, University of Saskatchewan)

Table of Contents

Table of Contents iii

List of Tables vi

List of Figures vii

Chapter 1 Introduction 1

1.1 Scope 2

1.2 Proposed methodology 3

Chapter 2 Background 8

2.1 Integrated circuit design process 9

2.2 Design reuse 12

2.2.1 Component models 12

2.2.2 Design reuse strategies 13

2.3 Design reuse issues 14

2.3.1 Design for reuse 15

2.3.2 Component repository 16

2.3.3 Module interface 17

2.4 Software reverse engineering 19

2.4.1 Architecture recovery 20

2.4.2 Concept recognition 20

2.5 Concluding remarks 25

Chapter 3 Legacy designs and Hardware Description Languages (HDLs) 26

3.1 Legacy designs 26

3.2 Hardware description languages 27

3.2.1 Verilog HDL 28

3.2.2 VHDL 29

3.3 Concluding remarks 31

Chapter 4 A language independent representation 32

4.1 FPDG elements 33

4.2 Extracting region conditions 36

4.3 Mapping HDL code to FPDG 41

4.3.1 VHDL architecture 42

4.3.2 Synthesis constraints and recognition of clock signals 45

4.3.3 Verilog HDL and FPDG 48

4.4 Concluding remarks 51

Chapter 5 From FPDG to signal concepts 52

5.1 Signal concepts 52

5.1.1 Some examples 56

5.2 Extracting signal concepts: signal analysis 58

5.2.1 Preprocessing 59

5.2.2 Signal analysis: algorithm 63

5.3 Concluding remarks 68

Chapter 6 Combinational function concepts 70

6.1 Merging signal concepts 70
6.1.1 Merging two signal concepts 72
6.1.2 Unusual cases 74
6.2 Combinational function concepts 76
6.3 Combinational function analysis 80
6.3.1 Identifying encoder, decoder and multiplexer concepts 80
6.3.2 Identifying demultiplexer and comparator concepts 85
6.4 Concluding remarks 87

Chapter 7 Concept grouping and design partitioning 88

7.1 Equivalences 88
7.2 Extra controls analysis 91
7.3 Functional partitioning 94
7.3.1 Fsm module behavior 94
7.3.2 Complex register module behavior 96
7.3.3 Simple register module behavior 96
7.3.4 Combinational module behavior 98
7.3.5 Algorithm to extract functional modules 98
7.3.6 Recent related literature 101
7.4 Concluding remarks 103

Chapter 8 Interactive exploration: VALET tool 104

8.1 VALET structure 106
8.2 VALET components and user interface 108
8.2.1 Main window: Navigator 108
8.2.2 Module viewer 110
8.2.3 Target Group viewer 113
8.2.4 Abstract Concept viewer 117
8.2.5 VALET implementation 118
8.3 Concluding remarks 122

Chapter 9 Test cases 123

9.1 PIC-16C5X microcontroller 123
9.1.1 Working register (reg_w) 124
9.1.2 Status register (reg_s) 125
9.1.3 Register I/O (reg_io) 128
9.1.4 Final observations 130
9.2 MIPS test case 131
9.2.1 Control component 131
9.2.2 Dmemory component 132
9.2.3 Execute component 133
9.2.4 Idecode component 136
9.2.5 Ifetch component 140
9.2.6 Observations about the MIPS design 143
9.2.7 Comparing results of analysis with actual documentation 145

Chapter 10 Conclusions 148

10.1 An alternative approach 148
10.1.1 Comparing Design Extractor to the proposed methodology 149
10.2 Contributions 151

10.3 Future work 153

Bibliography **155**

Appendix A Glossary **161**

Appendix B Detailed Algorithms **163**

B.1 Algorithm for signal analysis 163
B.2 Find subset of signal concept for encoder, decoder and demultiplexers 170

List of Tables

Table 2-1. Intel microprocessors size (Intel source) 9
Table 4-1. Example of intervals 39
Table 4-2. Intervals of basic expression a 40
Table 4-3. Region conditions for example 1 40
Table 4-4. Compare predicate encoding....................41
Table 4-5. Predicate values and region expressions 45
Table 5-1. Example sig1: signal concepts....................57
Table 5-2. Example sig2: signal concepts....................58
Table 5-3. Values of predicate A 58
Table 5-4. Results of signal analysis 68
Table 6-1. Values of function map for merged signal concepts 73
Table 7-1. Example extra controls analysis....................93
Table 8-1. Embedded abstract concept....................113

List of Figures

FIGURE 1-1. VALET tool integrated in the reuse scenario. 4
FIGURE 1-2. Extraction of abstract concepts (methodology) 5
FIGURE 2-1. Current design process for IC design 9
FIGURE 4-1. FPDG example 35
FIGURE 4-2. FPDG example 1: region condition with data and constant 39
FIGURE 4-3. VHDL code: example1 44
FIGURE 4-4. FPDG: example1 45
FIGURE 4-5. Verilog code: example1 50
FIGURE 5-1. Taxonomy of signal concepts 55
FIGURE 5-2. Example sig1: VHDL code and FPDG 56
FIGURE 5-3. Example sig2: VHDL code and FPDG 57
FIGURE 5-4. Example: Assignment with variable 59
FIGURE 5-5. Example: Assignment with variable 60
FIGURE 5-6. Example: Conditional statement with a variable 61
FIGURE 5-7. Example: Overwritten assignment 62
FIGURE 5-8. Example: Overwritten assignment 63
FIGURE 5-9. Pseudo-code: signal analysis 65
FIGURE 5-10. FPDG+VHDL: signal analysis example 66
FIGURE 6-1. Pseudo-code: merging algorithm 71
FIGURE 6-2. Example merge 74
FIGURE 6-3. Example drivers' conflict 75
FIGURE 6-4. Synopsys implementation of driver confict example 76
FIGURE 6-5. Example: encoder but not for synthesis 78
FIGURE 6-6. Synopsys implementation of encoder_s example 79
FIGURE 6-7. Pseudo-code: combinational function analysis 80
FIGURE 6-8. Pseudo-code for extracting encoders, decoders and multiplexers 83
FIGURE 6-9. Pseudo-code for extracting demultiplexers and comparators 86
FIGURE 7-1. Example equivalence1 89
FIGURE 7-2. Example equivalence2 89
FIGURE 7-3. Example extra controls analysis 93
FIGURE 7-4. Example: not fsm module 95
FIGURE 7-5. Example: fsm module 95
FIGURE 7-6. Example: simple regisetr 1 97
FIGURE 7-7. Example: simple register 2 97
FIGURE 7-8. Pseudo-code for functional partitioning 99
FIGURE 7-9. Complex register example 102
FIGURE 8-1. VALET structure 106

FIGURE 8-2. Navigator..109
FIGURE 8-3. Example of Module viewer..111
FIGURE 8-4. Example of Target Group viewer with no views..114
FIGURE 8-5. Example of Target Group viewer with views..116
FIGURE 8-6. Example of Abstract Concept viewer..118
FIGURE 9-1. VHDL of the component reg_w..124
FIGURE 9-2. Data dependencies amongst modules in reg_w..125
FIGURE 9-3. VHDL of the component reg_s..126
FIGURE 9-4. Data dependencies amongst modules in reg_s..127
FIGURE 9-5. VHDL of the component reg_io..129
FIGURE 9-6. Data dependencies amongst modules in reg_io..129
FIGURE 9-7. Data dependencies amongst abstract concept in reg_io..130
FIGURE 9-8. Comb2 combinational module of execute..134
FIGURE 9-9. A simple register and its connections in the component idecode..137
FIGURE 9-10. Complex register of ifetch..141
FIGURE 9-11. Data-flow dependencies amongst components of MIPS..143
FIGURE 9-12. Error in VHDL code of the component idecode..146

Acknowledgments

I would like to thank all faculty and staff of the Computer Science department who made my long days at Uvic as pleasant as they could be (including the gang of card players). Thanks to my supervisor who, although busy with his duty as dean of Engineering, was always able to find some time to listen to my doubts and requests. How could I forget to mention my colleaugues in the VLSI group who accepted (so to speak) my whining and my never ending talking during our weekly meetings. A special thanks to the real estate agents in Victoria who never found me the right place to open my bakery, so ... I had no choice but to finish my dissertation !!!

At last but not least a message for my mom:
"Sei una mamma meravigliosa, grazie per aver sempre accettato le mie stramberie ... lo prometto questa è l'ultima volta che faccio lo studente."

Chapter 1 Introduction

The steadily increasing complexity of Integrated Circuits (ICs) and increasing market pressure drive the need to continuously improve the IC design process. The introduction of logic synthesis systems [BRSW87] and Hardware Description Languages (HDLs) have moved the design process from gate-level schematic capture to Register Transfer Level (RTL) synthesis. Today, to keep pace with productivity requirements, design methodologies based on reuse of existing functions are exploited.

Design reuse is not a new idea. Standardized components (*e.g.* the 74xx TTL series) have been used in the past to improve productivity of circuit boards; while for microprocessor based systems reuse of RAM, ALU and I/O controllers is widely practiced. In the Integrated Circuit design community, design reuse methodology is based on reuse of components which have already been employed in previous designs and have already been verified. The complexity of such components may vary from a simple combinational logic module to a complex control logic block or even a whole microprocessor.

Today, a repository of Intellectual Property (IP) components is commonly available in any IC design center [KB98]. This repository constitutes a valuable resource for designers. It contains components which represent the legacy of previous designs and which designers may reuse in implementing a new design. To actually reuse a previously designed component, designers must overcome different problems. In the first place, designers must select an appropriate component from the repository. Methods to classify components in the collection and to retrieve a specific component based on given requirements are necessary. Once a component has been chosen, designers need to analyze its functionality in detail.

Designers must evaluate its suitability for a new design with regard to behavioral and interface constraints. Often, changes to a component are necessary to fit specific requirements in a new design. Changes may be made to include new functionality and to properly connect a component to other modules in a design.

1.1 Scope

Any of the above activities regarding reuse of previously designed components requires knowledge about such components. Particularly, designers need to know the functional behavior of a component, its internal structure and organization, as well as its input and output dependencies and usage constraints.

This dissertation focuses on the issue of analyzing the functionality of legacy designs which have been described using a Hardware Description Language (HDL) at the RTL level. Legacy designs are those designs which have been implemented and used in previous designs and which contain valuable Intellectual Property (IP) knowledge. These designs represent components which are reused and included in new designs. Since, quite often, no good documentation or partial documentation is available for legacy designs, designers must attempt to directly reuse HDL descriptions without having other information than their code. Even in cases when the documentation of a legacy design is complete, the effort to understand it and identify conditions for correct use of the component might be as complex as investigating the component description. In those cases, designers are forced to inspect the HDL code of a component to determine its behavior. To help designers during this difficult activity, a new methodology is proposed in this dissertation. The methodology is based on analyses and techniques which produce new views of a HDL code. These views represent the behavior of a design in more abstract terms than the HDL code. The challenge consists of defining and extracting, from a HDL description, a set of artifacts which are meaningful for the hardware designer and which are sufficient to help designers understand and reuse a component.

1.2 Proposed methodology

Analysis of HDL descriptions has been investigated by different researchers. In [NIY96][CFR+98] VHDL (VHSIC - Very High Speed Integrated Circuit - Hardware Description Language) descriptions are analyzed using a technique called *slicing* which was first introduced by Weiser [Wei84] in the software-engineering domain. For a signal assignment line in a VHDL description, it is possible to extract a slice which consists of all the code lines which directly or indirectly affect the assignment execution and value. The slice represents executable VHDL code which behaves like the original code with respect to the chosen signal assignment. The authors showed that slicing may be used to extract a portion of code to be reused and to identify code which is influenced by a modification. While slicing explicitly identifies relations among lines of code, no interpretation of the meaning of the code is obtained. A more semantic driven approach to VHDL code analysis is presented in [BBS96]. The goal is to identify properties of the final circuit from the synthesis and testability points of view. The analyses presented identify signals which are implemented as memory elements in the final circuit, and signals that can be used to propagate test patterns through a portion of the design. These analyses cannot be directly used to understand the function of the component.

Understanding VHDL code is the goal of the research work reported in [LWMG96] where a VHDL reverse-engineering tool-set called VYPER! is presented. In VYPER!, a set of interfaces graphically represents information about the VHDL code including: (i) the control flow of concurrent statements; (ii) the control structure of sequential descriptions, and (iii) the module hierarchy. By means of hyperlinks, the designer may readily investigate the VHDL code and search for information which is necessary to reuse or modify the components under analysis. In VYPER!, it is assumed that, by understanding the actual VHDL code structure, a designer is able to understand the function of a design. Consequently the focus is in helping designers in analyzing and understanding the code itself and there is no attempt to identify what function the VHDL code describes.

In this dissertation, it is assumed that the reuse scenario consists of a complete HDL-based design flow. A repository of components, which are described using synthesizable HDL

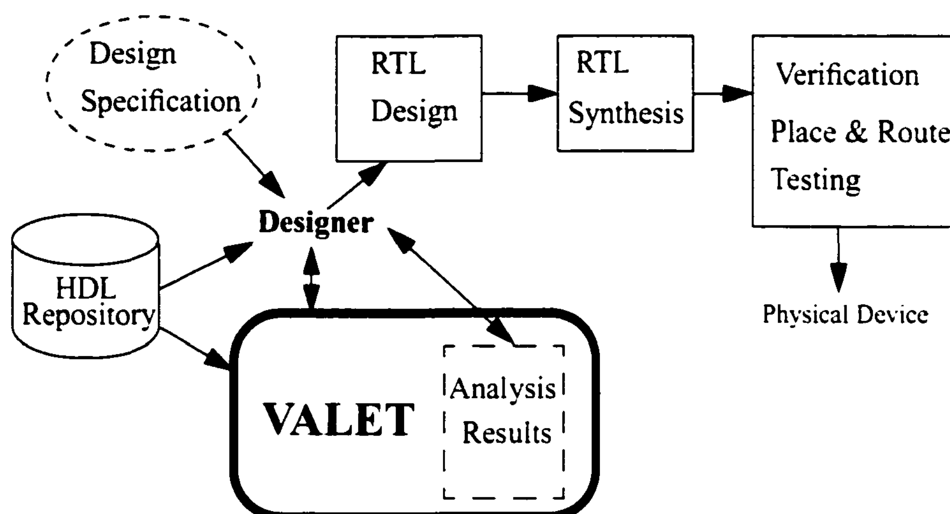


FIGURE 1-1. VALET tool integrated in the reuse scenario.

descriptions, is used. VHDL Assistant Low Effort Tool (VALET) is the prototype which implements the methodology which is proposed in this dissertation. Figure 1-1 shows the integration of VALET into a HDL-based IC design flow. VALET assists a designer during the creation of a RTL design description from a given design specification. By inspecting the results of various analyses performed by the tool, a designer increases his or her understanding of the selected component. Once a designer feels confident about his or her understanding of the chosen component. the associated HDL code may be extracted from the repository and used in the new design.

The proposed methodology aims to assist designers in retrieving the original intent embedded into a RTL synthesizable HDL description of a given legacy design. To achieve such goal, the methodology includes the capability of automatically analyzing HDL. of extracting meaningful information and of delivering the extracted information to designers. The methodology is based on principles which are borrowed from research work in the area of the reverse engineering of software systems. In particular, the idea that understanding is obtained by examining high level abstractions (artifacts), which represent aspects of an analyzed system, is considered. Extracting meaningful abstractions from software code is complicated, as is shown by the research work done in the area of program understanding in software engineering (see references in Section 2.4). However, dealing with synthesizable HDL descriptions is less complicated because the semantic

domain is bounded by the fact that these descriptions represent digital hardware designs. Moreover, the semantic domain is mathematically defined by Boolean algebra, therefore rules of that algebra can be exploited to extract meaningful abstractions. The approach followed in this dissertation consists of defining different types of artifacts which represent behaviors commonly found in digital designs. Given an HDL description, analyses are performed to extract instances of these artifacts. The behaviors represented by these instances and their relationships, represent "how" particular functionality is implemented in the analyzed HDL description.

In Figure 1-2, the steps followed by the proposed methodology to extract a hierarchy of artifacts (abstract concepts) is depicted. To make analyses independent from the use of a specific hardware description language, a language independent representation, called Finer Program Dependence Graph (FPDG), is defined and used (see Chapter 4). An HDL description is parsed and mapped to a set of FPDGs. From this set of graphs, an analysis, called *signal concept analysis*, extracts a set of artifacts called *signal concepts* (see

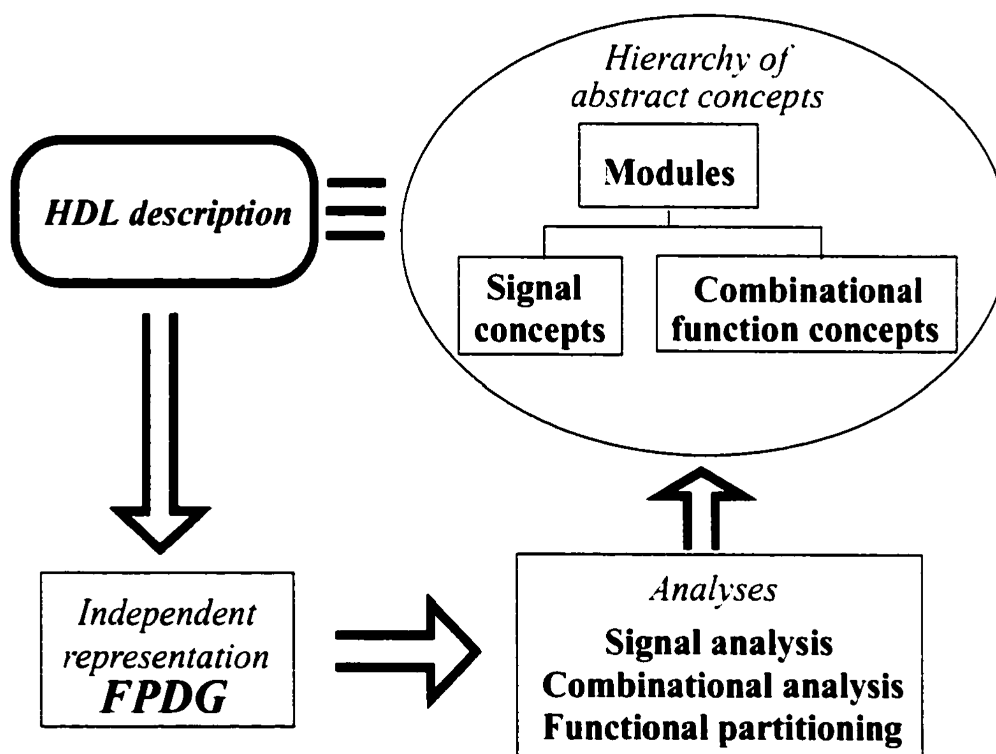


FIGURE 1-2. Extraction of abstract concepts (methodology)

Chapter 5). Signal concepts are abstractions for representing patterns of signal usages which are commonly found in digital designs. Although each signal concept represents a very simple behavior, that is a specific relation among signals, a collection of them may correspond to a complete algorithmic description. Looking at the set of signal concepts and their relationships, the behavior of the original HDL description may be reconstructed. Therefore, the extracted set of signal concepts represents a first level of abstraction obtained by the proposed methodology.

Another analysis, called *combinational function analysis*, which is described in Chapter 6, creates artifacts which belong to another level of abstraction. These artifacts, called *combinational function concepts*, represent basic combinational behaviors. Combinational function analysis extracts these concepts by analyzing and combining signal concepts. Combinational function concepts together with signal concepts, later referred to as abstract concepts, represent the behavior embedded in the analyzed HDL description. To further facilitate the understanding of a design, abstract concepts are grouped in sequential and combinational behaviors. *Functional partitioning analysis*, which is described in Chapter 7, collects abstract concepts in modules. Different types of sequential behaviors are recognized and classified as *fsm*, *complex register*, and *simple register* modules. *Combinational* modules which collect related combinational behaviors are also created. Modules and abstract concepts represent the artifacts which are automatically extracted in the proposed methodology. Modules partition a design into sequential and combinational behaviors. Data flow dependencies exist among modules. Abstract concepts in a module represent "how" a specific sequential or combinational behavior is realized in the analyzed HDL description.

The methodology also includes an exploration activity in which designers, by investigating the extracted artifacts, pursue the goal of understanding "what" functionality is provided by the HDL description. In Chapter 8, the VALET prototype is described. Various viewers and graphical representations, which allow a designer to interactively explore the extracted artifacts, are available. Data flow dependencies amongst modules and amongst abstract concepts are depicted using directed graphs. Each viewer has specific capabilities

to help a designer during investigation. For example, one type of viewer allows a user to explore all abstract concepts which refer to a selected signal.

Some test cases are analyzed in Chapter 9. VHDL descriptions of components are examined using VALET. In this chapter, it is shown that, using VALET, the activity of understanding the functional behavior of a given component is facilitated. In Chapter 10, the contributions of this dissertation are summarized and ideas for future work are given. Moreover, a different approach to extracting artifacts from legacy HDL code, which has been developed by the Escalade company, is also discussed.

Chapter 2 Background

After a brief introduction to the integrated circuit (IC) design process, design reuse is described as a way to deal with increased design complexity. Previous work regarding common issues which are important in dealing with reuse of hardware components is outlined. Research work in the area of software reverse engineering and particularly regarding the concept recognition problem is also described. Ideas from the latter work have been considered in this dissertation to approach the issue of understanding the behavior of a previously designed component for reuse.

Since the early 1970s, when the first available integrated circuits were used to produce commercial devices, integrated circuit technology has progressed at an astounding rate. The number of transistors which can be placed on a chip has doubled approximately every 18 months (Gordon Moore's law [Moo65]).

The first family of integrated circuits was called Small Scale Integrated circuits (SSI) and each chip contained no more than one hundred transistors. Since then, Medium Scale (MSI), Large Scale (LSI), Very Large Scale (VLSI) and Ultra Large Scale (ULSI) families [Dor98] of integrated circuits with up to several millions of transistors on a single chip have been manufactured. As an example, Table 2-1 shows the evolution of Intel microprocessors during the last 22 years. Such levels of integration have been primarily achieved through a continued improvement in silicon process technology which, nowadays, enables the manufacturing of transistors of 0.11 μm in length [Lah00]. However, the current

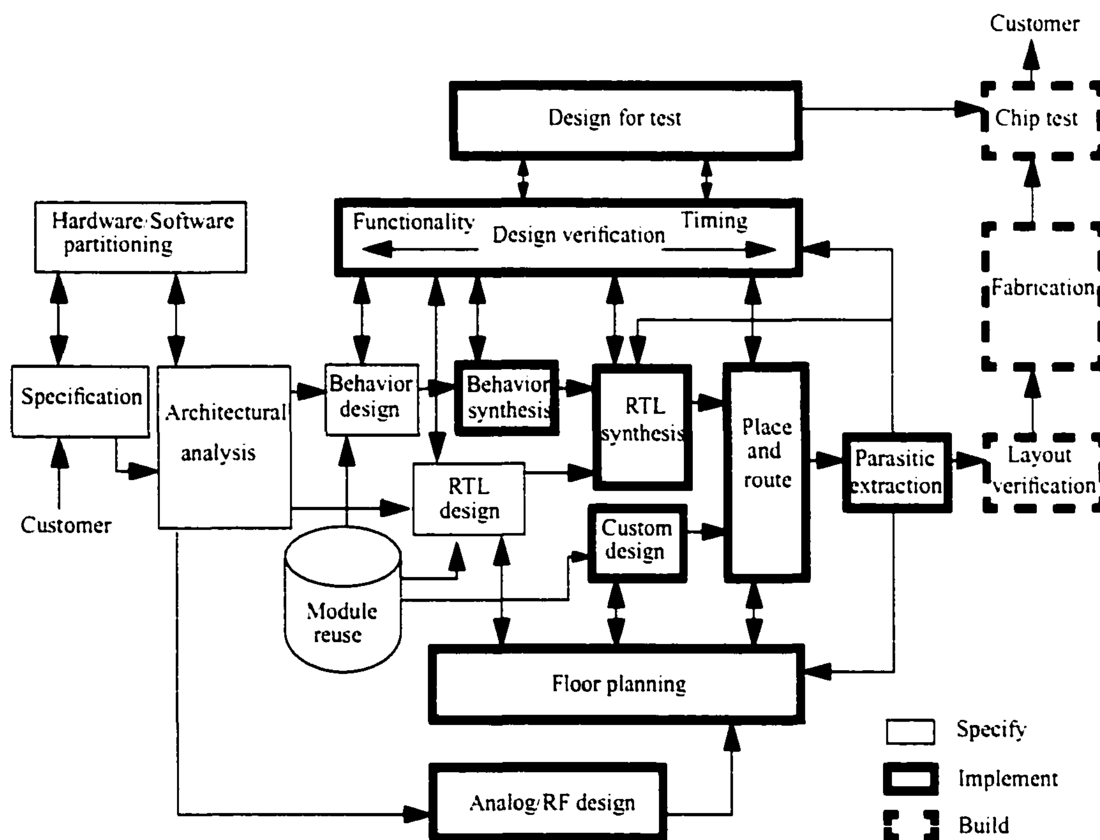
Year	Model	Number of Transistors
1978	8086	29,000
1985	80386	275,000
1989	DX Intel486	1.2 million
1993	Pentium	3.1 million
1997	Pentium II	7.5 million
1999	Pentium III	9.5 million

Table 2-1. Intel microprocessors size (Intel source)

functional complexity of integrated circuits could not be accomplished without improvements in the IC design process as well.

2.1 Integrated circuit design process

SSI circuits were designed by manually drawing, on a large sheet of paper, polygons representing the actual layout of transistors on the silicon substrate. After visual inspection



source: Semiconductor Industry Association

FIGURE 2-1. Current design process for IC design

and correction of the design, the layout was manually transferred to the photolithography masks and then reproduced on the actual silicon slice, or wafer. It was soon clear that such design methodology could not scale to larger designs with a thousand or more transistors. The availability of computers made it possible to improve the design process. Schematic based design methods were introduced first. On a computer screen, designers could directly draw a network of logic gates and simulate it. Modification became easier to handle and larger designs were obtained using hierarchies of components. In the early 80s, the introduction of logic synthesis techniques [BRSW87] enabled designers to implement larger designs and automatically explore different optimized solutions. Since then, further specialized software tools have been developed to assist designers in different phases of the design process.

In Figure 2-1 a block diagram [Gep99], which depicts a typical design process for integrated circuits, is shown. Three main design phases are highlighted in this design process:

- *Specification* phase. The goal is to identify a feasible architecture for the IC and to describe the system function by means of technology independent descriptions.

  The phase starts with customer specifications which are translated into textual and/or graphical formalisms which can be electronically manipulated. Informal as well as formal languages are used. Given the specifications, analysis tools help designers to identify a feasible architecture and to partition the system into hardware and software components. Currently available tools are not able to automatically determine an optimal architecture and the expertise of designers is required to identify a good system architecture. Hardware modules are then described at the behavioral and register transfer level (RTL) levels by means of hardware description languages, while software modules are specified using dedicated software description languages. These descriptions are generally verified against the original customer requirements using functional simulation or animation tools. A library of hardware modules, implemented by an external provider or internally by different design teams during previous projects, is often available for reuse purposes.

- *Implementation* phase. This phase is the core of the IC design process. It consists of multiple activities which cover different aspects of the implementation. The final goal is to implement the IC as a set of directives for the physical building process using a specific technology. In large designs, each activity is generally performed by different designers or teams of designers in a concurrent and incremental mode. First, each behavioral and RTL description is synthesized and a gate level description for each component is obtained. The gate level description is verified regarding functional and timing performances. Simulation, static timing analysis and formal verification techniques are currently used. The gate level network is also analyzed regarding power consumption which is of much concern in current designs for portable devices. Specific design for test structures are inserted in the network to facilitate the testing activity in the construction phase. Once the gate level implementation has been established to be correct, it is translated into transistor and layout level implementation. Based on high-level floorplanning activity which aims to position components relative to each other within the chip area, placing and routing of gate cells, transistors and wires is performed. At this point more detailed area, speed and power analyses are performed and parasitic parameters are extracted. These technology values are backannotated on gate and RTL level descriptions for a further verification. Eventually, the design is re-implemented to improve its characteristics in term of area, speed and power consumption. Some analog parts and specific critical custom modules might be directly implemented by designers using gate cells and/or layout components.

- *Construction* phase. In this last phase the IC is physically fabricated, tested and eventually delivered to the customer. Before fabrication, the layout is verified by checking minimal technology geometries, including width and spacing of routing paths and transistor areas. After fabrication each chip is tested to identify faulty components, that is, errors in the fabrication process.

For each design phase a variety of computer programs, called Electronic Design Automation (EDA) tools, are currently available and new products are continuously appearing on the market to keep pace with complex design requirements.

As described above, during the design process, several intermediate levels of abstractions are introduced to break down the process in various activities. Each level contains different information details regarding the circuit under development [GVNG94].

2.2 Design reuse

Moving design specification to higher levels of abstraction and using EDA tools to automate synthesis and analysis of most design steps, is an efficient way to deal with increased design complexity. For example, the introduction of hardware description languages as well as behavioral and logic synthesis tools, allows designers to improve productivity by working at a more abstract level, and to improve quality by exploiting different design alternatives and optimizations. This methodology has been proven to be effective and is a common practise in today's high tech design laboratories.

However, nowadays, due to time-to-market issues and increasing functional requirements, the above methodology seems inadequate since it suggests to develop ICs with millions of gates from scratch. Therefore, many design teams are considering a block-based design methodology that emphasizes design reuse. A new design consists of a certain number of functional blocks interconnected together. Some of these blocks may need to perform new advanced features and must be designed from scratch, while other blocks may consist of common behavioral modules like processor, DSP, memory or specialized, but previously designed, functions which do not need to be redesigned. By reusing previously designed components, productivity is increased, that is design time is decreased. Designers do not need to re-design part of the system, furthermore better quality is also achieved since the reused components have already been verified.

2.2.1 Component models

In the IC design community, it is commonly accepted to classify a component to be reused in terms of its reusability model [All96]:

- *A Soft model* represents a component which is generally available in the form of synthesizable Hardware Description Language (HDL) code. It has the advantage of being flexible but the disadvantage of not being as predictable in terms of performance.
- *A Firm model* is built using a generic technology. It is optimized for performance and area. It may be available in the form of parameterized generators, netlists or RTL sub-blocks. When specific technology mapping procedures and predetermined floorplanning are associated with this model, the model results are quite flexible and predictable.
- *A Hard model* consists of a netlist fully placed, routed and optimized for a specific technology. Such a rigid and validated layout model, with well defined timing, enables rapid integration at the expense of flexibility. Its performance is predictable but it is less flexible and portable due to process dependencies.

Based on the above classification, a component which is available to be reused is commonly referred to as a *soft*, *firm* or *hard macro* (*component* or *module*) depending on which reusability model it belongs to.

2.2.2 Design reuse strategies

According to [GC93][Gaj88], different design reuse strategies may be applied in an IC design flow, particularly:

- *Design exchange*. A design is reused by reading it in the currently employed design process. Exchanging designs among different design teams is obtained by using standard data formats. A design may be exchanged at different levels in the design process:

  - At the layout level, using GDSII standard format. In this case technology specific subcircuits may be reused assuming that the same technology process is used for a new design.

  - At the gate level, using netlist interchange formats like TDL or EDIF. In this case the logical netlist may be reused assuming that the same library of gates is available in the technology process which is used for a new design. A remapping from a specific library to another may also be performed.

- At the RTL level, using synthesizable functional descriptions in VHDL or Verilog format. The modules may be reused independently from the target technology. Modules may be available as parameterized objects representing a class of circuits. Example of parameters are input, output data widths.

  In this strategy, the reuse components have a fixed function and description which does not need to be modified. Understanding of the component function is required, and particular attention must be taken in setting available parameters.

- *Design evolution*. In this case, the component to be reused is chosen even if its function is not exactly the one needed in the new design. Changes to achieve new functionality and performance are made to the module to be reused. Register Transfer Level and behavioral level descriptions are generally only reused since they are technology independent and easy to modify. In this approach a deep understanding of the component description is necessary to perform the required modifications. It is important that local modifications are done such that the function of other parts of the module is not compromised.

Both strategies require that an appropriate component to be reused is selected from among those available; that the component is instantiated in a new design and that its interfaces with other components is verified. Selecting the appropriate component implies identifying its function and matching it with the required new function. Information about the characteristics of the component must be available or extracted from its description. Instantiating a component consists of inserting and interconnecting it into the current design description. The component design representation should be compatible with the representation of the new design. Finally, the component function should be verified as part of the new design.

2.3 Design reuse issues

In recent years, the IC design community as well as the research community have identified different issues which are related to design reuse strategies.

2.3.1 Design for reuse

In order to have an effective block-based design methodology, components must be specifically designed for reuse. In fact, the degree of effort to reuse a module depends on wheter it has been designed with or without reuse in mind. Design for reuse is a systematic process of designing and verifying modules for their subsequent reuse. Recently a handbook [KB98] has been published which depicts guidelines on how to design components for reuse. General rules are that a component must be:

- Designed to solve a general problem. The module should be configurable, for example in the number of bits accepted as input or output. In this way the module is more flexible and can be reused in more designs.
- Designed for use in multiple technologies. The module must be designed with no specific technology dependent features. For example a soft macro must be supplied with synthesis scripts which allow the designer to correctly implement it with a variety of libraries.
- Designed for use with different tools. This implies that the same module must be available in different formats, for example in VHDL as well as Verilog, to be used with various tools.
- Verified as a stand-alone module to a high level of confidence. A test bench must be written and verification techniques must be implemented which reach a high level of test coverage.
- Fully documented. Functionality, valid configurations and parameters, as well as usage condition and restriction must be well documented. Interfacing requirements and restrictions on how to use the module must also be documented.

In the manual, specific details on the RTL coding style of soft macros and on hard macro design for design reuse are given. Synchronization, timing path, clocking schemes, testing, HDL coding style and other technical factors are considered. Implementing a design for reuse does not involve only technical issues but also business as well as project management issues. In fact, the development of a module with specific design for reuse char-

acteristics, requires more design discipline, more documentation, including process documentation, thoughtful design, and particularly a design for reuse attitude in the design team. It has also been estimated that a typical block for reuse costs 2-3 times the cost of designing the same block for a single use [KB98].

2.3.2 Component repository

Due to the large number of potentially reusable components, adequate libraries (or repositories) which enable designers to easily access components are required. Such libraries should act as easily accessible, modifiable and maintainable database of components. Hundreds of components of varying complexity may be available in the library. Having a manageable repository with utility tools which facilitate the access to components, it is a key factor in the efficiency of a design methodology. Different open challenges exist in building such a library:

- How to keep track of all information required for each module, like parametrized behavioral specifications, design flow information, implementation characteristics and constraints, verification stimuli and testing characteristics.
- How to perform a quick search for a component which matches the requirements. A simple classification strategy is generally not sufficient, "intelligent" search engines must be provided.
- How to handle component versioning, concurrent access and component documentation consistency. A consistent database is necessary to maintain a high quality developing process.

In [KCGW98] a formal specification of a library of reusable components, called Intellectual Property (IP) components, is presented. In the library for each IP, a behavioral description which represents the component's function at a specific level, and design flow element for transforming the behavioral specification to the destination design level (*e.g.* gate) is given. Methods to retrieve a component from the library are presented. Given a formal behavioral specification, it is possible to automatically locate a component with the same, extended or similar specification. Use of parameterized behaviors is possible. Each

component must be archived with the appropriate detailed information following the formal directives.

In another research work [AOS94], feature characteristics are classified and associated to each design object and specification. A similarity function is defined and the user may retrieve all components which best fit a given specification. The similarity concept is important, in fact it is rare that a component which matches exactly the requirement exists in the repository. Since HDL soft macros are among the more flexible reusable components, specific research works have been done in the management of a library of VHDL modules. In the tool set presented in[OAIP98], library, package and configuration concepts, which are specific to the VHDL language, are used to organize the repository of elements to present to the user. No component search utility based on requirements is provided but versioning, team work management, internet and intranet retrieval capabilities are supported. Users have to manually browse through the database to find the target component. Similarly in [PHSM95] and [RR99], a reuse scenario is presented in which parametric VHDL descriptions are complemented by scripts to invoke different tool, by data for timing and functional verification, and by various informal documents. Even in this case, no automatic search for a component is allowed, while some management workflow, like revision control, is included in the database. In [RR99], classification categories for objects in the library are defined to help designers select the required component. A methodology which is based on an object-oriented extension of VHDL (Objective VHDL), is described in [BR99]. A reuse management system is presented which handles classification, modification, storage and retrieval of reusable components. Specific features embedded in Objective VHDL are exploited for design reuse.

2.3.3 Module interface

Once a module is instantiated in a new design it needs to be connected to the rest of the circuit. First, it is necessary to identify the interface protocol of each component. The protocol might be explicitly described in the module specification, or could be extracted looking at each module function. Once the protocols are available the designer must, eventually, implement new logic to match the interfaces among various modules. Interface

synthesis techniques are necessary to automatically, or semi-automatically, implement the required extra logic.

In [COB95], an approach to automatically synthesize channels between two modules is presented. It is assumed that for one module (main module) the interface specification is formally described by means of a protocol flow graph (PFG) and that the characteristics (for example data bus width and latency) of the media between the two modules is chosen and described by the designer. The automatically generated hardware is customized such that it satisfies the media characteristics and the main module interface.This approach is interesting because it requires that only the implementation of the main module must already exist, while the rest of the design is incrementally synthesized.

In [MH95], a more architectural level approach is undertaken. Processors and peripheral devices are considered as building blocks. For each module, the description of its interface must include detailed timing and bandwidth information. Specific algorithms are given which consider generic and memory I/O ports, which are often available in a microprocessor module to implement the required link with a device. This approach is particularly suitable for an embedded system on a chip, where standard modules may be used and their interface is well defined. All modules are already implemented and only the interface among them is synthesized *ad hoc*.

In [SM98], the authors consider the problem of generating interfaces between hardware subsystems which are described using hardware description languages. The link between hardware components is implemented following a predefined architecture which is composed by a finite state machine, an arbiter and buffers for sending and receiving data. A tool, called POLARIS, is presented. The tool requires designers to supply HDL models describing the bus functionality of each component; the names of the ports across which data have to be transferred; and the names of the ports which the interface cannot access. The tool is able to determine, from the HDL descriptions, the conditions for transferring data to or from a component. The extracted information is used to generate the finite state machine inside the standard protocol architecture which enables components to communicate through a common protocol. Further details must be supplied by the designer to iden-

tify the required sequence of data between two modules, such that data are correctly transfer from the transmitting module to the receiving one.

2.4 Software reverse engineering

Today, much of software production consists of maintaining and building upon existing code. Both these activities require an analyst to reconstruct the original software's design, which may be accomplished by understanding the software system. As stated by Ted Biggerstaff [BMW93, page 482]:

> *"A person understands a program when able to explain the program, its structure, its behavior, its effects on its operational context, and its relationships to its application domain in terms that are qualitatively different from the tokens used to construct the source code of the program ."*

Reverse engineering is a technique for understanding software [Arn93]. Reverse engineering is the process of analyzing a software system to identify the systems's components, their interrelationships and to create representations of the system in another form or at a higher level of abstraction [CC90].

It has been shown that program understanding is an NP-hard problem [WY96]. Often, the main objective is to gain sufficient design-level understanding to aid maintenance, to enhance a previous product with new features or to support replacement of a system. Reverse engineering methods and tools, which focus on extracting information from program code, concentrate on two types of artifacts [GC99]: structural and functional abstractions.

- *A structural abstraction* is an artifact which is based on the syntactic properties of a programming language. For example, encapsulation of a sequence of programming statements into a module is considered a structural abstraction.

- *A functional abstraction* is an artifact which is based on the semantics of a programming language. For example, the high level description of the function of a module, which is composed of a collection of programming statements, represents a functional abstraction.

2.4.1 Architecture recovery

Architecture recovery, as defined by [Kos99] is a discipline of reverse engineering which aims to recover artifacts, in the form of structural abstractions, from a given software system. Elements likes functions, procedures, global variables and constants, and user-defined types are extracted from the program's code. Considering data and control dependencies, coupling and cohesion information, and other criteria those elements are collected in modules and subsystems to form views of the analyzed software systems. These views represent software structures which may be used by a software engineer to form mental models when designing, documenting or analyzing a system.

2.4.2 Concept recognition

Concept assignment is the process of recognizing concepts within a program. The understanding of a program is obtained by relating the recognized concepts to portions of the program [BMW93]. Functional abstractions are equivalent to abstract concepts defined in [KNE92]. These concepts represent language independent ideas of computations, and they may be further classified in:

- *programming concepts* that include general coding strategies, data structures and algorithm;
- *architectural concepts* that are associated with interfaces to components of the execution environment such as operating systems, networks, database, etc.;
- *domain concepts* that are application or logic function implemented in the code.

Different automatic and semi-automatic reverse engineering techniques, which aim to extract functional abstractions from a given software system, have been proposed.

- *Plan based approaches*

  Most automatic approaches explicitly use a library of predefined concepts, which are represented by *plans* (referred to as *clichès* in [RW90b]), and locate instances of these concepts in the code. The library may be organized as a hierarchy of simple (lower level) and compound plans (higher level). A simple plan is made of components that are directly obtained from the code, while a compound plan contains components which may be plans (simple or compound). The goal is to localize as many higher level concepts as possible in the source code, allowing partial recognition. That is, generally, the aim is not to recognize the entire program as an instance of a single concept but as a set of concepts. A plan may be *localized* when it is located in contiguous sequences of code or *de-localized* when it is located in non-contiguous sequences of code [GC99]. In the following some research works, which explicitly use a library of concepts and employ different heuristic algorithms to locate plans in code are indicated. Both top-down and bottom-up search strategies are used to implement the mapping process. A top-down strategy searches for code which matches a specific concept, while a bottom-up strategy looks for concepts that match a specific set of code elements. We next examine some approaches which use libraries of plans or clichès.

  The PAT [HN90] system is based on a human expert's analysis model. It uses a deductive-inference-rule engine to perform program analysis. PAT extracts program events in the form of object-oriented abstract representation using a bottom-up strategy. These events are programming concepts like assignments, loops, function calls; or data structures like stacks, queues, trees; or standard algorithms like searching, and sorting. Events which correspond to programing concepts are extracted directly from the source code. A library of programming plans is used to identify higher level program events from those lower level program events. Each program plan represents rules which are used by the deductive-inference-rule engine and which encode the understanding knowledge. Program plans are designed to include some heuristic knowledge for diagnosing common coding errors. In this way, it is possible to infer some knowledge even from incomplete or buggy programs.

A top-down strategy is employed by the Recognizer system, described in [RW90a], which is based on clichė and graph-parsing techniques. A program is translated into a language-independent graphical representation called *plan calculus*. The plan calculus is a hierarchical graph structure whose nodes denote operations and tests, and edges denote control flow and dataflow information. A clichė represents commonly used programming structures and algorithms like list enumerations, binary search, sort algorithms; and it is defined by a plan graph. Using graph-parsing algorithms, clichės are mapped to subgraphs of the plan calculus representing the program. As a result, the Recognizer produces a design tree which represents the program. Elements in the design tree are clichės which have been recognized in the program. From the design tree the system is able to generate documentation by combining textual templates associated with the recognized clichės.

In [KNE92], concept recognition is applied to the process of transforming a program, that is to the process of rewriting one program into another by repeated application of a set of transformation rules. A library of plans and a rule based system is used to recognize concepts in a bottom-up fashion. Then, horizontal transformation rules which are based on recognized concepts, are applied to achieve specific maintenance goals like platform migration.

- *Plan mapping approaches as Constraint Satisfaction Problem (CSP)*

  Two works [WY95] and [QYW98] formulate the problems of mapping code to a specific plan or vice-versa as instances of a Constraint Satisfaction Problem (CSP) which is an NP-complete problem [Mac77]. A CSP consists of three components: a set of variables, a finite domain value set for each variable and a set of constraints among variables. A solution for a CSP is a set of domain value to variable assignments such that all inter-variable constraints are satisfied. The plan recognition problem in program understanding has a closed perception. That is, the source program under consideration contains all available information and exact coverage must exist for a plan to be recognized. Consequently, it is possible to map the plan recognition problem to an instance of CSP. In [WY95] program components are mapped to variables of a CSP. Plans which may explain each component are domain values, while constraints are either rules to

form larger plans from smaller plans or rules which describe how program components are structurally related. In a different manner in [QYW98], a plan is composed by actions which are mapped to variables in a CSP. The set of domain values for each variable is the set of program components and sub-plans already recognized. Constraints between actions of a plan correspond to inter-variable constraints in a CSP. A large variety of methods exist to solve instances of CSPs [MMH85] [Nad89][Pro93] which allow a systematic study of plan mapping on a spectrum of heuristics.

- *Grouping based approaches*

  There are research works which do not make use of a library of concepts but extract concepts by grouping elements in a program.

  In [BL91], concepts are extracted as abstractions in object-oriented and functional notations. Such abstractions express the functionality of examined code. The program functionality is expressed by means of classes, objects and processes. Classes are formed as extension of predefined abstract data types, like files. Methods of classes are identified by looking at operations among variables of each data type, and by postulating invariants to be proven by a logical reasoning tool. Sometimes, hints from a designer are needed to identify invariants and consequently determine the methods. A pattern matching technique is used to express operations on data in a normalized form.The program is rewritten using calls to objects of identified classes. This approach targets data processing applications written in COBOL and scientific applications written in FORTRAN. It is based on formal method, particularly logical reasoning, but it relies on specific features of the target applications to extract abstractions.

  Based on programming analysis techniques such as control flow and data dependency analysis, the work in [BR97] produces a set of abstraction called *candidate chunks*. Intuitively a chunk is a sequence of software instructions which achieve a coherent purpose. The described software tool generates chunks in two steps. In the first step, a set of possible chunks is created by analyzing each statement and each definition-use relation among variables at a specific control level in the code. Each potential chunk may be seen as a function with input parameters being variables which are read, and with output parameters being variables which are written and depend on input parameters. In

the second step, heuristics are used to assign a weight to each possible chunk. The heuristics are based on size and connectivity attributes. For example one heuristic assigns high weight to smallest chunks that produce single output. Potential chunks with higher weight are selected as candidate chunks. Those chunks are evaluated by a user who must map them to program/problem domain concepts and assign them a conceptual description. Clearly, in this work, while an automatic process identifies candidate chunks, the actual concept mapping is performed by the user who has the necessary domain knowledge.

Automatic approaches which try to recognize instances of a library of known plans, clichés or chunks have different limitations. They require large libraries, in fact every domain has its own domain-specific design elements, each of which require a set of patterns to represent its different implementations. New libraries must be implemented for each novel application and domain. Automatic methods are based on computationally expensive algorithms, inference engines or heuristics and, therefore, they tend not to scale well with the size of a program. Moreover, automatic approaches derive knowledge from the structure and semantic of a specific programming language without considering informal clues and pre-defined domain knowledge of a user.

- *Semi-automatic approaches*

  Semi-automatic approaches which try to solve some limitations of automatic approaches have been proposed in some research works. In those approaches, user interaction is an integral part of the understanding process.

  In [BMW93] an opportunistic, non-deterministic approach is considered for recognizing concepts. The author refers to human oriented concepts, that is computational intents which contain a rich context of knowledge about an application domain and which are expressed in human oriented terms. Such concepts could not be inferred using algorithmic reasoning but plausible reasoning is necessary. Features like natural language tokens in the code, proximity of statements, design conventions, application domain design conventions are used in plausible reasoning. A program understanding assistant, called DESIRE, is described. It is composed of naive and intelligent assistant

facilities. The naive assistant part assumes that the user is the intelligent agent and its facilities provide computationally intensive services, like slicing [HR92], text search, call graph, dataflow dependencies, group clustering. The intelligent assistant part is based on a connection-based inference engine which is driven by a domain model. The domain model is built as a network of concepts (nodes) connected with weighted edges. The program code is examined for relevant evidences such as syntactic clues, lexical terms, clustering clues. These evidences activate node in the network representing a domain model. A concept is recognized when some threshold of confidence is reached.

The DECODE system, described in [CQ96], is a cooperative program understanding environment. It contains an automated program understanding component which recognizes domain independent plans to produce an initial knowledge base. A structured notebook component which provides the user a graphical view of initial understanding is also available. Users may expand the initial knowledge by linking arbitrary source code fragments and then performing further conceptual queries. A database of knowledge is used in which automatically extracted design information are recorded and which may be edited by a user for augmenting the design information. During editing a user may define a new design concept and associate it to source code or previously recognized domain independent plans. While adding this new knowledge the system creates appropriate links which are used during conceptual queries activity. A user may inquire the system about the purpose of a specific piece of code, about which part of the code refers to an identified design element or which part of the code has not been linked to any design concept.

2.5 Concluding remarks

The information given in this chapter provides the context for this dissertation. Design reuse and particularly its related issue of understanding the functional behavior of a block to be reused represents the problem attached in this dissertation. Soft blocks which are described using hardware description languages are considered. In this dissertation, ideas from work on program understanding are exploited. Particularly, the notions of abstract artifacts and concept recognition are used.

Chapter 3 Legacy designs and Hardware Description Languages (HDLs)

The definition of legacy designs and the reason for their existence is given in the first part of this chapter. Subsequently, a brief description of the most commonly used hardware description languages is presented. The goal of the chapter is to introduce the reader to the format and types of components which are considered and analyzed in this dissertation.

3.1 Legacy designs

Design reuse methodology is based on the availability of previously designed components to be reused. It means that a design team should have access to a repository of designs which were previously developed and which represent a collection of proprietary behaviors. These components are often referred to as Intellectual Property designs (IPs). According to [KB98], at different companies, there are three models which have been used for developing IP components:

1. *IP-based.* A dedicated design team implements a component using specific design for reuse rules. The component is designed and documented such that it may be easily reused. Components which are developed by the dedicated team are collected in a repository and they are shared and used by different design teams inside a company. Sometimes, a technical department is appointed to support new users of a component.

2. *Rework-based*. A block is designed for a specific project and, after completion, a dedicated team re-engineers the design to make it reusable. In this case, design for reuse rules are considered only when a component is re-designed by the dedicated team. The newly designed component is included in the company's IP repository and made available to other design teams.

3. *As-Is*. A component is designed as part of a specific chip-development project and put in the company IP repository. No design for reuse rules are considered in designing the component. The quality and reusability of the component depends on the design efforts of the specific design team in the project

Using the first or second model, a design center may achieve a high level of reusability of its intellectual property, but investments in resources, knowledge, time and reorganization are required. For these reasons, often, the *as-is* model is followed and, unless specific discipline is enforced on using design for reuse rules, the company IP repository ends up including legacy designs.

Legacy designs are components implemented with no design for reuse in mind, often, poorly documented, but which have valuable IP content. Teams which want to reuse such designs need to identify the original design intent directly from the component description. In this dissertation, I focus on analyzing soft models of legacy designs which are described using Hardware Description Languages.

3.2 Hardware description languages

In the early 1980s, Hardware Description Languages (HDLs) were introduced as part of the IC design process. In those days, IC design complexity was reaching 100K gates and the gate level description, usually in the form of schematics, became unmanageable. It was necessary to identify methods to express designs in more abstract ways. In addition, logic synthesis technology [BRSW87], which could automatically perform gate level implementation, became available. Hardware description languages were defined to provide behavioral as well as RTL level specification and as an entry point to logic synthesis tools. Like programming languages, HDLs allow designers to express the circuit's behavior by

means of high level statements, that is independently from an actual technology and implementation. These descriptions can be simulated to verify that what they described is what is intended by the designer. Currently, there are two HDLs which are commonly used in industry: Verilog HDL [VER95] and VHDL [VHD93].

Direct comparison between VHDL and Verilog languages can be found in the literature [Mag92]. Advantages and disadvantages of using one or the other language for designing digital systems have been discussed. For this dissertation, it is important that, regarding modeling of IC hardware designs at RTL level, the expressiveness of these two languages is equivalent.

3.2.1 Verilog HDL

The Verilog Hardware Description Language, usually called just Verilog, was designed at Gateway Design Automation in 1984 and 1985 [Bha97]. In 1986, the company merged with Cadence Design System Inc. and together produced the Verilog-XL simulator. At that time it was a very efficient method for doing gate level simulation. In 1991, after other languages were proposed on the market, Cadence transfered the Verilog language from proprietary to public domain through the Open Verilog International (OVI) organization. In 1995, Verilog HDL become a recognized IEEE standard (IEEE 1364 [VER95]). Currently, international working groups exist which are promoting extensions to the language in the areas of analog mixed signal specification and formal verification.

The Verilog language was developed to describe mainly IC hardware systems. It provides designers with behavioral constructs which are useful for the early specification stage of a design, and structural constructs for the later implementation stage. It allows designers to represent their designs in the familiar gate and switch level description as well as behaviorally [TM91]. Verilog has built-in predefined hardware net types (like wire, wired-or, wired-and, tristate) as well as gate and switch level devices (like logic gates, transistors, nmos, pmos, cmos, pullup and pulldown elements). In a Verilog description a digital design is structured by means of modules which are instantiated into other modules to form a required hierarchy. A *module* represents a logical unit which, generally, describes a

piece of hardware or a test generation function. Each module is characterized by a name; its external interfaces, consisting of input output and bidirectional ports; and its content. The module content is expressed using concurrent and sequential domain statements. Concurrent statements are executed asynchronously, without a defined order, and are mainly used for dataflow (continuous assignments) and structural descriptions (instances of modules or built-in hardware elements).

Sequential statements are encapsulated in two specific concurrent statements called *always* and *initial*. The statements inside the *initial* statement are executed only once, while the code inside the *always* statement is continuously repeated. Event control statements, like positive and negative edge detection, and *wait* statements may be used to control the activation and the suspension of an *always* statement. Available sequential statements are executed in order and they consist of the usual conditional, assignment and loop statements which are found in most procedural languages. Since Verilog syntax is based on the C language, the sequential statements follow the C syntax. Functions as well as procedures (called *tasks*) are also elements of the language.The Verilog language is defined by a simulation semantic, that is, each statement is described by how it has to be executed by a simulator for the language.

3.2.2 VHDL

VHDL (VHSIC - Very High Speed Integrated Circuit - Hardware Description Language) was designed by Intermetrics, IBM and Texas Instrument under exclusive contract of the American Department of Defense in 1985. Since 1986, it has been considered by the IEEE Computer Society as a possible industry standard language for describing digital circuits. In 1987 it was approved as IEEE 1076 standard [VHD87]. Lately in 1993, the standard was reviewed and a new standard adopted [VHD93].

VHDL was originally intended to serve two main purposes. First, as a documentation language for describing the structure of complex digital systems. Second, as a means to model the behavior of digital systems and to simulate them. Since the early 1990s, different IEEE working groups have been formed to extend the language and promote different

applications for it, including its use for logic synthesis. Extensions like object oriented, math, microwave, system and test have been proposed. Recently, in March 1999, an extension of the VHDL standard for analog and mixed-signal has been approved as IEEE 1076.1 [VHD99].

In VHDL, a design is partitioned into hierarchically related modules with explicitly defined interfaces, called entities and packages [Per98]. An *entity* represents a logic unit interface and consists of a name; input, output, bidirectional, buffer port declarations; and parameters (called *generics*). A *package* is a repository of declarations, including user defined data types and subprograms (like functions and procedures). To each entity one or more contents can be associated by means of the *architecture* construct. The architecture constructor is used to describe the behavior of a hardware component, or to specify a test generation behavior. Concurrent as well as sequential domain statements are available to describe an architecture. Concurrent statements are executed asynchronously, without defined order, and are mainly used for dataflow (concurrent signal assignments), structural (instances of components) and behavioral descriptions (process). Sequential statements are used only inside a *process* statement. A *sensitivity list*, which consists of a set of signals, may be associated to a process. In this case, during simulation, the process is activated only when an event occurs on one of the signals in the sensitivity list. If no sensitivity list is specified for a process, the process is always activated and continuously repeated. A *wait* statement exists which may be used to control the activation and the suspension of a process execution. Available sequential statements are executed in order inside a process and they consist of the usual conditional, assignment and loop statements which are found in most procedural languages.

The structural hierarchy of a design is explicitly defined by a designer by instantiating components. A component is composed of an entity and one of its associated architectures. Other structuring capabilities exist in terms of different language constructs. In fact, besides the common functions and procedure statements, the *library* construct enables one to collect entities and packages under a common name scope. The *configuration* construct enables explicit binding of entity and architecture to components. In this way, for the same

system design more than one configuration may be defined to select different abstraction models for existing components. Furthermore, statements like *assertions* and *report* exist for handling specific reporting activities during simulation. Syntactically, VHDL is inspired from the ADA programming language therefore it is a strongly typed language. The benefit is that automatic type checking can be performed at compilation time, and possible inconsistency errors may be identified. The VHDL language semantic is based on simulation execution, that is each statement is described by how it has to be executed by a simulator for the language.

3.3 Concluding remarks

In this dissertation, we focus on the analysis of VHDL descriptions at register transfer level (RTL) which may be synthesized by an automatic logic synthesis tool, like *design compiler* by *Synopsys* [Syn00]. Therefore we are dealing with a subset of VHDL constructors and a restricted use of some of the available statements. Particularly VHDL descriptions which satisfy syntax and semantic rules described in the IEEE P1076.6 standard document [IEE99] are considered. Detailed information about the restricted usage of VHDL constructs will be given in Section 4.3 where mapping of VHDL statements to an internal representation is described.

Chapter 4 A language independent representation

In this chapter, a graph which is used to represent the algorithmic behavior of a hardware description language specification is defined and fully described. Since this dissertation deals with synthesizable designs, rules related to the use of hardware description languages to describe synthesizable blocks are considered. The mapping of VHDL and Verilog constructors into such a representation is explained. How region conditions are obtained as Boolean expression is also explained. Region conditions are important since they are used in the signal analysis which is described in the next chapter.

A HDL specification consists of one or more text files containing statements which describe a circuit's behavior. In its text form, HDL code is not a proper representation for the kind of algorithmic analyses and manipulations which are investigated in this dissertation. Therefore a model which retains all code information and which facilitates algorithmic analyses is defined and used.

In the hardware engineering literature, control flow graphs (CFG), data flow graphs (DFG) or a combination of the two graphs are often employed for the analysis of HDL descriptions [GVNG94]. Given some HDL code, its associated CFG explicitly represents the sequencing of operations in the code. The CFG is often used for scheduling of resources during high level synthesis activity and for synchronization analyses. DFG holds detailed

information about dependencies among data and operations. DFGs are often used for testing analyses and resource allocation activities. In the software engineering literature, the program dependence graph (PDG) is commonly used [HR92]. The PDG combines control flow, data dependency and control dependency information in a single representation.

In this research, an HDL description must be analyzed at different level of detail, therefore data and control dependencies as well as control and data flow information are required. Moreover, a specific hardware related element, *i.e.* clock, must be explicitly represented. In this dissertation I propose a new graph representation, called Finer Program Dependence Graph (FPDG). The FPDG is a PDG which has been enriched with some data-flow information and an explicit attribute representing a clock.

4.1 FPDG elements

A FPDG is a directed graph composed of different kinds of nodes which are connected by several kinds of edges. The graph has a starting root node.

- **Statement node**. A statement node is depicted by an ellipse in the FPDG, and represents an assignment statement. A *sub-graph* is associated with the node. This sub-graph explicitly describes data-flow relations between a target signal and signals or variables which are used in the expression which is assigned to the target. The sub-graph is composed of nodes representing signals or variables, and directed edges representing data-flow information.

- **Predicate node**. Predicate nodes, which are depicted as squares in the FPDG, represent conditional statements and contain *conditional expressions*. There are two types of predicate nodes which identify a two-way conditional statement or a multi-way conditional statement. In a two-way conditional statement, the expression may assume only the values: true and false. In this case, a predicate node has only two outgoing edges. The *if-then-else* statement is a good example of a two-way conditional statement. In a multi-way conditional statement, an expression may assume different values. In this case, a predicate node has one or more outgoing edges. The *case* statement is a good example of a multi-way statement.

- **Region node**. A region node, which is depicted by a circle in the FPDG, is the entry point for a set of statements which depend on the same condition. Region nodes divide the FPDG into sub-trees. Each sub-tree has a region node as its root node. A Boolean expression is assigned to each region node. This Boolean expression, called *region condition*, represents the condition for which statements in its sub-tree are executed. Furthermore, an attribute is associated with each region node to indicate whether or not the elements of the associated sub-tree belong to a clocked portion of a design.
- **Control dependence edges**. These are directed edges which explicitly identify control dependencies among nodes in the FPDG. If no label is associated with a control dependence edge, it means a direct dependency between a source and a sink node always exists. A true (T) label, means that the dependency between a source node (predicate node) and a sink node (region node) exists when the conditional expression associated to the predicate node is true. Similarly, a false (F) label, means that the same dependency exists when the conditional expression associated to the predicate node is false. A label consisting of an expression is used to identify the value for which the dependency between the source node (predicate node) and the sink node (region node) exists. Following control dependence edges through a FPDG and considering conditional expressions from visited predicate nodes and labels, it is possible to determine conditions under which assignments are activated.
- **Control flow edges**. These are directed edges which explicitly indicate the order of execution of statements. By traversing a FPDG following these edges it is possible to directly reconstruct the sequence of statements. In most of this dissertation, to reduce the number of edges drawn when depicting a FPDG, the control flow edges are implicitly defined. In fact the children of a region node are ordered such that from a region node, control moves to the left most child of that region, then from left to right among siblings until it reaches a predicate node. At the predicate node, control flows along outgoing control dependence edges. When a leaf node of the predicate node sub-tree is reached, control goes on from left to right among siblings of the predicate node.

- **Data dependence edges**. These are directed edges which connect nodes with data dependencies. In particular, data dependence edges represent dependencies between assignments to a variable and uses of the variable. The edges are obtained by solving the data flow analysis problem: *reaching definition* [Muc97, Chapter 8]. Edges connect statement nodes to nodes in sub-graphs. In this way, connections between data and fine data flow information are established.

In Figure 4-1, an FPDG is depicted. The region node *R0* is the root node of the whole FPDG and is the starting point of the algorithm which is described by the graph. Region nodes may have only statement or predicate nodes as children, while predicate nodes's children are always region nodes. Statement nodes are always leaves of a FPDG. The conditional expression *clock*='1', in the predicate node *P2*, is a two-way condition therefore two outgoing control dependence edges exists. At the control dependence edge which is labeled with false (F), the region node *R3* with no children is connected. The lacking of children explicitly indicates that no activities are executed when the condition *clock*='1' is false. In this example, control flow edges which do not coincide with control predicate

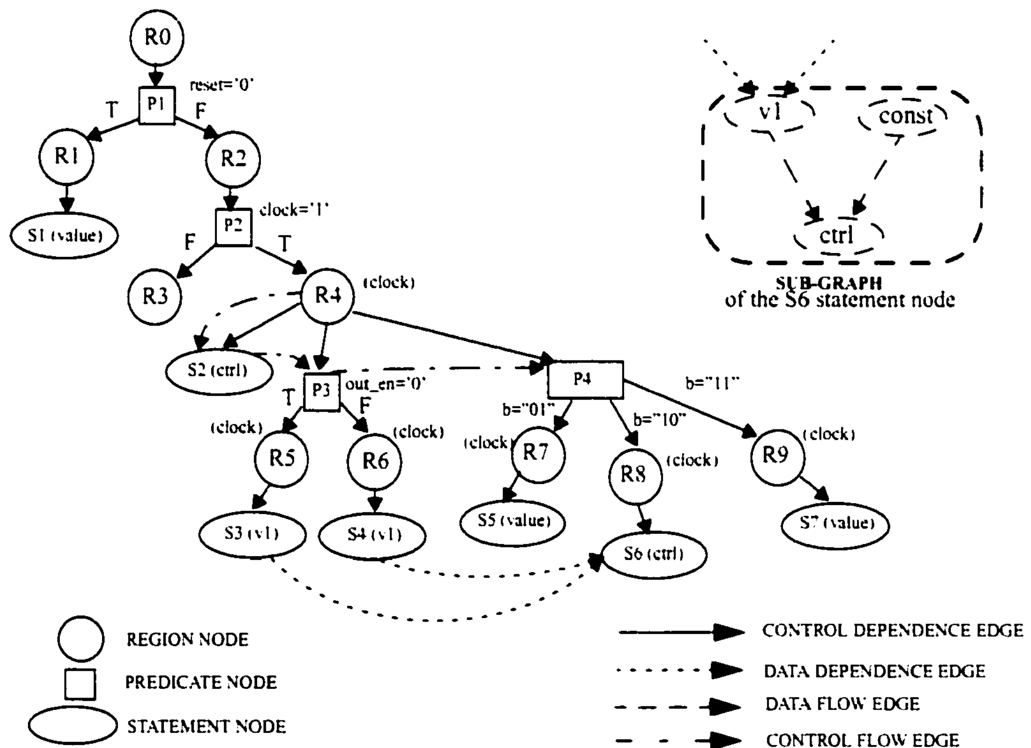


FIGURE 4-1. FPDG example

edges, are explicitly drawn. Three control flow edges are depicted to explicitly indicate that a sequence dependence exists among nodes *S2*, *P3* and *P4*. In fact, under the condition identified in the region node *R4*, the statement node *S2* is executed before the predicate node *P3*, and when the last one has been considered the predicate node *P4* is evaluated. For the remainder of this dissertation, control flow edges are not drawn since they are explicitly defined by the order of the child nodes of a region node. In Figure 4-1, the sub-graph of the statement node *S6* is also depicted. The sub-graph shows that the target *ctrl* has data-flow dependencies with the variable *v1* and a constant value (indicated by the keyword *const*). It means that this sub-graph represents, for example, an assignment like *ctrl <= v1 + "010"*. The two incoming data dependence edges to the statement node *S6* express the fact that the target signal of the statement node *S6* depends on the value obtained from the statement nodes *S3* or *S4*. Particularly, as shown in the sub-graph, the dependence is due to the variable *v1*. Finally, the attribute *clock* (in brackets in the figure), which is associated to region nodes *R4*, *R5*, *R6*, *R7*, *R8* and *R9* indicates that actions performed by children of those region nodes belong to synchronous parts of the specified design.

4.2 Extracting region conditions

Starting from the root node of a FPDG and following control dependence edges, it is possible to identify conditions for which a specific statement is executed. For example, referring to the example in Figure 4-1, the statement node *S5* is executed when *reset*='1', *clock*='1' and *b*="01". This condition is obtained by considering conditional expressions of predicate nodes and labels of control dependence edges which belong to the path from the *R0* region node to the *S5* statement node. Instead of calculating conditions for each assignment, conditions are calculated for each region node and saved as region conditions while a FPDG is built. Each of these conditions is obtained by identifying the path from the root node of the FPDG to a region node and by considering predicate nodes and labels of control dependence edges which belong to the path.

In order to be able to easily compare and manage region conditions, a method to represent each of them with a Boolean expression is developed. Since region conditions are derived

from conditional expressions in predicate nodes, the latter ones are built as Boolean expressions.

Since in this research only HDL code which is directly synthesizable by a synthesis tool is considered, some restrictions apply on the form of possible conditional expressions in the HDL code, as indicated by the standard documents [IEE99] and [VSI99]. Taking into account these restrictions, and that:

- for any multi-level logical expression, an equivalent two-level logical expression exists [Kat94], and
- the complement of a relational expression is a relational expression,

each conditional expression (*condExpr*) in a predicate node can be expressed as (using Backus-Naur Form BNF):

```
condExpr ::=      <logTerm> [{<logOper> <logTerm>}]
logOper ::=       AND | OR | NOR | NAND | XOR | XNOR
logTerm ::=       [NOT] <expr> |
                  <expr> <compOper> <expr>
compOper ::=      = | /= | < | <= | > | >=
expr ::=          <basicExpr> |
                  <constExpr> |
                  <composedExpr>
constExpr ::=     '0' |
                  '1' |
                  "0 | 1 {0 | 1 }"
composedExpr ::= <funciton_call> |
                  <expr> <arithOper> <expr>
arithOper ::=     + | - | * | / | mod | rem |abs | &
```

where *<basicExpr>* (basic expression) is a name corresponding to a variable or a signal; *<function_call>* is an identifier representing a call to a function with its parameters.

Before evaluating each expression (*<expr>*) of a logic term (*<logTerm>*), *constant propagation* and *constant-expression evaluation* techniques [Muc97 Chapter 12], which are commonly employed in a compiler, are used in this dissertation. In this way constant value are statically evaluated. For example an expression like *ALU_OPER + "01"* in which

ALU_OPER has been declared as a constant with value "10" is evaluated to constant value "11".

In this dissertation, each logic term (*<logTerm>*) can be represented by values of a *predicate*. The values of a predicate are represented using Boolean variables. In particular, an ordered set $Values_{pd}$ of Boolean products is used to represent all possible values of the predicate *pd*. Three different types of predicate are defined: *simple predicate*, *compare predicate* and *complex predicate*.

- *A simple predicate* is used to represent a logic term composed of a basic expression (*<basicExpr>*) or a relational expression in the form *basicExpr compOper constExpr*. Since HDL specifications considered in this dissertation represent a synthesizable circuit, a logic term composed of a *basicExpr* is equivalent to a relational expression in the form *basicExpr compOper constExpr* where *constExpr* is equal to one (true). The simple predicate is defined by a *basicExpr* and by a list of Boolean variables which are used to specify all possible values of the predicate. To determine the number of required Boolean variables, all logic terms in the examined HDL code where the same *basicExpr* is compared with a constant value (*<constExpr>*) are considered. All constant values are collected. These values are subdivided into non-overlapping intervals which cover the range of possible values for the basic expression. Following the idea in the research paper [CK96], if N is the number of non-overlapping intervals, $m = \lceil \log_2 N \rceil$ is the number of required Boolean variables.

  Each non-overlapping interval is identified by a Boolean product in m Boolean variables. The ith non-overlapping interval is represented by the binary representation of the value i using m bits. For example, assuming that the following three non-overlapping intervals are recognized ["000"],["001" to "110"] and ["111"], each interval is represented by a Boolean product in two variables as shown in Table 4-1.The ordered set $Values_{pd}$ of Boolean products representing non-overlapping intervals of values of the *basicExpr* constitutes the set of values of the simple predicate *pd*. A sum of Boolean products in $Values_{pd}$ can be used to represent any values of the *basicExpr*. It means, that

Interval	Boolean product
["000"]	$\bar{x}_0\bar{x}_1$
["001" to "110"]	$\bar{x}_0x_1$
["111"]	$x_0\bar{x}_1$

Table 4-1. Example of intervals

the original logic term in the form *basicExpr compOper constExpr* can be expressed by a Boolean expression in terms of Boolean variables belonging to the simple predicate. Due to the way in which the non-overlapping intervals are identified, not only the originally considered logic term but each comparison in the HDL code involving the same *basicExpr* and a constant value can be expressed as a sum of products over the m Boolean variables using the same simple predicate.

Consider, as an example, the FPDG in Figure 4-2. The only constant value used in relational expressions with the basic expression a is the value "0010". The range of values for the basic expression a is from "0000" to "1111", because only binary values must be considered, since a HDL specification represents a digital circuit. The range is divided into the non-overlapping intervals ["0000" to "0001"], ["0010"], ["0011" to "1111"]. To represent those intervals two boolean variables ($2 = \lceil \log_2 3 \rceil$), x_0 and x_1, are necessary. Therefore the non-overlapping intervals are encoded as shown in Table 4-2. A simple predicate *pda* with the basic expression a and Boolean variables x_0 and x_1 is defined. Its possible values are: $\bar{x}_0\bar{x}_1$, $\bar{x}_0x_1$ and $x_0\bar{x}_1$; that is the set $Values_{pda}=\{\bar{x}_0\bar{x}_1, \bar{x}_0x_1, x_0\bar{x}_1\}$.It is possible to express the conditional expression in *P1* and *P2* using the simple predicate. The conditional expression in *P1* is expressed by $\bar{x}_0x_1$,

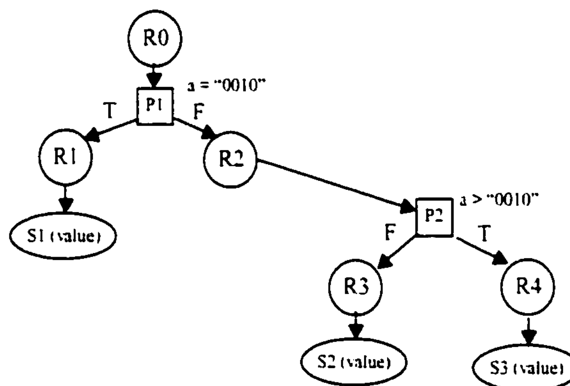


FIGURE 4-2. FPDG example 1: region condition with data and constant

Interval	Boolean product
["0000" to "0001"]	$\bar{x}_0\bar{x}_1$
["0010"]	$\bar{x}_0 x_1$
["0011" to "1111"]	$x_0\bar{x}_1$

Table 4-2. Intervals of basic expression *a*

while the conditional expression in *P2* is expressed by $x_0\bar{x}_1$. By considering paths in the FPDG to each regional node, it is possible to determine all region conditions. Table 4-3 reports those region conditions. The region condition for *R3* has been calculated by negating, because of label *F*, $\bar{x}_0 x_1$ (boolean expression representing *P1* condition) and ANDing the result, $x_0 + \bar{x}_1$, with the negation of $x_0\bar{x}_1$ which is equal to $\bar{x}_0 + x_1$. Therefore $(x_0 + \bar{x}_1)(\bar{x}_0 + x_1) = \bar{x}_0\bar{x}_1 + x_0 x_1$. The product $x_0 x_1$ is not considered because it does not represent any meaningful value of the predicate.

- A *compare predicate* is used to represent a logic term which consists of a relational expression in the form *basicExpr compOper basicExpr*. The compare predicate is defined by two basic expressions and by one or two Boolean variables. The Boolean variables are used to represent the actual relationship between the two basic expressions. To determine the number of required Boolean variables, all logic terms in the examined HDL code, where the two basic expressions are compared, are considered. If at least one comparison including less or greater operators exist, then two Boolean variables are necessary, otherwise only one Boolean variable is used. Table 4-4 depicts the encoding of all possible relationships between two basic expressions. When only one Boolean variable is necessary, the possible values for the compare predicate *pd* are the Boolean products which are shown in the first two row in Table 4-4 and $Values_{pd} = \{\bar{x}, x\}$. Otherwise, when two Boolean variables are necessary, the values for the compare predicate *pd* are the values depicted in the whole Table 4-4 and $Values_{pd} = \{\bar{x}_0, x_0, \bar{x}_1, \bar{x}_0, x_1, \bar{x}_0\}$.

Region	Region expression
R1	$\bar{x}_0 x_1$
R2	$x_0 + \bar{x}_1$
R3	$\bar{x}_0\bar{x}_1$
R4	$x_0\bar{x}_1$

Table 4-3. Region conditions for example 1

Type of comparison	Boolean product
data1 = data2	$\bar{x}_0$
data1 = data2	x_0
data1 < data2	$\bar{x}_1.\bar{x}_0$
data1 > data2	$x_1.\bar{x}_0$

Table 4-4. Compare predicate encoding

Each possible comparison between the same two basic expressions which may be found in a conditional expression of a predicate node may be expressed using values of the defined compare predicate. Note that the condition *data1>=data2* is expressed by $x_0 + x_1$ while condition *data1<=data2* by $x_0 + \bar{x}_1$. All other possible conditions are directly depicted in Table 4-4.

- A *complex predicate* is used to represent a logic term which cannot be represented by a simple predicate or a compare predicate. This predicate is defined by one logic term and one Boolean variable. The complex predicate may assume only two values false or true which are represented respectively by Boolean literal $\bar{x}$ and x, that is $Values_{pd}=\{\bar{x},x\}$. The logic term in the original conditional expression of a predicate node is represented by x. In the case that an identical logic term is found in the conditional expression of another predicate node the same complex predicate is used.

For the rest of this dissertation, the term *predicate* is used to represent all three types of predicate: simple, compare or complex. Since all conditional expressions may be represented by values of predicates, that is by Boolean expressions, all region conditions, which are obtained from conditional expressions, are also represented by Boolean expressions.

4.3 Mapping HDL code to FPDG

FPDG representation has been defined to be language independent. VHDL, Verilog, or other HDL descriptions using only the synthesizable subset of their statements may be parsed and translated to a set of FPDG representations. In this dissertation, VHDL has been used for all examples and test cases since it is a richer language than Verilog HDL.

Nevertheless, specific considerations about Verilog HDL are reported at the end of this section.

4.3.1 VHDL architecture

In VHDL, the behavior of a design or a part of it is described inside an *architecture*. Different concurrent constructors are available to describe an architecture. Some of these constructors are mapped to an FPDG, others are not:

1. *Process*. This is the main constructor which allows designers to describe a behavior as a sequential algorithm. High level sequential statements like if-then-else, case, loops, wait and procedural call are available. Two different types of data are available in a process: *signal* and *variable* data. While variable data behave as expected in any high level language, that is they assume a new value at the moment in which a new assignment is executed, signal data behave differently. Inside a process, an assignment to a signal is scheduled to be executed at the end of the process, therefore uses of such signal, after the assignment, still consider the previous value of the signal, *i.e.* the value of the signal when the process was entered. The algorithm which is described by a process is directly mapped to a FPDG. The root node of the FPDG represents the entry point of a process, and sequential statements of a process are transformed in statement and predicate nodes. Children of a region node are organized such that their order corresponds to the sequential order of statements in a process. Data dependence edges are created only from statement nodes with a variable as target data. In these cases, values are immediately assigned to variables and their use is explicitly indicated. As described above, an assignment to a signal is not immediately executed and therefore its new value is not used at the same execution time. The data dependence information between assignments to signals and uses of signals is considered only later during specific analyses.

   Each procedure call inside a process is expanded and its sequential statements are included in the FPDG of the process as if they would belong to the process itself. When building statement and predicate nodes from procedure code, parameter data are

replaced by the corresponding data which belongs to the calling process. Moreover if, in a procedure code, new variables whose names clash with existing data are defined (*aliasing* problem), such variables are renamed to make them unique.

2. *Concurrent signal assignments*. There are three types of concurrent signal assignments: *simple*, *conditional* and *selected*. Each one of those assignment types is equivalent to a process and therefore is mapped to a FPDG. The *simple* assignment is equivalent to a process with one single statement which corresponds to the original simple signal assignment. The *conditional* assignment is equivalent to a process where nested if-then-else constructors and simple assignment constructors are used. The *selected* assignment is equivalent to a process where case constructors and simple assignments constructors are used.

3. *Procedure call*. This constructor is used to instantiate different copies of a same algorithmic specification. Only variables may be declared in a procedure, and the algorithm is described using sequential statements. Each procedure call is translated in a FPDG in the same way as a process constructor.

4. *Component instantiating*. This constructor is used to instantiate a component as part of a design. Since only structural information in associated with an instance of a component, no FPDG representation is needed and used. The structure of a complete design is considered as a hierarchy of connected entity and associated architectures.

5. *Block*. Generally this constructor is used to group concurrent statements to improve readability. Often, the block constructor has no effect on the construction of the FPDGs for the internal concurrent statements and is not considered. In some cases, the block constructor is employed with a *guard expression*. The guard expression is a conditional expression which controls the execution of the concurrent assignments inside a block. In this case, the mapping of each controlled concurrent assignment to a FPDG must be changed. The guard expression becomes the conditional expression of a predicate node which must be the first child of the root node in the FPDG.

6. *Generate*. This constructor is generally used to control the way components are instantiated. Regular design structures where the same component is instantiated more times

with well defined connections are often described using such constructors. Since this constructor has an impact on instances of components and it is not directly involved in algorithm understanding it has not been considered in this research work.

From the above discussion, it should be clear that, given a description of an architecture, a set of FPDG representations are created which contain all information about the algorithmic parts in the architecture. A different structure is also maintained to keep track of structural information regarding the design. However, in this dissertation, I concentrated on the problem of understanding which functionality is described inside an architecture, therefore relations among instances are not considered during each analysis. Instead, structural information is used to navigate through the design.

Consider the VHDL code shown in Figure 4-3 and how it is mapped into a FPDG representation. The result is depicted in Figure 4-4. Two FPDGs are created, one for the conditional signal assignment and one for the *do_it* process. Some of the sub-graphs but not all are also depicted in the figure. Data dependencies exist between the statement nodes *S5* and *S4*, and between *S6* and *S4*. In Table 4-5 Boolean products representing values of each predicate are reported together with extracted Boolean expression for each region. A Boolean variable is also used for the *clock* signal such that rising edge or falling edge conditions are distinguished.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY example1 IS
PORT ( clock   : IN std_logic;
     reset : IN std_logic;
     ctrl : IN std_logic;
     out_en  : IN std_logic;
     data_in : IN std_logic_vector(7 downto 0);
     data_out : OUT std_logic_vector(7 downto 0));
END example1;

ARCHITECTURE twoProc OF example1 IS
SIGNAL value : std_logic_vector (7 downto 0);
BEGIN
  data_out <= value WHEN out_en = '1'
         ELSE "ZZZZZZZZ";

do_it: PROCESS (clock, reset)
VARIABLE v1 : std_logic_vector(3 downto 0);
  BEGIN
    IF reset = '0' THEN
       value <= "00000000";
    ELSIF clock'EVENT AND clock = '1' THEN
       IF ctrl = '1' THEN
          v1 := data_in(3 downto 0);
       ELSE
          v1 := data_in(7 downto 4) OR "0110";
       END IF;
       value <= v1 & data_in(3 downto 0);
    END IF;
  END PROCESS;

END twoProc;
```

FIGURE 4-3. VHDL code: example1

Predicate and Boolean products	Region expression
$\bar{x}_0$: out_en='0'	R1 : x_0
x_0 : out_en='1'	R2 : $\bar{x}_0$
$\bar{x}_1$: reset = '0	R4 : $\bar{x}_1$
x_1 : reset = '1'	R5 : x_1
$\bar{x}_2$: clock = '0' (falling edge)	R6 : $x_1\bar{x}_2$
x_2 : clock = '1' (rising edge)	R7 : x_1x_2
$\bar{x}_3$: ctrl = '0'	R8 : $x_1x_2x_3$
x_3 : ctrl = '1	R9 : $x_1x_2\bar{x}_3$

Table 4-5. Predicate values and region expressions

4.3.2 Synthesis constraints and recognition of clock signals

In this dissertation, VHDL descriptions which are synthesizable and describe a design at the RTL level are considered. A restricted set of VHDL constructors is used in these descriptions. Particularly, the considered VHDL descriptions satisfy the syntax and semantic rules described in the IEEE P1076.6 standard document [IEE99].

The following is a list of information which has been extracted from the standard document and which is relevant for the analysis here. The list is divided into categories.

1. *Data types*. Primitive as well as user defined types are supported but with the following exceptions:

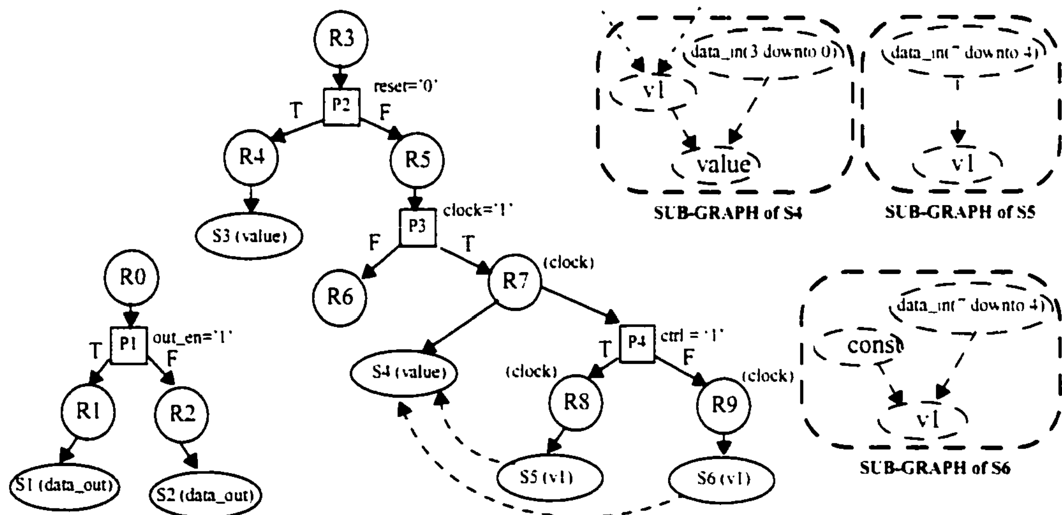


FIGURE 4-4. FPDG: example1

- File, real and physical types are not supported or ignored since they cannot be represented directly as hardware elements.
- Scalar and enumeration types are supported since they can always be represented by vector of bits.
- Resolution functions which handle specific data types are not supported, except for the standard resolution function associated to the STD_LOGIC standard type. Moreover, beside binary values '0' and '1', only the 'Z' value representing high impedance is accepted.
- Array types are supported but only with single index.

2. *Data: signals and variables.* Some limits on the use of data exist.

- Shared variables are not supported since they introduce indeterminism in the simulation and communication channels between components with not clearly defined protocols.
- Data may be indexed by a single range. It means that a data may be referred in an expression as whole data or as a continuous subset of elements (generally bits) but not as a set of non-continuous elements.
- Only attributes which refer to static property of a data are allowed. The only exceptions are EVENT and STABLE attributes, but they must be used only for clock expressions.

3. *Sequential and concurrent statements.* Some statements are not supported and others are supported with specific limitations.

- Assertion and report constructors are not supported since they do not represent any hardware element but they are used for debugging and simulation purpose.
- As described in the standard document, "while" loops are not supported, but "for" loops are. The discrete range of a "for" loop must be statically defined and it must be of integer type. Because of this condition FPDGs do not contain any loop element since unrolling [Muc97 Chapter 17] of loops may always be executed while building an FPDG.

- Wait statements are supported but only with the "until" condition. Moreover, if it is used in a process, it must occur only once and as the first statement in the process. The last limitation is forced in order to maintain a single synchronization point in a process.
- Delay indications are ignored since they are used for simulation purposes and cannot be guaranteed by hardware synthesis.
- Conditional and selected concurrent assignments can not have their target signals used in their conditional expressions. In fact, if a target signal appears in a conditional expression a combinational loop exists which, often, can not be correctly synthesized.

4. *Other limitations.* There are some limitations in the use of constructors which are meaningful for our analysis.
   - Procedure and function recursion is allowed only if bounded by statically determinable values. It means that unrolling the recursion is always possible.
   - When declared a constant, its value should be assigned, no deferring is allowed.
   - The discrete range for a data slice which represents a subset of bits must be statically defined.
   - The guarded block constructor is not supported in the standard. However, as described above, the FPDG representation may handle it.

More limitations exist in the use of package and configuration constructors. However, these constructors are mainly used to handle the hierarchy of a design and they are not involved in the understanding problem which is tackled in this dissertation.

Besides describing the limitations on usage of VHDL constructors, the standard document includes a clause on writing styles for modeling edge sensitive signals, *i.e.* clock signals. These rules are used in this dissertation to distinguish clock signals from other signals in VHDL code. The actual rules are:

1. Only one clock signal may be specified and used in a process.
2. A clock edge expression is represented by one of the following forms:

- *rising_edge(<signal>)*
- *falling_edge(<signal>)*
- *<signal> 'EVENT* AND *<signal>* = '1' (the order is not important)
- *<signal> 'EVENT* AND *<signal>* = '0' (the order is not important)
- NOT *<signal> 'STABLE* AND *<signal>* = '1' (the order is not important)
- NOT *<signal> 'STABLE* AND *<signal>* = '0' (the order is not important)

3. The above expressions are recognized as clock edge conditions if, and only if, they are used in one of the following templates:

- As the condition of a single *if-then-endif* statement:

```
PROCESS
BEGIN
    IF <clock edge expression> THEN
        ....
    END IF;
END PROCESS;
```

- As the condition of an *elsif-then-endif* statement:

```
PROCESS
BEGIN
    IF .... THEN
        ....
    ELSIF <clock edge expression> THEN
        ...
    END IF;
END PROCESS;
```

- As the condition of a *wait until* statement which is the first statement in a process. In this case, also a simpler condition *<signal>* = '1' , or *<signal>* ='0' may be used as the condition for the *wait until* statement in a process.

4.3.3 Verilog HDL and FPDG

A document which specifies rules and constructors to be used for describing digital design at the RTL level using Verilog HDL is currently under development by an Open Verilog Initiative (OVI) working group. A draft version [VSI99] of the proposed standard is available and currently under review. This draft was examined to verify if and how synthesizable Verilog descriptions could actually be represented by FPDGs.

Considering permitted Verilog constructors and comparing them to VHDL constructors:

- The same type of data are available. In fact in Verilog only scalar types are allowed, no enumeration types exist. Real and time values are not supported or are ignored. Moreover the value for high impedance is the same as in VHDL, that is *Z*.
- The available sequential statements in Verilog are equivalent to the VHDL ones, therefore statement nodes and predicate nodes may represent them correctly. In Verilog more loop statements exist: *while*, *forever* and *repeat*, but they are not supported for synthesis. The only supported loop statement is the *for* loop as in VHDL. Even in Verilog the range of the loop must be statically defined to be accepted for synthesis.
- In Verilog *task* and *always* statements are equivalent to *procedure* and *process* constructors in VHDL.
- In Verilog, signals are classified as *wire* and *reg*. They are used in different instances. *Wire* signals may be used only for continuous concurrent assignments and are equivalent to signal in VHDL. *Reg* signals are used inside an *always* statement and correspond to signals or variables in VHDL depending on the type of assignment. In fact Verilog allows two different type of sequential assignments, *non-blocking* assignments (<=) and *blocking* assignments (=). The former has the same semantic meaning as the assignment to signal in VHDL, the latter has the same semantic meaning as the assignment to variable in VHDL. Since RTL rules impose that *reg* signals must appear in an always statement only with one or the other type of assignment, a parser is able to identify the two different usages. Therefore the general distinction, used in the FPDG representation, between a variable and a signal is also valid for Verilog.
- The RTL document for Verilog define rules to identify clock signals, therefore the clock attribute associated to region nodes in a FPDG can still be used. The parser is responsable for identifying those signals and for appropriately setting the attributes in region nodes.

- In Verilog, specific operators exist which manipulate bits of a vector of bits. Such operators are not available in VHDL therefore I did not consider them. However, these operators do not change the structure of an FPDG. They must be considered in the database which maintains the actual expression assigned to a target signal and in the algorithm which extracts the region conditions.
- In Verilog it is possible to directly instantiate logic gates (AND, NAND, OR, NOR, NOT, XOR, XNOR) of any fanin and special electric components like buffer, power supply and ground. This type of description was not intended to be represented by the FPDG representation. In fact, directly instantiating logic gates corresponds to a gate level description and not a register transfer level description which is considered in this dissertation.

Considering the above notes on Verilog language and its rules for RTL design, it is clear that FPDG representation can accommodate Verilog synthesizable descriptions as well as it accommodates VHDL synthesizable descriptions. As an example, in Figure 4-5 a Verilog code which describes the same circuit as the VHDL code in Figure 4-3 is depicted. A

```
module example1 (clock,reset,ctrl,out_en,data_in,data_out)
    input clock;
    input reset;
    input ctrl;
    input out_en;
    input [7:0] data_in;
    output [7:0] data_out;
    reg [7:0] value;
    reg[3:0] v1;

    assign data_out = (out_en == 1'b1) ? value : 8'bzzzzzzzz;

    always @(negedge reset or posedge clock)
    begin
        if (! reset)
            value <= 7'b0000000;
        else
            if (ctrl == 1'b1)
                v1 = data_in[3:0];
            else
                v1 = data_in[7:4] | 4'b0110;
            value <= { v1 , data_in[3:0]};
    end
endmodule
```

FIGURE 4-5. Verilog code: example1

manual mapping of the Verilog description to FPDG representations results in the same set of FPDGs as the ones shown in Figure 4-4. This example illustrates that a Verilog representation and a VHDL representation may be accommodated by a same set of FPDGs.

4.4 Concluding remarks

In this chapter, the Finer Program Dependence Graph (FPDG) which is used to represent the algorithmic behavior of an HDL description has been described. This representation is language independent, that is, designs described using synthesizable VHDL and Verilog HDL description may be mapped to instances of FPDG. Moreover, in Section 4.2, a technique to represent conditional expressions using Boolean logic function was described. Using such a technique, instead of using a single Boolean variable for each bit of a signal, a minimum set of Boolean variables are used to represent meaningful non-overlapping interval values of a signal.

Chapter 5 From FPDG to signal concepts

Signal concepts represent a first level of abstraction of an HDL specification. They form the basic elements for further analyses which aim to extract the behavior which is embedded in a design. In this chapter, different types of signal concepts are first defined, and then an algorithm to extract these concepts from a FPDG is described. Before extracting signal concepts from a FPDG, it is necessary to preprocess the original FPDG to eliminate some uses of variables. It would be useful to review the definition of *predicate* and its values before reading this chapter (see Section 4.2).

As illustrated in the previous chapter, all behavioral information in a VHDL description may be represented by a set of FPDGs, therefore the FPDG may be used as the data structure for performing code analyses. A goal of this dissertation is to represent a design in more abstract terms than VHDL statements. As a first step to accomplish this goal, a procedure, called *signal analysis*, which extracts, from a FPDG, a set of domain specific abstract concepts, called *signal concepts*, is developed.

5.1 Signal concepts

Signal concepts are abstractions which intend to represent patterns of signal use which are commonly found in digital designs. These patterns consist of relations among signals. Different types of signal concepts are identified; each type represents a specific relation among signals.

A signal concept consists of a 5-tuple $sc = \langle tg, ctrl, cond, E, map \rangle$ with *target* signal tg, a *controller* $ctrl$, an *activation condition* $cond$, a set E of expressions including the null expression ϕ, and the function *map*. The target signal may be either a complete signal or a range of bits of a signal, including a single bit of a signal. The controller is a predicate. The activation condition consists of a Boolean expression which must be satisfied for the signal concept to be valid. It is obtained considering values of predicates, excluding the controller, and clock signals. The activation condition may be always true, which means that the signal concept is always valid. The set $E=\{e_1,e_2,\ldots,e_n,\phi\}$ consists of expressions which are assigned to tg. The function $map{:}\,Values_{ctrl} \rightarrow E$ is a one to one mapping from $Values_{ctrl}$ to E, where $Values_{ctrl}$ is the ordered set of Boolean values representing all possible values of the predicate $ctrl$. It means that values of $ctrl$ determine which expression e_i is assigned to tg, eventually the null (ϕ) expression.

By examining various synthesizable VHDL descriptions for different applications, the following set of useful types of signal concepts are defined. Each type is named after the activity which is performed by the controller:

1. *Enabler*: For a specific value of the controller, an expression is assigned to the target signal. For all other values of the controller there is no assignment to the target signal. Formally, in an enabler $sc = \langle tg, ctrl, cond, E, map \rangle$, with $Values_{ctrl}=\{u_1,u_2,\ldots,u_n\}$, there exist i,j with $i \le j$ such that, for each k with $i \le k \le j$ $map(u_k) = e \in E$. For $h < i$ or $h > j$, $map(u_h)=\phi$.
2. *Set/reset*: For a specific value of the controller, the target signal is assigned to a constant value *val*. For all other values of the controller, if an expression is assigned to the target signal, the value of the expression is not a constant. Formally, in a set/reset $sc = \langle tg, ctrl, cond, E, map \rangle$, with $Values_{ctrl}=\{u_1,u_2,\ldots,u_n\}$, there exist i,j with $i \le j$ such that, for each k with $i \le k \le j$ $map(u_k) = e \in E$ and e is a constant. For $h < i$ or $h > j$, $map(u_h) = e_i \in E$ and e_i is not a constant or is ϕ.

3. *Tristate enabler*. For a specific value of the controller, the target signal is assigned to an expression different from high impedance. For all other values of the controller, high impedance is assigned to the target signal. Formally, in a tristate enabler $sc = \langle tg, ctrl, cond, E, map \rangle$, with $Values_{ctrl} = \{u_1, u_2, \ldots, u_n\}$, there exist i,j with $i \leq j$ such that, for each k with $i \leq k \leq j$ $map(u_k) = e_1 \in E$. For $h < i$ or $h > j$, $map(u_h) = e_2 \in E$ with e_2 equal to high impedance.

4. *Tristate disabler*. For a specific value of the controller, the target signal is assigned to high impedance. For all other values of the controller, if an expression is assigned to the target signal, the value of the expression is not high impedance. Formally, in a tristate disabler $sc = \langle tg, ctrl, cond, E, map \rangle$, with $Values_{ctrl} = \{u_1, u_2, \ldots, u_n\}$, there exist i,j with $i \leq j$ such that, for each k with $i \leq k \leq j$ $map(u_k) = e \in E$ and e is equal to high impedance. For $h < i$ or $h > j$, $map(u_h) = e_i \in E$ and e_i is not equal to high impedance or is ϕ.

5. *Input chooser*: For some but not all values of the controller, the target signal is assigned to an expression different from high impedance. For all other values of the controller there is no assignment to the target signal. Formally, in an input chooser $sc = \langle tg, ctrl, cond, E, map \rangle$, with $Values_{ctrl} = \{u_1, u_2, \ldots, u_n\}$, there exists at least an i such that $map(u_i) = \phi$. For all $k \neq i$, $map(u_k) = e_i \in E$ and e_i is not equal to high impedance or is ϕ.

6. *Single assignment*: Independent of the value of the controller, the target signal is assigned the same expression. Formally, in a single assignment $sc = \langle tg, ctrl, cond, E, map \rangle$ for every $u \in Values_{ctrl}$, $map(u) = e \in E$.

7. *Selector*: An expression is assigned to the target signal, for all possible values of the controller. Formally in a selector, for every $u \in Values_{ctrl}$, $map(u) = e_i \in E$ and e_i is not equal to ϕ.

8. *Partial selector*: An expression is assigned to the target signal, for some but not all values of the controller. Formally, in a partial selector $sc = \langle tg, ctrl, cond, E, map \rangle$, with $Values_{ctrl} = \{u_1, u_2, \ldots, u_n\}$, there exists at least a i such that $map(u_i) = \phi$. For all $k \neq i$, $map(u_l) = e_i \in E$.

The order in which these signal concept types have been defined is important, since the algorithm which performs signal analysis uses this order.

There are cases in which an expression is assigned to a target signal independent of any predicate. For those cases, a new signal concept has been defined:

- *Logic net*. It consists of a *target* signal, an *expression* and an *activation condition*. The expression is assigned to the target independent of any predicate. Generally, the activation condition is always true, that is the assignment is always valid. In some cases the condition represents a clock edge which indicates that the signal concept depends on a clock signal. This signal concept is named after the behavior of the target signal.

To have a properly working digital circuit, each of its wires must be driven by at most one value different from high impedance at each given time. In term of a VHDL specification, it means that at most one assignment to a target signal whose value is different from high impedance and which is independent from any predicate may exist. Having at most one such assignment for a target *tg* means that if a *logic net* concept is identified for a target *tg*, then no other signal concept exists with *tg* as the target signal.

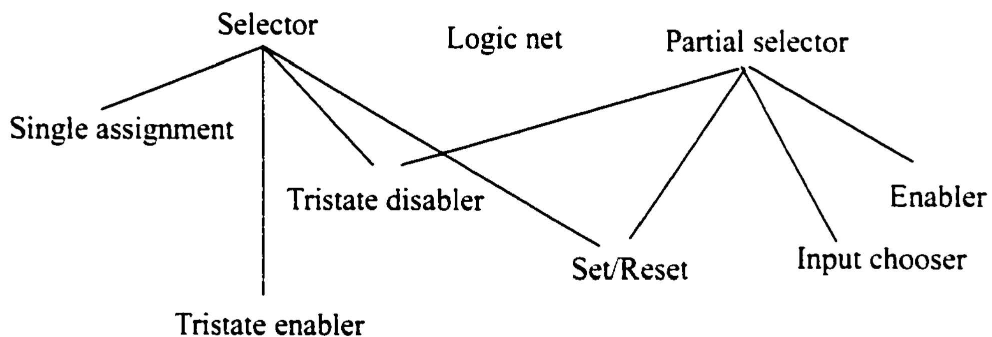


FIGURE 5-1. Taxonomy of signal concepts

Figure 5-1depicts the relations among the types of signal concepts. Single assignment and tristate enabler types are specializations of the selector type. Input chooser and enabler are specializations of the partial selector type. The set/reset and tristate disabler types may be specializations of either a selector or a partial selector type.

5.1.1 Some examples

Before describing the algorithm which is used to perform signal analysis, some examples showing the results of signal analysis may clarify the scope of the algorithm.

Consider the VHDL code and the derived FPDG of the example *sig1* in Figure 5-2. Two predicates *A*, *B* and three target signals *Z1*, *Z2*, *Z3* exist. Expressions are assigned separately to the whole *Z1* data and to its second bit, therefore two different target *Z1* and *Z1*(*2*) must be considered when extracting signal concepts. In Table 5-1, all signal concepts identified by signal analysis are reported. The assignments column report the statement nodes which contain assignments which have been used to identify the signal concept. For example, considering the target *Z2* and the controller *B*, a *tristate disabler* signal concept is identified under condition *A*='1'. In fact, under the condition that *A*='1', when *B*='1' high impedance is assigned to *Z2*; when *B*='0' no expression is assigned to *Z2*. It means that *B* behaves as the controller in the definition of a tristate disabler signal

```
ARCHITECTURE beh OF sig1 IS
BEGIN
proc: PROCESS(A,B,C,D)
BEGIN
    IF (B ='1') THEN
        IF (A ='0') THEN
            Z1(2) <= '1';
            Z2 <= C
            Z3 <= '1';
        ELSE
            Z1 <= "0100";
            Z2 <= 'Z';
            Z3 <= C AND D;
        END IF;
    END IF;
END PROCESS;
END beh;
```

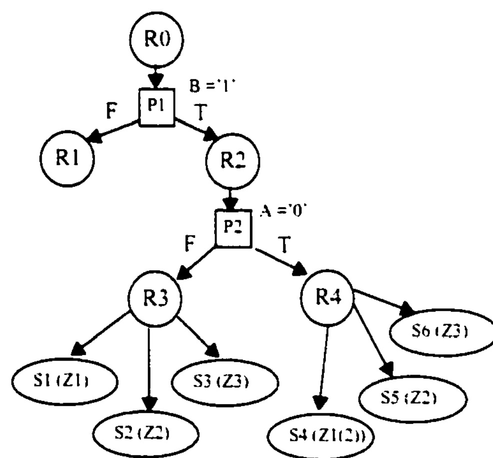


FIGURE 5-2. Example sig1: VHDL code and FPDG

Signal concept	Target	Controller	Activatiion Condition	Assignments
Enabler	Z1	B	A='1'	S1
Enabler	Z1	A	B='1'	S1
Enabler	Z1(2)	B	A='0'	S4
Single assignment	Z1(2)	A	B='1'	S1, S4
Enabler	Z2	B	A='0'	S5
Tristate disabler	Z2	B	A='1'	S2
Tristate enabler	Z2	A	B='1'	S2, S5
Enabler	Z3	B	A='0'	S6
Enabler	Z3	B	A='1'	S3
Set/reset	Z3	A	B='1'	S3, S6

Table 5-1. Example sig1: signal concepts

concept, and therefore this type of signal concept may be inferred with condition *A*='1' and target *Z2*. However, under a different condition, *A*='0', an *enabler* signal concept is identified with the same target *Z2* and controller *B*. In this case the expression *C* is assigned to *Z2* when *B*='1', and no assignment exists for *B*='0'. In this example, the *single assignment* signal concept is determined for the target *Z1(2)* and controller *A*, under condition *B*='1', because both assignments *S1* and *S4* assign value '1' to *Z(2)*.

Another example is depicted in Figure 5-3, with extracted signal concepts reported in Table 5-2. In this case there is only one predicate *A* which is a multi-bits signal. Its values,

```
ARCHITECTURE beh OF ex_sig2 IS
BEGIN
proc: PROCESS
BEGIN
    IF (A = "0010") THEN
        Z1 <= "11";
        Z2 <= B;
        Z3 <= C;
    ELSE
        Z2 <= C;
        IF (A > "0101") THEN
            Z1 <= B;
        ELSIF (A = "0000") THEN
            Z1 <= "00";
        END IF;
    END IF;
END PROCESS;
END beh;
```

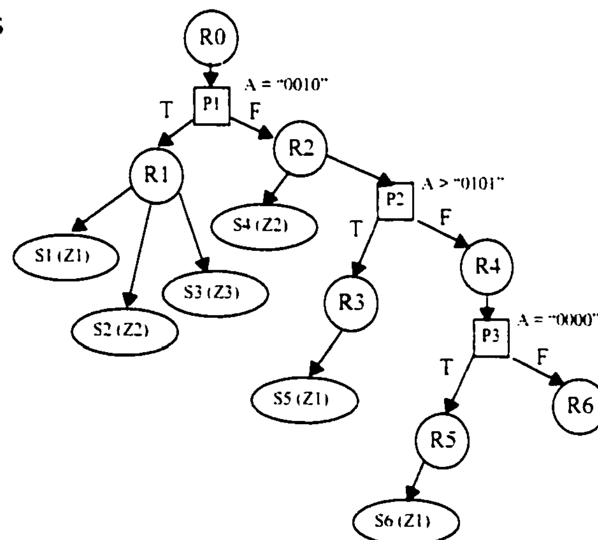


FIGURE 5-3. Example sig2: VHDL code and FPDG

Signal concept	Target	Controller	Activation Condition	Assignments
Input chooser	Z1	A	Always	S1, S5, S6
Selector	Z2	A	Always	S2, S4
Enabler	Z3	A	Always	S3

Table 5-2. Example sig2: signal concepts

Interval	Boolean product
["0000"]	$\bar{x}_0\bar{x}_1\bar{x}_2$
["0001"]	$\bar{x}_0\bar{x}_1x_2$
["0010"]	$\bar{x}_0x_1\bar{x}_2$
["0011" to "0101"]	$\bar{x}_0x_1x_2$
["0110" to "1111"]	$x_0\bar{x}_1\bar{x}_2$

Table 5-3. Values of predicate A

reported in Table 5-3, are Boolean products representing non-overlapping intervals which are used to specify conditional expressions and, consequently, region conditions. For example the region condition $\bar{x}_0\bar{x}_1\bar{x}_2+\bar{x}_0x_1x_2$ for the *R6* region node represents values ["0001"] and ["0011" to "0101"] for the *A* predicate. Considering the target *Z1*, an *input choice* signal concept is identified. In fact, different expressions are assigned to *Z1* depending on the values of *A*, but, no expressions are assigned to *Z1* for the values of the predicate *A* which are identified by the condition in region *R6*. For all extracted signal concepts, the *condition* is always true because they do not depend on specific values of other predicate.

5.2 Extracting signal concepts: signal analysis

Given an architecture, signal analysis is performed on each FPDG created from the VHDL description of the architecture. Conceptually, signal analysis considers all predicates which are used in a FPDG to represent conditional expressions of predicate nodes and all existing target signals with its sub-ranges. Given a controller (a predicate) and a target signal, signal analysis collects all assignments which are executed under the same condition but depends on values of the controller, and uses them to identify an appropriate signal concept type. Before proceeding to extract signal concepts, if it is necessary, signal analysis simplifies and modifies each FPDG by means of a preprocessing activity.

5.2.1 Preprocessing

Unrolling of loops and unfolding of procedure calls are operations which are performed during the construction of a FPDG. Other operations are executed on a given FPDG before extracting signal concepts.

In a VHDL process, variables, unlike signals, assume their values as soon as their assignment statements occur. Therefore, sometimes, when a variable is used, its value may be statically determined. Using techniques commonly employed in compilers, and particularly the idea of $\phi - function$ from the Static Single-Assignment SSA form [Muc97], preprocessing substitutes the use of a variable with its appropriate conditioned or unconditioned expression assigned to the variable. In a FPDG, there are two instances in which a variable may be used: in an assignment expression or in a conditional expression.

1. *At the right hand side (RHS) of assignments.* The value of a variable which is used in a RHS of a signal assignment may depend on previous lines of code in the process. There might be more than one assignment which defines a value for the variable, depending on the control flow in the process. For example, in the partial VHDL code and FPDG shown in Figure 5-4, the variable *V* is used in the signal assignment *Z1 <= V OR D*. There are two different possible values which variable *V* may assumed when used: value "01" under the condition *A*='1' and value *B AND C* under the condition *A*='0'. In this case, it is possible to remove the assignments to variable *V* and its use in the assignment *Z1 <= V OR D* by substituting the possible values of *V* in it. Particularly, the original assignment to *Z1* may be substituted by two new assignments *Z1 <= "01" OR D* and *Z1 <= (B AND C) OR D* with respectively attached condition *A*='1'

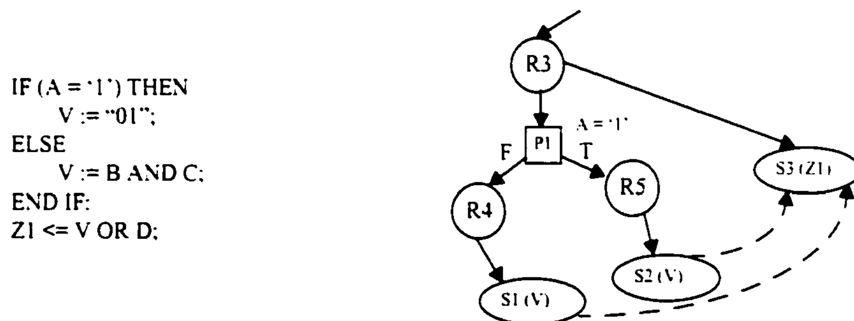


FIGURE 5-4. Example: Assignment with variable

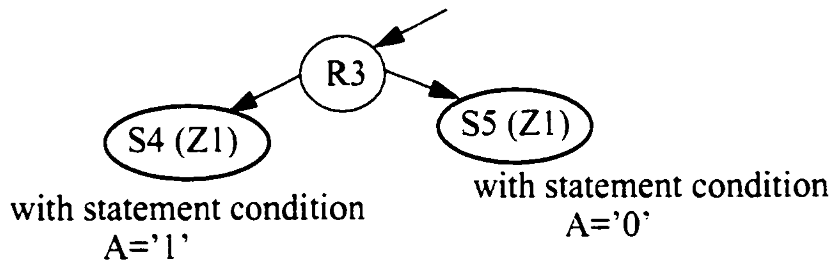


FIGURE 5-5. Example: Assignment with variable

and *A*='0'. During preprocessing, for each use of a variable in a signal assignment, data dependence edges are considered and recursively followed in order to determine all possible values for the variable. The statement node which contained the signal assignment is replaced by as many statement nodes as many possible values have been found for the variable. Each of these new statement node contains the original signal assignment with the variable replaced by one of its value. Moreover, these statement nodes differ from the previously defined statement nodes of a FPDG because they have a condition associated with them, called the *statement condition*. This condition, in the form of a Boolean expression composed of Boolean variables which are associated to predicates, represents a further condition which must be satisfied for the assignment to be valid. The statement condition is obtained while following data dependence edges when calculating all possible values for a variable. Considering the example above, after preprocessing a new FPDG is created as shown in Figure 5-5. Two new statement nodes *S4*, *S5*, with respectively assignment *Z1 <= "01" OR D* or *Z1 <= (B AND C) OR D*, replace the old *S3* node. The statement nodes *S1*,*S2* nodes, which contain the propagated assignments to *V*, are removed.

2. *In the expression of a conditional statement.* Again data dependence edges are considered and recursively followed in order to determine all possible values and associated conditions for a variable in the expression. The original expression in the conditional statement is replaced by a new expression which take into account all possible values and associated conditions for the variable. The new expression consists of a logic OR expression in which each term is obtained by substituting a value of the variable into the original expression and ANDing the result with the condition attached to the value of the variable. Let's consider the example in Figure 5-6. The expression in the predi-

```
IF (A = '1') THEN
    V := "01";
ELSE
    V := B+C;
END IF;
IF (V="01") THEN
    Z1 <= C OR D;
END IF;
```

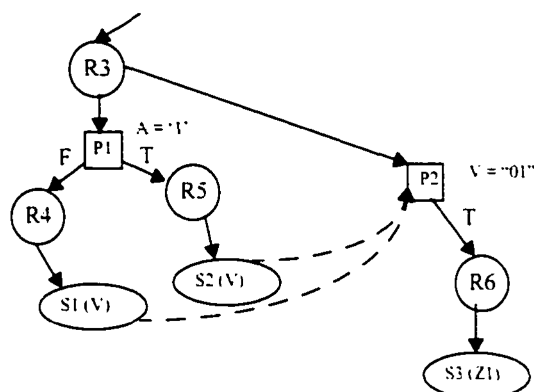


FIGURE 5-6. Example: Conditional statement with a variable

cate node *P2* contains the variable *V*. Following the data dependence edges, two possible values are identified: value "01" under the condition *A*='1' and value *B+C* under the condition *A*='0'. Considering these values, the condition in *P2* is replaced by condition: ("01" = "01" AND *A='1'*) OR (*B+C*="01" AND *A='0'*), which using *constant-expression evaluation* technique [Muc97 Chapter 12], is simplified to (*A*='1') OR (*B+C*="01" AND *A*='0').The statement nodes *S1*,*S2* nodes, which contain the propagated assignments to *V*, are removed

There might be cases in which there is an execution path from the root of a FPDG to a use of a variable in which no assignments to the variable exist. In these cases, the assignment which uses the variable or the conditional expression which contains the variable is kept as it is in the FPDG. This variable is treated as a signal in all analyses.

The algorithm which is employed by signal analysis and which extracts signal concepts by traversing a FPDG requires that: each assignment, in a statement node, has a direct effect on the behavior, represented by the FPDG, whenever its associated execution condition is valid. This requisite is not satisfied by an FPDG when a statement node contains an assignment to a signal which is overwritten by an assignment to the same signal. That is, two assignments to the same signal exist which are valid for two region condition $cond_1$ and $cond_2$ which are not mutually exclusive. Considering an FPDG, the above condition occurs only when a statement node containing an assignment to a signal *s* is a child of a region *R1* which belongs to the subtree of a region *R2* which has a statement node with an assignment to the same signal *s* as a child. Let's consider for example the portion of

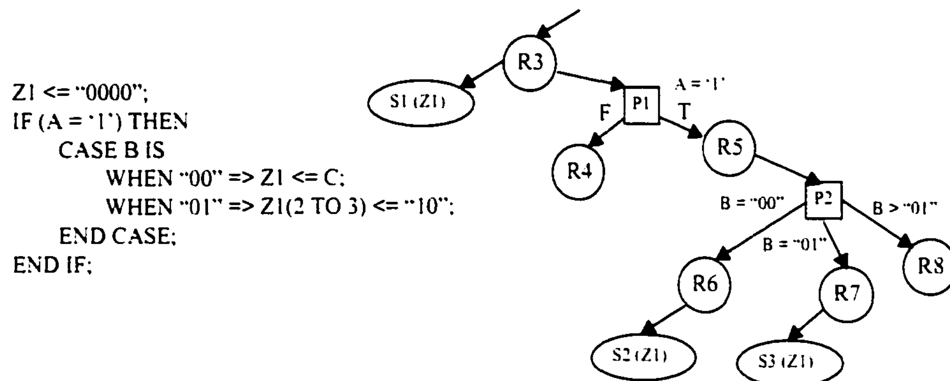


FIGURE 5-7. Example: Overwritten assignment

VHDL code and its associated partial FPDG in Figure 5-7. The assignment *Z1 <= "0000"* of the *S1* statement node is not always valid when the region condition of *R3* is true, but it is overwritten by later assignments. In fact in the case that *A*='1' and *B* = "00", the value *C* is assigned to *Z1*. Similarly when *A* ='1' and *B* ="01", values '1' and '0' are assigned respectively to bit two and bit three of *Z1*.

In dealing with this kind of situation, the preprocessing activity modifies an FPDG by removing from the region the statement node containing the overwritten assignment to a signal *s*. Subtrees of predicate nodes connected to the region where an overwriting statement exists are traversed. Copies of the removed statement node are created and connected to regions of the subtree were no assignment to the signal *s* exist. When a statement node which contains an assignment to the signal *s* is found in the subtree, it is kept or eventually modified to include assignment to those bits which are assigned in the removed assignment but are not assigned by it. Figure 5-8 shows the new FPDG of the above example which is obtained after this preprocessing activity. The original *S1* statement node has been removed and replaced by two statement nodes *S4* and *S5* which are copies of the original *S1* statement node. Moreover, the original statement S3: *Z1(2 TO3) <= "10"* has been modified into *Z1<= "0010"* by considering the statement *S1* for bits zero and one of *Z1*. This preprocessing operation guarantees that the final FPDG contains statement nodes whose assignments are never overwritten.

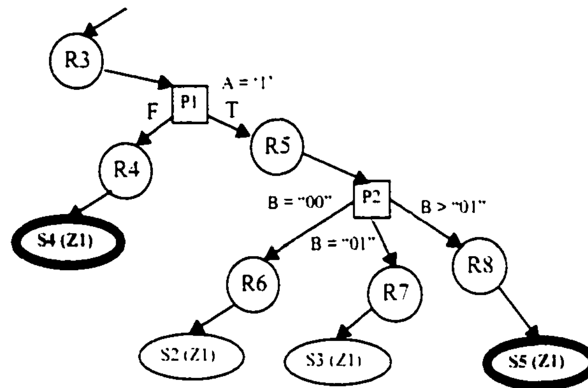


FIGURE 5-8. Example: Overwritten assignment

5.2.2 Signal analysis: algorithm

Signal analysis is carried out by an algorithm which performs a *postorder* traversal of the tree structure of a FPDG which is made up of control dependence edges. At each region and predicate node, the algorithm creates and collects *conditional assignments* which are then used to determine signal concepts.

The following definitions are used to describe the algorithm:

- *Target.* It is a signal to which at least one value is assigned in a VHDL description.
- *Predicate.* It is an element that comes from the decomposition of a conditional expression. Its possible values are subdivided in non-overlapping intervals which are represented by products of Boolean variables, see Section 4.2 on page 36.
- $Values_{pd}$ is the set of Boolean products representing values of a predicate *pd*.
- *Asssign(tg)*. It is an assignment statement with target tg.
- A *conditional assignment* is a 4-tuple: $ca = \langle tg, pd, co, AL \rangle$, where *tg* is a target, *pd* is a predicate, *co* is a condition represented by a Boolean expression in terms of predicate values and *AL* is a list of assignments. Each assignment $assign(tg) \in AL$ is associated to an interval $v \in Values_{pd}$, that is *AL* is isomorphic to a subset $V \subseteq Values(pd)$ and

$|AL| \leq |Values_{pd}|$. A conditional assignment represents a collection of assignments to a specific target which are activated for different values of a predicate. A global condition exists which identifies when the conditional assignment itself is valid.

- *Stmcond*, statement condition, is a Boolean expression of predicate values associated to a statement, see Section 5.2.1.
- *Regcond*(R_i), region condition, is a Boolean expression in term of predicate values associated to the region node R_i.
- *RegionCA*(R_i) is the set of conditional assignments which are collected in the region node R_i.
- *predicateCA*(P_i) is the set of conditional assignments which are collected in the predicate node P_i.

In Figure 5-9, the pseudo-code of the algorithm which performs signal analysis is depicted. The last step of the algorithm consists of mapping each conditional assignment to a signal concept. The mapping is performed by considering each signal concept type in the order given in Section 5.1. That is, as soon as a signal concept definition is matched against a conditional assignment, the algorithm does not perform any further search. A conditional assignment $ca = \langle tg, pd, co, AL \rangle$ is mapped to a signal assignment $sc = \langle tg, ctrl, cond, E, map \rangle$ in the following way:

- The target *tg* of the conditional assignment becomes the target *tg* of the signal assignment.
- The predicate *pd* becomes the controller *ctrl*.
- The condition *co* becomes the activation condition *cond*.
- Considering each value $v \in Values_{pd}$ of *pd*. If there exists an assignment $a \in AL$ for the given value *v*, the expression used in the assignment is inserted in the set *E*. If no assignment is associated to the value *v*, the null expression ϕ is included in *E*. At the same time, the mapping function *map* is built.

```
Given a preprocessed FPDG and following a postorder traversal
  at each region node R_i with regcond(R_i)
    create an empty set regionCA(R_i)
    visit each child following the order defined by control flow edges
      if the child is a statement node with assign(tg) and asscond then
        each ca ∈ regionCA(R_i) with target tg is removed
        for each predicate pd on which regcond(R_i) depends
          new conditional assignments as nca = ⟨tg,pd,f,AL⟩
           are created and inserted in set regionCA(R_i). AL contains
           one or more assign(tg) and condition f is calculated
           considering regcond(R_i), asscond and Values_pd.
        if no predicate pd exists which depend on regcond(R_i) the
          assign(tg) is used to form a logic net signal concept
      if the child is a predicate node P_i
        each pca = ⟨tg,pd,pco,pAL⟩ ∈ predicateCA(P_i) is
         compared and eventually merged with existing conditional
          assignments in regionCA(R_i) with same target
  at each predicate node P_i
    initialize predicateCA(P_i) with the regionCA of one of the region child
    visit each remaining region child R_i with no specific order
      each rca = ⟨tg,pd,rco,rAL⟩ ∈ regionCA(R_i) is compared and
       eventually merge with existing conditional assignments in
       predicateCA(P_i) with same target
  after traversal, each collected condition assignment is mapped to a
   signal concept
```

FIGURE 5-9. Pseudo-code: signal analysis

Note that, at least, a conditional assignment is mapped to a *selector* or *partial selector* signal concept type.

Merging and updating operations in the above algorithm takes into account regional conditions, assignment conditions and conditions of collected conditional assignments. A detailed description of the algorithm employed by signal analysis may be found in Appendix B.1.

As an example, let's consider the VHDL code and the FPDG in Figure 5-10. For predicate A the non-overlapping interval values are '0' and '1' and are represented respectively by

$\bar{x}_0$ and x_0 Boolean terms. For predicate *B*, its useful values are "00", "01" and ["10","11"] which are represented respectively by Boolean products $\bar{x}_1.\bar{x}_2$, $\bar{x}_1.x_2$ and $x_1.\bar{x}_2$. Therefore conditions at various region nodes are: *regcond(R0)*=1, *regcond*(*R1*)=$\bar{x}_0$, *regcond*(*R2*)=$\bar{x}_0x_1+\bar{x}_0x_2$, *regcond*(*R3*)=$\bar{x}_0\bar{x}_1.\bar{x}_2$ and *regcond*(*R4*)=$\bar{x}_1.x_2$. In Figure 5-10, numbers in bold near each node indicate the visiting order number during postorder traversal. The actions taken at each region and predicate node are as follow:

- In region *R2*, two different conditional assignments are created from the assignment *Z2<=C* for predicate *A* and *B*.
  $regionCA(R2)=\{<Z2,A,x_1+x_2,[Z2<=C]\}>,<Z2,B,\bar{x}_0,[Z2<=C,Z2<=C]>\}$

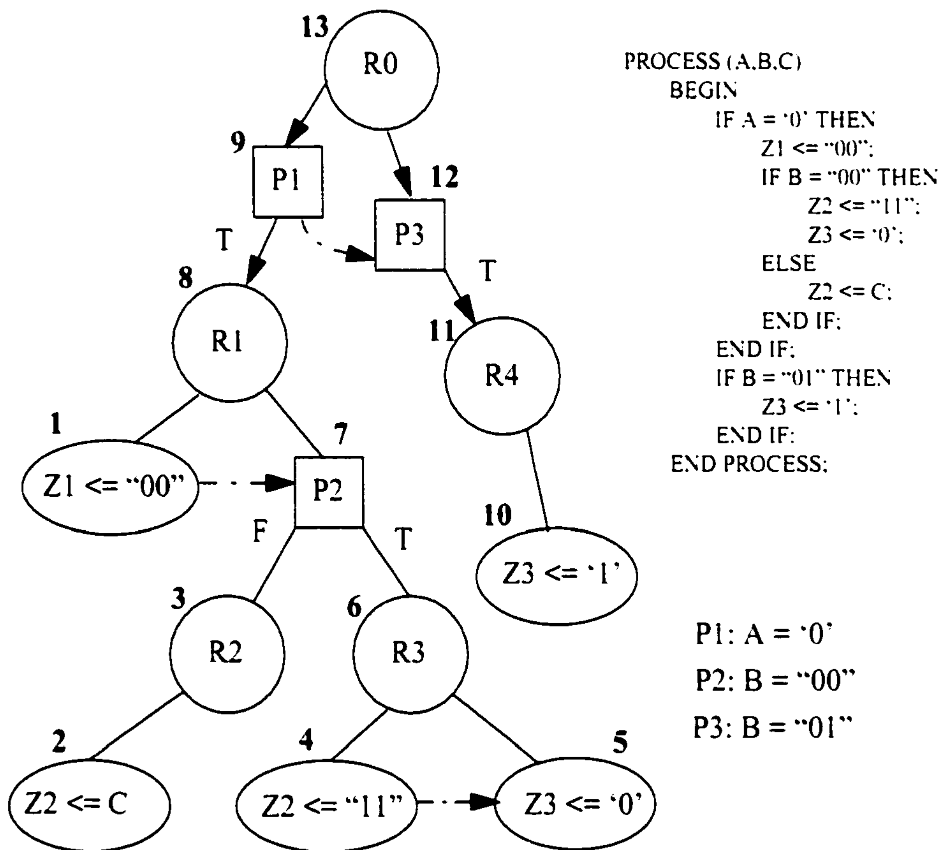


FIGURE 5-10. FPDG+VHDL: signal analysis

- In region *R3*, following the control flow edge, assignment $Z2 <= \text{"11"}$ is considered first and then assignment $Z3 <= \text{'0'}$. Four different conditional assignments are created.

  $regionCA(R3)=\{<Z2,A,\bar{x}_1\bar{x}_2,[Z2<=\text{"11"}]>,<Z2,B,\bar{x}_0,[Z2<=\text{"11"}]>,$

  $<Z3,A,\bar{x}_1\bar{x}_2,[Z3<=\text{'0'}]>,<Z3,B,\bar{x}_0,[Z3<=\text{'0'}]>\}$

- In predicate *P2*, first:

  $predicateCA(P2)=\{<Z2,A,x_1+x_2,[Z2<=C]>,<Z2,B,\bar{x}_0,[Z2<=C,Z2<=C]>\}$

  that is equal to *regionCA(R2)*, then conditional assignments in *regionCA(R3)* with target *Z2* are merged with conditional assignments in *predicateCA(P2)* with target *Z2*, if compatible. In fact, together, conditional assignments $<Z2,B,\bar{x}_0,[Z2<=\text{"11"}]>$ and $<Z2,B,\bar{x}_0,[Z2<=C,Z2<=C]>$ form $<Z2,B,\bar{x}_0,[Z2<=\text{"11"},Z2<=C,Z2<=C]>$. Conditional assignments in *regionCA(R3)* with target *Z3* are simply added to *predicateCA(P2)*.

  $predicateCA(P2)=\{<Z2,A,x_1+x_2,[Z2<=C]>,<Z2,A,\bar{x}_1\bar{x}_2,[Z2<=\text{"11"}]>,$

  $<Z2,B,\bar{x}_0,[Z2<=\text{"11"},Z2<=C,Z2<=C]>,$

  $<Z3,A,\bar{x}_1\bar{x}_2,[Z3<=\text{'0'}]>,<Z3,B,\bar{x}_0,[Z3<=\text{'0'}]>\}$

- In region *R1*, considering the assignment $Z1<=\text{"00"}$, a conditional assignment is created for predicate *A*. $regionCA(R1)=\{<Z1,A,1,[Z1<=\text{"00"}]>\}$. Then conditional assignments of *predicateCA(P2)* are considered. Since the latter have targets different from *Z1*, they are just added to *regionCA(R1)* set.

  $regionCA(R1)=\{<Z2,A,x_1+x_2,[Z2<=C]>,<Z2,A,\bar{x}_1\bar{x}_2,[Z2<=\text{"11"}]>,$

  $<Z2,B,\bar{x}_0,[Z2<=\text{"11"},Z2<=C,Z2<=C]>,<Z1,A,1,[Z1<=\text{"00"}]>,$

  $<Z3,A,\bar{x}_1\bar{x}_2,[Z3<=\text{'0'}]>,<Z3,B,\bar{x}_0,[Z3<=\text{'0'}]>\}$

- In predicate *P1*, the *predicateCA(P1)* set becomes equal to the *regionCA(R1)* set.

- In region *R4*, considering the assignment $Z3<=\text{'1'}$, a conditional assignment is created for predicate *B*. $regionCA(R4)=\{<Z3,B,1,[Z3<=\text{'1'}]>\}$.

- In predicate *P3*, conditional assignments in *regionCA(R4)* set are added in $predicateCA(P3)=\{<Z3,B,1,[Z3<=\text{'1'}]>\}$.

Signal concept	Target	Predicate	Activation Condition	used expressions
Enabler	Z2	A	B/="00" (x_1+x_2)	C,ϕ
Enabler	Z2	A	B="00" $(\bar{x}_1\bar{x}_2)$	"11",ϕ
Selector	Z2	B	A='0'$(\bar{x}_0)$	"11",C
Enabler	Z1	A	Always	"00",ϕ
Enabler	Z3	A	B="00" $(\bar{x}_1\bar{x}_2)$	'0',ϕ
Input chooser	Z3	B	A='0'$(\bar{x}_0)$	'0','1',ϕ
Enabler	Z3	B	A='1'(x_0)	'1',ϕ

Table 5-4. Results of signal analysis

- In region *R0*, conditional assignments in *predicateCA*(*P1*) are added in *regionCA*(*R0*) set. Then the condition assignment <*Z3,B,1,[Z3<='1']*> in *predicateCA(P3)* is merged with <*Z3,B,*$\bar{x}_0$*,[Z3<='0']*> in *regionCA*(*R0*) to form a new predicate assignment <*Z3,B,*x_0*,[Z3<='0',Z3<='1']*>. Moreover, the same conditional assignment <*Z3,B,1,[Z3<='1']*> is also added into *regionCA*(*R0*) after having updated the condition, that is as <*Z3,B,*x_0*,[Z3<='1']*>. In fact, assignment *Z3<='1'* is valid also when *A*='1' (x_0). As a result:

  regionCA(*R0*)={<*Z2,A,*x_1+x_2*,[Z2<=C]*>,<*Z2,A,*$\bar{x}_1\bar{x}_2$*,[Z2<="11"]*>,
  <*Z2,B,*$\bar{x}_0$*,[Z2<="11",Z2<=C,Z2<=C]*>,<*Z1,A,1,[Z1<="00"]*>,
  <*Z3,A,*$\bar{x}_1\bar{x}_2$*,[Z3<='0']*>,<*Z3,B,*$\bar{x}_0$*,[Z3<='0',Z3<='1']*>,
  <*Z3,B,*x_0*,[Z3<='1']*>}

At the end each conditional assignment is mapped to a signal concept. The result of the mapping is shown in Table 5-4.

5.3 Concluding remarks

Signal analysis extracts all possible signal concepts which are embedded in a FPDG. Particularly, for each assignment in a FPDG, at least one signal concept is extracted for each predicate on which the assignment depends. Each predicate is used as a controller in at least one signal concept which contains the expression of the assignment. The set of signal

concepts extracted from a FPDG, by algorithmic construction, satisfies the following properties:

- *Property 1*: the set of signal concepts which are extracted from a FPDG contains all assignments which effect the behavior described by the FPDG.
- *Property 2*: $SC = \{sc_1, sc_2, \ldots, sc_n\}$ is the set of extracted signal concepts with the same target *tg* and the same controller *ctrl*. Each element $sc_i = \langle tg, ctrl, cond_i, E_i, map_i \rangle$ of the set *SC* differs from each other for the activation condition and the set of expressions. Particularly, $\forall (i, j)$ with $i \neq j$, the Boolean product $cond_i \cdot cond_j = 0$ and $E_i \neq E_j$.
- *Property 3*: Each signal concept $sc_i = \langle tg, ctrl, cond_i, E_i, map_i \rangle$ contains in E_i all possible expressions which are assigned to tg_i in the original FPDG when the Boolean sum $\sum_{u \in U} cond_i \cdot u$ is true. Where $U = \{u \mid map_i(u) = e_k \in E_i \text{ with } e_i \neq \phi\}$

In Appendix B.1, a detailed description of the algorithm, which performs signal analysis, is reported.

Signal concepts are used as basic elements for further analysis. In the next chapter, these concepts are collected together to form higher level abstractions.

Chapter 6 Combinational function concepts

After signal analysis, the behavior of an architecture is described by a set of signal concepts. Looking separately at each signal concept, designers may retrieve only low level and local information regarding a target signal and its relationship with one predicate at a time. This chapter describes a technique which analyzes and combines signal concepts to create new artifacts at a higher level of abstraction. The new artifacts are called combinational function concepts. Algorithms which extract different types of combinational function concepts are described. Before reading this chapter, it would be useful to review the definitions of *signal concept*, *predicate* and *restriction* of a Boolean function (denoted $f|_p$) in Appendix A.

6.1 Merging signal concepts

Theorem 6.1. Signal concepts s_i=<tg_i,$ctrl$,$cond_i$,E_i,map_i> of a set S are active at the same time if, and only if, the Boolean product $b = cond_1 \cdot cond_2 \cdot \ldots \cdot cond_n \neq 0$.

Proof. The proof is straight forward and it is based on the definition of a Boolean product.❑

Signal analysis guarantees that signal concepts with the same target and controller which are extracted from the same FPDG cannot be active at a same time. However, signal anal-

```
Given an architecture and the set S of extracted signal concepts
    initialize S' = ∅
    for each target tg
        for each control ctrl
            consider subset L ⊆ S of signal concepts with the same target tg and
              the same controller ctrl
            initialize L'={}
            for i=1 to |L|
                initialize M'={s(i)}
                for j=1 to |L'|
                    consider:
                    s(i)=<tg,ctrl,cond_i,E_i,map_i> with s(i) ∈ L
                    s(j)=<tg,ctrl,cond_j,E_j,map_j> with s(j) ∈ L'
                    if cond_i · cond_j ≠ 0 then
                        M = merge(s(i),s(j))
                        if M ≠ ∅ then
                            remove s(j) from L'
                            remove s(i) from M'
                        M' = M' ∪ M
                L' = L' ∪ M'
            S' = S' ∪ L'
```

FIGURE 6-1. Pseudo-code: merging algorithm

ysis is executed on each separate FPDG at a time, therefore nothing can be said about signal concepts which are extracted from different FPDGs. When more than one FPDG is created during the parsing of an architecture, *e.g.* in the case that more than one process is used, there might exist signal concepts with the same target and the same controller which are active under the same compatible condition.Therefore, before proceeding with a new analysis, the set S of signal concepts, which is extracted by signal analysis from different FPDGs in the same architecture, is considered.

A merging algorithm, which evaluates all possible merging amongst signal concepts in S with the same target and the same controller, is executed. A new set $S' \subseteq S$ is obtained. The pseudo-code is depicted in Figure 6-1. The algorithm runs in polynomial time. For each $|L|=n$, it is upper bounded by $n \times (n+1)/2$ calls to the procedure merge($s(i)$, $s(j)$).

By construction, each signal concept sc=<$tg,ctrl,cond,E,map$> in the list S' contains in its E all possible expressions assigned to tg in the original architecture when the Boolean sum $\sum_{u \in U} cond \cdot u$ is true, where $U=\{u|\ map(u)=e_k \in E \ with \ e_k \neq \phi\}$. This property is called *completeness* of a signal concept.

6.1.1 Merging two signal concepts

In the algorithm described in Figure 6-1, the merging of two signal concepts is executed by the procedure *merge(s(i),s(j))*. Given two signal concepts $s(i)$=<$tg,ctrl,cond_i,E_i,map_i$> and $s(j)$=<$tg,ctrl,cond_j,E_j,map_j$>, this procedure generates a set M of new signal concepts. Depending on the activation conditions $cond_i$ and $cond_j$, a maximum of three different signal concepts are generated:

1. The signal concept s_1=<$tg,ctrl,cond_1,E_1,map_1$> with $cond_1 = cond_i \cdot cond_j$.

2. The signal concept s_2=<$tg,ctrl,cond_2,E_2,map_2$> with $cond_2 = cond_i \cdot \overline{cond_j}$ is generated if, and only if, $cond_2 \neq 0$.

3. The signal concept s_3=<$tg,ctrl,cond_3,E_3,map_3$> with $cond_3 = \overline{cond_i} \cdot cond_j$ is generated if, and only if, $cond_3 \neq 0$.

If generated, the signal concepts s_2 and s_3 indicate, respectively, that the signal concept $s(i)$ and $s(j)$ are valid concepts under different activation conditions. In fact, for the signal concept s_2, $E_2 = E_i$ and $map_2 = map_i$, while for the signal concept s_3, $E_3 = E_j$ and $map_3 = map_j$. For the signal concept s_1, the set E_1 and the function map_1 are obtained considering each value $v \in Values_{ctrl}$ and E_i,Ej,map_i,map_j. In Table 6-1, the possible values of function $map_1(v)$ for a given $v \in Values_{ctrl}$, based on the values of $map_i(v)$ and $map_j(v)$, are depicted. The symbols e_i and e_j represent, respectively, expressions different from 'Z' and ϕ which correspond to values of $map_i(v)$ and $map_j(v)$. The set E_1 is equal to

		$map_j(v)$		
		e_j	Z	ϕ
	e_i	conflict	e_i	e_i
$map_i(v)$	Z	e_j	Z	Z
	ϕ	e_j	Z	ϕ

Table 6-1. Values of function map for merged signal concepts

$E_i \cup E_j$ minus the expression 'Z' if expression 'Z' does not belong to the image of the function map_l, and minus expression ϕ if the function map_l does not contains ϕ in its image.

From Table 6-1, when $map_i(v)=e_i$ and $map_j(v)=e_j$ there is a conflict and no value may be determined for $map_l(v)$. In this case, no merge is obtained and the signal concept s_l is not generated. Since, in this dissertation, only components (legacy designs) which are synthesizable and which constitute correct digital circuits, are considered, the conflict depicted in Table 6-1 should never occur. In fact, in a synthesizable VHDL description of a correct digital circuit, at any possible combination of input values, if n assignments, with $n>1$, exist for a target signal, at most one assignment can be different from high impedance. This condition derives directly from the property of digital logic circuits that a wire is driven to a logic value zero or one by only one of the gate outputs connected to the wire. When more than one of the n assignments are different from high impedance, a logic synthesis tool advises designers that a bus conflict exists. Similarly, the procedure, which performs the merge of two signal concepts, issues a warning message saying that a conflict occurs on the target *tg* for the controller *ctrl*.

As an example, Figure 6-2 depicts an architecture which contains two processes. Process *p1* is a concurrent conditional assignment and *p2* is an explicit process with sequential code. Two signal concepts:

sc_1=<*T,A,cond*$_1$: *always*,['*1*','*Z*'],{*0*→'*1*',*1*→'*Z*'}> and

sc_2=<*T,A,cond*$_2$: *B*='1',['*Z*','*0*'],{*0*→'*Z*',*1*→'*0*'}>

which have been identified by the signal analysis are also reported in Figure 6-2. Since $cond_1 \cdot cond_2 \neq 0$, the merge procedure generates a set M of merged signal concepts. A

ARCHITECTURE beh OF merge IS
BEGIN
p1: T <= '1' WHEN A='0' ELSE 'Z';
 	
p2: PROCESS (A)
BEGIN
IF (B='1') THEN
IF (A ='1') THEN
T <= '0';
ELSE
T <= 'Z';
END IF;
END IF;
END PROCESS;
END beh;
	sc_1: *Tristate enabler* for *p1*:
	target: T
	activation condition: always
	controller: A
	values: '1' for A='0', 'Z' for A='1'
	sc_2: *Tristate enabler* for *p2*:
	target: T
	activation condition: B='1'
	controller: A
	values: 'Z' for A='0', '0' for A='1'

FIGURE 6-2. Example merge

new signal concept with condition $cond = cond_1 \cdot cond_2$ is generated. For each value of the controller *A*, two expressions are assigned to target *T*, but only one of them is different from high impedance and it is kept. The new signal concept consists of target *T*, controller *A*, activation condition *B*='1' and value '1' for *A*='0' and value '0' for *A*='1': *sc=<T,A,B='1',['1','0'],{0→'1',1→'0'}>*. It is classified as a *selector*. This signal concept, in some sense, introduces a higher level of abstraction for the behavior of target *T* related to controller *A*. In fact it emphasizes that *A* is used to select values '0' or '1' for signal *T* and it hides technical details on the usage of high impedance. A second signal concept is created by the procedure merge. In fact, since $cond_1 \cdot \overline{cond_2} \neq 0$, a new signal concept *sc=<T,A,B='0',['Z','1'],{0→'1',1→'Z'}>* is created. It is equal to the tristate enabler sc_1 but with a new activation condition. This new concept is generated in order to keep the information that a tristate enabler behavior exists in the code, under a more restrictive condition *B*='0'. Since $\overline{cond_1} \cdot cond_2 = 0$, the tristate enabler behavior represented by signal concept sc_2 is discarded.

6.1.2 Unusual cases

Even if only legacy designs are considered in this dissertation, it might happen that the procedure merge finds a conflict between two signal concepts, that is it finds that more than one expression which is different from high impedance is assigned to the target *tg*. In

ARCHITECTURE beh OF drivers_conflict IS
SIGNAL T : STD_LOGIC;
BEGIN
p1: T <= '1' WHEN B='0' ELSE 'Z';
 	
p2: L <= '1' WHEN A='0' ELSE T;
 	
p3: PROCESS (A)
BEGIN
IF (A ='1') THEN
L <= '0';
ELSE
L <= 'Z';
END IF;
END PROCESS;
END beh;
	sc_1: *Selector* in *p1:*
	target: T
	activation condition: always
	controller: B
	values: '1' for B='0', 'Z' for B='1'
	sc_2:*Selector* for *p2*:
	target: L
	activation condition: always
	controller: A
	values: '1' for A='0', T for A='1'
	sc_3:*Tristate enabler* for *p3*:
	target: L
	activation condition: always
	controller: A
	values: 'Z' for A='0', '0' for A='1'

FIGURE 6-3. Example drivers' conflict

this case the algorithm produces a warning message and the merge operation is not performed. Assuming that the VHDL description is synthesizable and represents a correct digital circuit, the above condition occurs only when the conflict cannot be resolved using only static dataflow analysis. In fact, there are cases in which a circuit behaves correctly only when specific relations among values of inputs are satisfied. Figure 6-3 shows an example in which the algorithm detects a conflict and displays a warning message. In this example, for *A*='1' the signal concept recognized in process *p2* assigns value *T* to *L*, while the signal concept recognized in process *p3* assigns a conflicting value '0' to *L*. Static analysis is not able to resolve the conflict, while a designer, looking at *tristate enabler* for *p2* and *selector* for *p1*, may be able to extrapolate that the circuit works only if *A='1'* always implies *B*='1'. In this case , the logic synthesis tool *Design Compiler* by Synopsys Inc. [Syn00] produces a warning message saying that there is a bus which is not driven by only tristate elements. However the tool implements a working design which is shown in Figure 6-4.

Since, in this dissertation, dynamic analysis techniques are not used, cases like the one in Figure 6-3, in which behaviors depend on relation among values of inputs, are not explicitly handled. However, the warning message, which is displayed by the algorithm, should

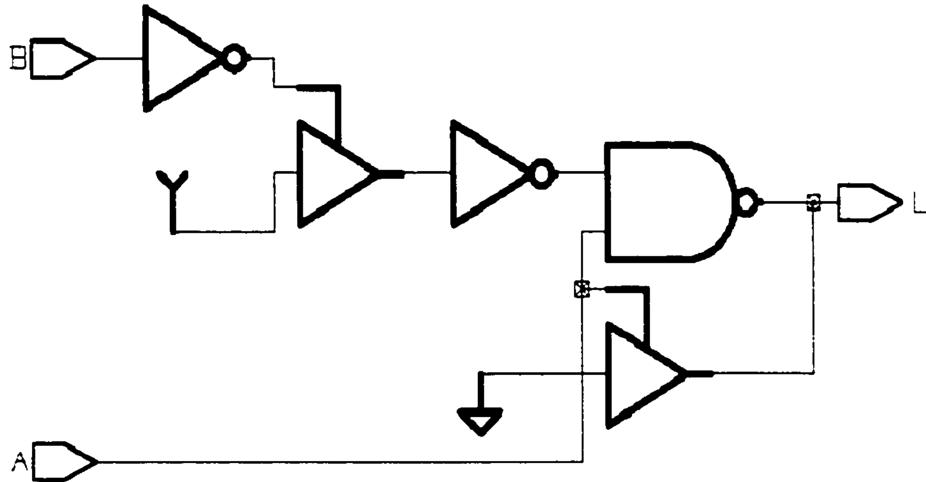


FIGURE 6-4. Synopsys implementation of driver confict example

alert a designer that an explicit investigation of signal concepts regarding the assignments to the target signal is necessary.

In cases of conflict, the property of completeness of signal concepts does not hold. In the rest of this chapter, it is assumed that each signal concepts in the set *S'* satisfies the completeness property. In fact, it is expected that when a warning is issued by the algorithm, that is when only dynamic analysis could solve the conflict, designers will manually update and modify the list of signal concepts such that the property is satisfied. Considering the example in Figure 6-3, a designer will manually merge signal concepts sc_2 and sc_3 and build a new concept with activation condition *B*='1', and values '*1*' for *A*='0' and '0' for *A*='1'.

6.2 Combinational function concepts

Considering combinational modules which are commonly used in hardware digital designs, a set of new abstract concepts, called *combinational function concepts*, are defined. Combinational function concepts intend to represent basic combinational behaviors. Each combinational function concept includes an *activation condition* which indicates when the concept is valid. That is, the behavior represented by the concept is meaningful only when the activation condition is asserted. The activation condition consists of a Boolean expression whose terms represent specific values assigned to predicates.

The different types of combinational function concepts, which are considered in this dissertation, are defined as follows:

- *Encoder*: consists of an *output o*, an *input i*, called also *main input*, and an *activation condition*. For each possible value of i, a particular constant value is assigned to o. The number of bits in i is greater or equal to the number of bits in o. This concept represents a behavior in which for each binary value of the input i, the output o receives a coded binary value.

- *Decoder*: consists of an *output o*, an *input i*, called also *main input*, and an *activation condition*. For each possible value of i, a particular constant value is assigned to o. The number of bits in i is less than the number of bits in o.This concept represents a behavior in which for each binary value of the input i, the output o receives a decoded binary value.

- *Multiplexer*: consists of an *output o*, a *selector s*, also called *main input*, a set I of signals and an *activation condition*. For at least k distinct possible values of s, some $i \in I_r \subseteq I$, where the set I_r is composed of port inputs or internal signals in the architecture, are assigned to o. For the remaining values of s, an assignment to o exists which can be an expression composed of element of the set I. The value k is defined by a designer using a parameter *md* which is called the *multiplexer degree*. $k = \lfloor md \times Values(s) \rfloor$, where $Value(s)$ is the number of possible values of s. For *md* equal to 1, a complete multiplexer behavior is represented. In this case for each value of s, a different signal from I is assigned to o. When *md* is less than 1 an extended multiplexer behavior is defined. A designer may freely choose *md*, between 0 and 1, provided k is greater than or equal to 2. This concept represents behaviors in which different signals $i_k \in I_r \subseteq I$ are assigned to the same output based on the values of a selector s.

- *Demultiplexer*: consists of an *input i*, a *selector s*, called also *main input*, a set O of signals and an *activation condition*. For each value of s, i is assigned to a target signal $o \in O$. The set O, with $|O| \geq k$, consists of port outputs or internal signals in the archi-

tecture. The value k is defined by a designer using a parameter *dd* which is called the *demultiplexer degree*, $k = \lfloor dd \times Values(s) \rfloor$, where *Value*($s$) is the number of possible value assignments of s. For *dd* equal to 1, a complete demultiplexer behavior is represented. In this case, for each value of s, i is assigned to a particular signal $o \in O$. When *dd* is less than 1 an extended demultiplexer behavior is defined. A designer may freely choose *dd*, between 0 and 1, provided k is greater than or equal to 2. This concept represents behaviors in which the same signal i is assigned to different signals $o_k \in O$ based on the values of the selector s.

- *Comparator*: consists of a set O of target signals, two *inputs* i_1, i_2 with the same bit length, called also *main inputs*, and an *activation condition*. For each possible comparison relation between i_1 and i_2, a particular constant value is assigned to each $o_i \in O$. Constant values assigned to a $o_i \in O$ are all different. When inputs i_1, i_2 are represented by a compare predicate, by itself the compare predicate represents a comparison operation. However, the comparator concept is more specific. In fact, it represents a behavior in which specific constant values are assigned to signals depending on the relation between two signal i_1 and i_2.

Even though, intuitively, each combinational function concept in the above list seems to

```
ENTITY encoder_s IS
PORT (   B,C : IN std_logic;
         A : IN std_logic_vector(1 downto 0);
         Y : OUT std_logic_vector(2 downto 0));
END encoder_s;

ARCHITECTURE beh OF encoder_s IS
BEGIN
proc1: PROCESS (A,B)
    BEGIN
        IF (B = '0') THEN
            IF (A = "01") THEN
                Y <= "110";
            ELSIF (A = "10") THEN
                Y <= "001";
            ELSE
                Y <= "ZZZ";
            END IF;
        END IF;
    END PROCESS;

proc2: PROCESS (A,C)
    BEGIN
        IF (C = '1') THEN
            IF (A = "11") THEN
                Y <= "101";
            ELSIF (A = "00") THEN
                Y <= "010";
            ELSE
                Y <= "ZZZ";
            END IF;
        END IF;
    END PROCESS;
END beh;
```

FIGURE 6-5. Example: encoder but not for synthesis

correspond to an equivalent combinational logic module, in reality these concepts represent more generic behaviors and not physical modules. In fact there are design descriptions which include the above abstract behaviors, but whose corresponding synthesized circuits do not include equivalent combinational logic modules. For example, the VHDL description in Figure 6-5, contains a behavior which corresponds to the definition of an encoder concept with input *A*, output *Y* and activation condition *B*='0' AND *C*='1'. In fact, when *B*='0' AND *C*='1' for each possible value of *A*, a constant is assigned to target *Y*. However, when this description is synthesized by a logic synthesis tool like Design Compiler by Synopsys Inc. [Syn00], a circuit including latches, combinational gates and tristate buffers is implemented (see Figure 6-6). The encoder behavior is distributed across the circuit and not clearly identifiable by a designer.

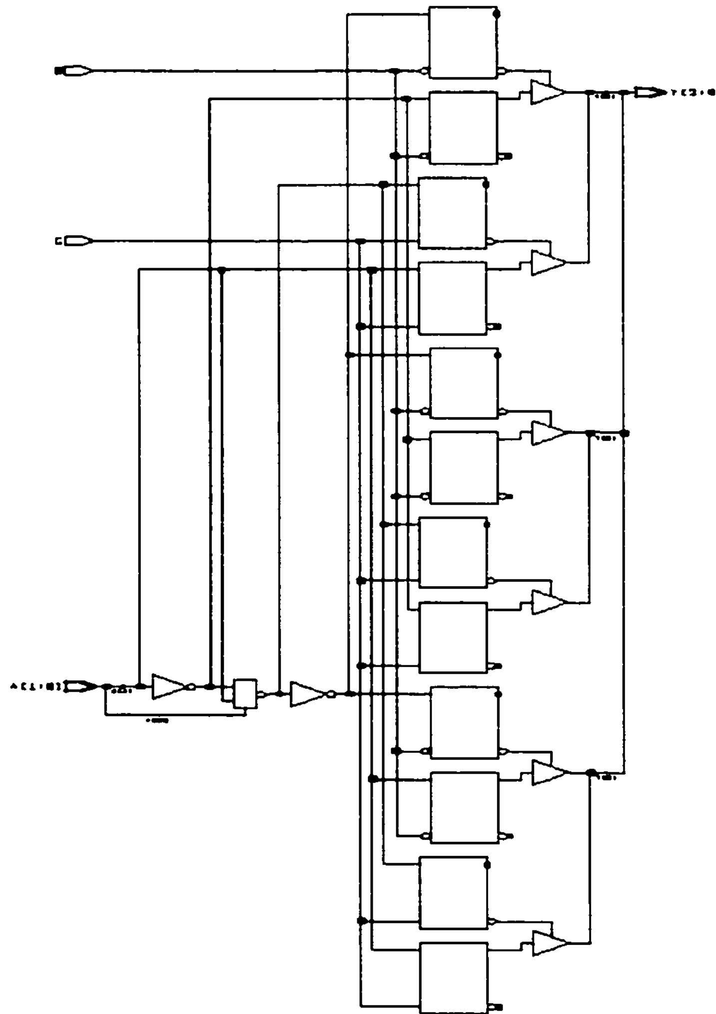


FIGURE 6-6. Synopsys implementation of encoder_s example

Given an architecture and the set S of extracted signal concepts
    Create set S' by merging signal concepts
        for each existing target signal *tg*
            collect in a set L each signal concept $s \in S'$ with target *tg*
            build all possible encoders ,decoders and multiplexers with output tg
        for each existing predicate *pd*
            collect in a set L each signal concept $s \in S'$ with controller *pd*
            build all possible demultiplexers with selector *pd*
            build all possible comparator with inputs i_1,i_2 defined by *pd*

FIGURE 6-7. Pseudo-code: combinational function analysis

6.3 Combinational function analysis

An analysis, called *combinational function analysis*, identifies combinational function concepts looking at extracted signal concepts. A combinational function concept is identified by considering one or more signal concepts. Since an architecture may contain more than one process, it is necessary to examine signal concepts which were extracted from different processes (FPDGs). In fact, the algorithm which performs *combinational function analysis*, considers all signal concepts identified in an architecture disregarding the fact that signal concepts could have been extracted from different processes in the architecture. In Figure 6-7, the pseudo-code of the algorithm which exhaustively looks at each signal concepts and extracts all useful combinational function concepts is depicted.

6.3.1 Identifying encoder, decoder and multiplexer concepts

Theorem 6.2. Given a set S of signal concepts s_i=<$tg,ctrl_i,cond_i,E_i,map_i$> with the same target *tg*, and data *d* which is composed by concatenating $ctrl_k \in C = \{ctrl|$ *ctrl is the controller for some* $s_i \in S\}$.The set $P = Values_{ctrl_1} \times Values_{ctrl_2} \times \cdots \times Values_{ctrl_n}$ represents the domain of *d* as Boolean products. The set S forms a combinational function concept in which for each value $p \in P$ an expression is assigned to *tg* if, and only if, the following conditions are satisfied:

- *cnd1:* Each signal concept s_i=<$tg,ctrl_i,cond_i,E_i,map_i$> must have $\phi \notin E_i$ and $|E_i| = |Values_{ctrl_i}|$.

- *cnd2*: Let $A_p \subseteq D$ be defined as $A_p = \{cond_l | cond_l|_p \neq 0\}$ with $p \in P$. For each $p \in P$, A_p must not be empty.
- *cnd3*: the Boolean expression $b = \prod_{p \in P} \prod_{a \in A_p} a|_p \neq 0$.

Proof. The condition *cnd1* is necessary. In fact, assuming that there is a value $u \in Values_{ctrl_i}$ for which *map*(u)=ϕ. It means that no expression is assigned to *tg* when condition $cond_i \cdot u$ is true. Due to the property of *completeness* of the signal concept s_i, it means that no other signal concept s_j exists which assigns an expression to *tg* when $cond_i \cdot u$ is true. Clearly, $cond_i \cdot u \neq 0$ because $cond_i$ does not depend on $ctrl_i$ by construction, and $u \in Values_{ctrl_i}$. Having $cond_i \cdot u \neq 0$ means that there is at least one input value $p \in P$ for which no expression is assigned to *tg* which is contrary to the theorem. Therefore every $s_i \in S$ must have a $e \in E_i$ for each $u \in Values_{ctrl_i}$.

The condition *cnd2* is necessary. In fact, if *cnd2* is not true, it means that for at least one value $p \in P$ there is no signal concept $s_i \in S$ with $cond_l|_p \neq 0$, that is no signal concept may be activated and no expression is assigned to target *tg*. The existence of an input value for which there is no assignment to *tg* is contrary to the theorem therefore *cnd2* is necessary. However, the condition *cnd1* and *cnd2* are not sufficient as can be shown by the following counter-example.

Consider a set *S* which consists of the following selectors, which satisfy *cnd1* by definition:

- Selector s_1:

  ctrl: *A*;

  activation condition: *B*='0';

  expressions "11" for *A*='0', and "00" for *A*='1'

- Selector s_2:

  ctrl: B;

  activation condition: A='0' AND C='1';

  expressions "11" for B='0', and "01" for B='1'

- Selector s_3:

  ctrl: B;

  activation condition: A='1' AND C='0';

  expressions "00" for B='0', and "10" for B='1'

The condition *cnd2* is satisfied. The domain of d, which is formed by concatenating A and B, is P={"00","01","10","11"} (using explicit values instead of Boolean products for clarity). For each value $p \in P$ at least one *selector* concept is activated, particularly for p="00" selector s_1 and s_2 are activated, for p="01" selector s_2, for p="10" selectors s_1 and s_3 and for p="11" selector s_3. However when p="01" selector s_2 is activated, but only if C='1', while when p="11" selector s_3 is activated, but only if C='0'. Conditions C='*1*' and C='0' cannot be true at the same time, so s_2 and s_3 cannot be activated at the same time. Therefore, there is no common condition which may be considered as an activation condition for the combinational function which must be formed by all signal concepts in the set S.

It can be proved that together the *cnd1*, *cnd2* and *cnd3* conditions are necessary and sufficient, that is $cond1 \wedge cond2 \wedge cond3 \leftrightarrow ED2$ .Given *cnd1* and *cnd2* are true, which must be since they are necessary conditions, it means that for each $p \in P$ there is at least one signal concept $s_i \in S$ which is activated. The signal concept $s_i \in S$ has $cond_l|_p \neq 0$ and it assigns an expression to its target *tg* for $p \in P$. If *cnd3* is true, it means that $b = \prod_{p \in P} \prod_{a \in A_p} a|_p \neq 0$. By construction, b is the logic AND of all $cond_l|_p \neq 0$ representing conditions on which expressions in signal concepts are assigned to *tg* when activated by a $p \in P$. Since $b \neq 0$, it represents a common activa-

tion condition which is satisfied by every selector activated for each $p \in P$. The condition *cond3* is also necessary in fact if b=0, it means that there is no common condition for which all signal concepts in the list S are activated at the same time. Therefore the union of signal concepts in the list S cannot represent a combinational function concept which, by definition, requires a common activation condition. ❑

The procedure which identifies and creates encoder, decoder and multiplexer concepts is based on theorem 6.2. The procedure starts by considering a set L of signal concepts $s \in S'$ with the same target. The pseudo-code, in Figure 6-8, depicts the basic steps of such procedure. The complexity of this procedure is exponential because of step 4.2. In this step subset of signal concepts which may form an encoder, decoder, or multiplexer are identified. Explanations and details of the algorithm which implements step 4.2 may be found in Appendix B.2.

Steps

1 find $L' \subseteq L$ in which each signal concept s_i=<$tg, ctrl_i, cond_i, E_i, map_i$> has $|E_i| = |Values_{ctrl_i}|$, $\phi \notin E_i$, and $ctrl_i$ is not a compare predicate.

2 build the ordered set C={$ctrl$| *ctrl is the controller for some* $s_i \in L'$}, with $|C|=n$

3 build a set $R=\{R_1, R_2, ..., R_n\}$ where $R_i \subseteq L'$ and contains all signal concepts with controller $ctrl_i$

4 for each R_i in R

4.1 set $C = C - ctrl_i$

4.2 considering each possible $2^{|C|}$ concatenations of controllers in C find the largest collection $Q=\{Q_1, Q_2, ..., Q_m\}$ where $Q_k \subseteq R_i$, such that:
- each Q_k satisfies *cnd2* and *cnd3* of theorem 6.2 with data d_k and represents a combinational function concept
- for each Q_k with data d_k, there is no Q_j with data d_j and $j \neq k$, such that d_j is contained in d_k and Q_k is compatible with Q_j

4.3 Each $Q_l \subseteq Q$ is saved as a combinational function concept

FIGURE 6-8. Pseudo-code for extracting encoders, decoders and multiplexers

By definition, a signal concept s_i=<$tg,ctrl_i,cond_i,E_i,map_i$> in L' may represent an encoder, decoder or multiplexer. In fact, a $s_i \in L'$ has $|E_i| = |Values_{ctrl_i}|$ with $\phi \notin E_i$, therefore for each value of $ctrl_i$ an expression is assigned to tg, which is a necessary condition in the definition of an encoder, a decoder and a multiplexer. The actual values of the expressions in E_i determine if the signal concept s_i represents a decoder, an encoder, a multiplexer, or none of them. Considering only single concepts corresponds to looking for encoders, decoders and multiplexers that have in their main input a controller of a single signal concept. However, there might be encoder, decoder and multiplexer behaviors which have the main input consisting of a concatenation of controllers. Steps 2 through step 4.2 in the pseudo-code of Figure 6-8 identify combinational function concepts with main inputs consisting of a concatenation of controllers which are obtained using signal concepts of a set S satisfying theorem 6.2. Step 4.2 identifies combinational function concepts with the largest main input for any possible activation condition which can be obtained from each set R_i. That is, assuming that a combinational function concept c with main input i consisting of controllers $\{ctrl_1,ctrl_2,...,ctrl_m\}$ and activation condition $cond$ exists in R_i. All possible combinational function concepts c_k, in R_i with activation condition $cond_k$, such that $cond \cdot cond_k \neq 0$, and with main input i_k which consists of any combination of controller $\{ctrl_1,ctrl_2,...,ctrl_m\}$ are discarded. The collection Q of step 4.2 consists of subsets $Q_k \subseteq R_i$ of signal concepts which satisfy theorem 6.2 and represent such combinational function concepts. Step 4.2 is repeated n times, one time for each R_i with $|C|=n-i$, that is considering the controller $ctrl_i$ and the remaining controller in the order set C. In this way all possible 2^n combinations of original controllers in C of step 2 are considered.

In step 4.3, each set Q_k in the collection $Q=\{Q_1,Q_2,...,Q_m\}$ is identified as a decoder, an encoder, or a multiplexer. Particularly:

- An encoder or a decoder is built if every signal assignment s_i=<$tg,ctrl_i,cond_i,E_i,map_i$> in Q_j has $\forall v \in V$ equal to some constant. The input i is composed by concatenating $ctrl_i$ of signal concepts in Q_j. The output o is equal to target tg and the activation condi-

tion is given by the expression b of $cnd3$ in theorem 6.2. Moreover, if the number of bits of i is greater or equal to the number of bits of o an encoder concept is actually identified, otherwise a decoder concept is identified.

- A multiplexer is built if there are at least $k = \lfloor md \times Values_s \rfloor$ values $v \in Values_s$ for which tg is assigned a port input or an internal signal in the architecture. The selector s is composed by concatenating all $ctrl_i$ of signal concepts in Q_j. Depending on the value of md, a complete ($md=1$) or extended ($md<1$) multiplexer is created. In this case, the output o is equal to tg, s is the selector, all signals used in every V_i form the set I, and the activation condition is given by the expression b of $cnd3$ in theorem 6.2.

6.3.2 Identifying demultiplexer and comparator concepts

The procedure which identifies and creates demultiplexers and comparators starts by considering a set L of signal concepts $s \in S'$ with the same controller $ctrl$. The procedure aims to group signal concepts with the same controller but different outputs. Signal concepts which are valid for the same condition may exist in the set L. In fact, the merge operation, see Section 6.1, only guarantees that, in S', signal concepts with the same controller and the same target could never be valid at the same time. However, signal concepts, with the same controller but different targets, which can be valid for the same common condition, may exist in S'. The pseudo-code, in Figure 6-9, depicts the basic steps of such a procedure.

By means of the for loop described by step 2 and its sub-steps, the algorithm creates the sets D and C. Each set D_i in D consists of signal concepts which are valid under the same condition and which have the same signal e' which is assigned to some target signals. Each set C_i in C consists of signal concepts $s_i = <tg_i, ctrl, cond_i, E_i, map_i>$ which are valid under the same condition and which have only constant in E_i. Steps 2.1.1 and 2.2.1 use expression $cond_i \cdot cond_k \neq 0$ from theorem 6.1 in order to guarantee that the signal concepts which are collected in each D_i and C_i are valid under the same condition.

Steps	
1	initialize sets D={} and C={}
2	for each signal concept $s_i=<tg_i,ctrl,cond_i,E_i,map_i>$ in L
2.1	if *ctrl* is a compare predicate and $\forall e \in E_i$ is a different constant
2.1.1	if there exists set $C_i \in C$ which contains a signal concept $s_k=<tg_k,ctrl,cond_k,E_k,map_k>$ and $cond_i \cdot cond_k \neq 0$
2.1.1.1	add s_i in the set C_i
2.1.2	else
2.1.2.1	make $C = C \cup \{s_i\}$
2.2	if controller is not a compare predicate and $\forall e \in E_i$ is equal to the same expression e' which consists of a signal
2.2.1	if there exists set $D_i \in D$ which contains a signal concept $s_k=<tg_k,ctrl,cond_k,E_k, map_k>$ with every $e \in E_k$ equals to signal e' and $cond_i \cdot cond_k \neq 0$
2.2.1.1	add s_i in the set D_i
2.2.2	else
2.2.2.1	make $D = D \cup \{s_i\}$
3	for each C_i in C
3.1	a comparator is built
4	for each D_i in D
4.1	a demultiplexer is identified and built if possible

FIGURE 6-9. Pseudo-code for extracting demultiplexers and comparators

In step 3.1, for each set C_i a comparator is created. The set O is composed of all targets of concepts in C_i, the inputs i_1,i_2 are identified by *ctrl*, and the activation condition is given by the Boolean product $cond = \prod cond_k$ of activation conditions of signal concepts $s_k \in C_i$.

In step 4.1, based on parameter *dd* which is set by designer, only D_i which satisfies the condition $|O| \geq k = \lfloor dd \times Values_s \rfloor$, with $O=\{tg|\ tg$ *a target of signal concepts in* $D_i\}$, are recognized as complete or extended demultiplexer. The input i of the demultiplexer is given by the value e' of step 2.2.1, the selector s is equal to *ctrl*, and the activation condi-

tion is given by the Boolean product $cond = \prod cond_k$ of activation conditions of signal concepts $s_k \in D_i$.

The complexity of the algorithm is linear in the number of concepts in the starting set L.

6.4 Concluding remarks

As in the case for signal analysis, combinational function analysis extracts useful behaviors which are embedded in a component description. Combinational function concepts are artifacts which represents functional relations among signals which are more complex than the functional relations identified by signal concepts. Combinational function analysis locates combinational function concepts under different activation conditions. For example, two encoders with the same input i and the same output o, but with different activation conditions may be identified. These encoders corresponds to two different ways of looking at the functional relation between input i and output o. Examples of combinational function concepts will be presented in Chapter 9 where different designs are analyzed.

Chapter 7 Concept grouping and design partitioning

Once signal analysis and combinational functional analysis have been completed on a VHDL architecture, a set of signal and combinational function concepts exists. Some of these concepts are directly related to each other. For example, they may have a target signal in common, a common activation condition, or they may exhibit data dependencies through their main inputs, inputs and outputs. In this chapter, analyses, which use different relations amongst the above set of identified abstract concepts, are described. The notion of equivalence amongst abstract concepts is discussed in the first section. The extra control analysis, which is used by the interactive tool (see Chapter 8) to group concepts is also explained. In the last section, functional partitioning analysis is described. This analysis partitions the behavior of an architecture amongst modules which represent new abstract artifacts. Before reading this chapter, it would be useful to review the definitions of signal and combinational concepts and of the main input of an abstract concept.

7.1 Equivalences

Considering an assignment in a VHDL description, there might be more than one predicate whose values influence the execution of the assignment. Intuitively, it might happen that two or more predicate influence the same set of assignments. Therefore, since abstract

ARCHITECTURE beh OF equivalence1 IS
BEGIN
PROCESS (A,B)
BEGIN
IF (A ='1') OR (B='0') THEN
Z <= "0110";
ELSE
Z <= D+E;
END IF;
END PROCESS;
END beh;
	sc_1: *Selector*:
	target: Z, controller: B
	activation condition: A='0'
	values: "0110" for B='0', D+E for B='1'
	sc_2: *Selector*:
	target: Z , controller: A
	activation condition: B='1'
	values: D+E for A='0', "0110" for A='1'

FIGURE 7-1. Example equivalence1

concepts are derived from conditional assignments, which represent a collection of assignments to a specific target signal which are activated for different values of a predicate (see Section 5.2.2), it could happen that two or more abstract concepts refer to the same set of expressions. As a simple example, Figure 7-1 depicts a VHDL architecture and two abstract concepts extracted by signal analysis after having analyzed the equivalent FPDG.

It is evident that the two abstract concepts represent the same behavior, it is only the point of view that is changed. In the first abstract concept sc_1, the information regarding controlling of assignments by the predicate A is explicitly expressed, while in sc_2 the information regarding controlling of assignments by predicate B is underlined. Moreover, even if the two abstract concepts have different conditions, they are compatible, that is they may be both true at the same time. In a slightly different example, depicted in Figure 7-2 , a detailed analysis of the extracted signal and combinational function concepts shows that,

ARCHITECTURE beh OF equivalence IS
BEGIN
PROCESS (A,B)
BEGIN
IF A ='0' THEN
IF B='1' THEN
Z <= "0110";
ELSE
Z <= "1111";
END IF
END IF;
END PROCESS;
END beh;
	Decoder:
	output: Z, input: B
	activation condition: A='0'
	values: "1111" for B='0', "0110" for B='1'
	Enabler:
	target: Z , controller: A
	activation condition: B='1'
	values: "0110" for A='0'
	Enabler:
	target: Z , controller: A
	activation condition: B='0'
	values: "1111" for A='0'

FIGURE 7-2. Example equivalence2

together, the two enabler concepts represent the same behavior of the decoder concept. In fact, in both cases, the same values are assigned to Z for the same values of A and B.

From the above examples, it is clear that the set of abstract concepts which is obtained by signal and combinational function analysis contains some redundant behavioral information. Therefore, even if it is important to extract all possible abstract concepts to keep all available information in a VHDL description, it is also important to be aware of equivalences among them. In fact, such equivalences emphasize different ways of interpreting the same behavior and may improve the understanding of the original design.

Equivalence definition. Two set $A=\{a_1,a_2,...,a_n\}$ and $B=\{b_1,b_2,...,b_m\}$ of abstract concepts are *equivalent* if:

- all $a \in A$ and all $b \in B$ have the same set O of outputs
- all $a \in A$ have the same main input c_A, all $b \in B$ have the same main input c_B and $c_A \neq c_B$
- considering the set $Values_{c_A}$ of all possible values of c_A and the set $Values_{c_B}$ of all possible values of c_B, the cartesian product $V = Values_{c_A} \times Values_{c_B}$ represents the domain of c_A concatenated with the domain of c_B. If O_a is the ordered set of the expressions assigned to all outputs in O for $v \in V$ by abstract concepts $a \in A$, and O_b is the ordered set of the expressions assigned to all outputs in O for $v \in V$ by abstract concepts $b \in B$, then O_a must be equal to O_b.

General equivalence problem. Given a set L of abstract concepts which have the same set O of output signals, finding all possible tuples from sets $A \subseteq L$ and $B \subseteq L$ which are equivalent is an intractable problem. In fact given $I=\{i_1,i_2,...,i_n\}$ the set of main inputs such that for each $i \in I$ there is at least one abstract concept $h \in L$, it is necessary to consider $\binom{n}{2}$ possible tuples $t=<i_j,i_k>$ with $j \neq k$. For each of such tuple $t=<i_j,i_k>$, let $F \subset L$

be the set of abstract concepts with main input i_j, and let $S \subset L$ be the set of abstract concepts with main input i_k, it is necessary to consider each set $A \in 2^F$. For each set A it is necessary to consider each set $B \in 2^S$ and check if A is equivalent to B. Therefore for each of the $\binom{n}{2}$ tuples the complexity of the algorithm to find all possible equivalent tuple of set of abstract concepts is upper bounded by $2^{|F|} \cdot 2^{|S|}$. There might be an exponential number of solution of the general equivalence problem.

In this dissertation, it is not necessary to solve the general equivalence problem. The interactive tool, which is described in Chapter 8, needs to solve the problem of identifying abstract concepts which are equivalent to a given set of abstract concepts. In Chapter 8, details on finding those equivalences are given.

7.2 Extra controls analysis

Let $C=\{c_1,c_2,...,c_n\}$ be a set of abstract concepts with the same main input i and target tg in their output set. By construction (see *property 2* in Section 5.3 and the merge operation in Section 6.1), each abstract concept $c_j \in C$ has its activation condition $cond_j$ which is different from each activation condition $cond_k$ of $c_k \in C$ where $j \neq k$. Moreover the Boolean product $cond_j \cdot cond_k = 0$ for each $k \neq j$. If the set C consists of more than one abstract concept, it means that, depending on different activation condition. different expressions are assigned to tg depending on values of input i. Since these behaviors are valid under different conditions, it means that there are values of predicates, different from i, which identify when each of the abstract concepts is valid. It is the intention of an analysis, called *extra controls analysis*, to look at each activation condition of the abstract concepts in C and to extract predicates which control the activation of the concepts. Particularly the extra control analysis identifies and classifies a predicate as:

- *Extra selector*: if each value of the predicate contributes to activating one different abstract concepts in C.

- *Extra partial selector*: if each value, but not all, of the predicate contributes to activating one different abstract concepts in C. Remaining values of the predicate do not contribute to activating any abstract concept.
- *Extra enabler*: if a specific value of the predicate contributes to activating one or more abstract concepts in C. Remaining values of the predicate do not contribute to activating any abstract concept.

A value of a predicate contributes to activating a concept if the value is explicitly used in the activation condition of the concept. For example, if a concept c_1 has $cond_1 = x_0\bar{x}_1$, where x_0 corresponds to A='1' and $\bar{x}_1$ corresponds to B='0', then value '1' (x_0) of A contributes to activate the concept as well as value '0' ($\bar{x}_1$) of B. Not all predicates whose values are used in the activation conditions of concepts in C, may be classified in one of the above category. In fact, there are cases where different values of a predicate activate more than one abstract concept, therefore the predicate does not belong to any of the above categories.

After identifying all predicates, different from i, which satisfy one of the above situations, the extra controls analysis extracts the larger possible set $G = \bigcup_{k=1}^{n} G_k$ with $G_j \cap G_l = \varnothing$ for each $j \neq l$, and $G \subseteq C$. Each G_k is a set of abstract concepts whose activation is contributed to by an extra selector, an extra partial selector or an extra enabler. In the case that more than one set G of maximum size exists, the extra controls analysis choses the set G with the minimum number n of partitions G_k.

The idea behind extra control analysis is to collect together in a group, abstract concepts whose activation is explicitly controlled by a predicate. The predicate is the dominant predicate among the predicates which contribute to activating the abstract concept in the group. In this way, designers may concentrate on a subset of abstract concepts while inspecting the behavior of a specific target tg based on values of a specific input i. As an example, Table 7-1 reports the set C of abstract concepts with the same input A and output

Abstract concept	Output	Input	ActivatiionCondition
Decoder	T	A	E='1' AND B='0'
Set/reset	T	A	E='0' AND B='0' AND D='0'
Selector	T	A	E='0' AND B='0' AND D='1'

Table 7-1. Example extra controls analysis

T, which have been obtained from the VHDL description shown in Figure 7-3. The extra controls analysis recognizes:

- Predicate *B* as an extra enabler. In fact value '0' of *B* contributes to activating all three abstract concepts.
- Predicate *D* as an extra selector. In fact value '0' of *D* contributes to activating the set/reset concept, while value *'1'* of *D* contributes to activate the selector concept

Predicate *E* is not categorized since *E='0'* contributes to the activation of two abstract concepts.

At this point, the extra controls analysis uses the above information to extract the set *G*. The extra enabler *B* controls all three concepts in *C* which form a set G_1. The extra selec-

```
ARCHITECTURE beh OF extra_controls IS
BEGIN
proc1: PROCESS(A,B,E)
BEGIN
    IF E='1' AND B='0' THEN
        IF A='0' THEN T <= "0010";
        ELSE T <= "1100";
        END IF;
    END IF;
END PROCESS;
proc2: PROCESS (A,B,D,E)
BEGIN
    IF E='0' THEN
        IF A='1' THEN
            IF B='0' THEN T <= F AND "0101"; END IF;
        ELSE
            IF D = '0' THEN T <= "0111";
            ELSE T <= F+G;
            END IF;
        END IF;
    END IF;
END PROCESS;
END beh;
```

FIGURE 7-3. Example extra controls analysis

tor D controls only two of the three concepts in C. The two concepts form a set G_2. Since $G_1 \cap G_2 \neq \varnothing$, the extra control analysis identifies $G=G_1$. The set G_1 is chosen because it is a larger set then G_2.

7.3 Functional partitioning

The analyses presented in the previous two sections focus on the relations among abstract concepts with a common output signal. Those analyses aim to clarify how multiple abstract concepts cooperate to define values for a specific output signal. In this section an analysis, called *functional partitioning*, whose goal is to identify the role of a signal or a set of signals within the overall behavior of an architecture, is described. The idea is to partition the behavior found in a design into a set of modules which represents standard hardware behaviors that are commonly found in digital designs.

By looking at data dependencies among the set of abstract concepts, which have been extracted by signal and combinational function analysis and which constitute the behavior of an architecture, functional partitioning analysis is able to recognize four different hardware behaviors: *fsm module*, *complex register module*, *simple register module* and *combinational module* behaviors.

7.3.1 Fsm module behavior

Definition. An *fsm module* is characterized by a signal called *state*. The state signal requires a memory element to maintain its value, and its value is explicitly controlled by its own previous value. It means that there is at least one value of the state signal which, possibly together with other signal values, explicitly determines a new value for itself.

An fsm module represents sequential behavior with an internal signal, *i.e.* state signal, which controls that behavior. The word "explicitly" in the definition is important, in fact it is used to point out that the control activity of the state signal must have been explicitly declared in the description of the analyzed design. By tying the definition to a characteristic of the description of a design, functional partitioning analysis is able to pinpoint a spe-

```
ARCHITECTURE beh OF not_fsm IS
SIGNAL A,B : STD_LOGIC;
BEGIN
    B <= A XOR I;

    proc1: PROCESS(CK)
    BEGIN
        IF CK='1' AND CK'EVENT THEN
            A <= B;
        END IF;
    END PROCESS;
END beh;
```

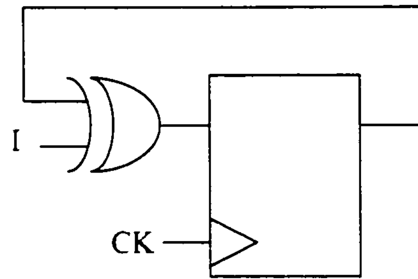


FIGURE 7-4. Example: not fsm module

cific behavior which was important for the original designer. The assumption is that a designer explicitly describes the control function of a signal because it is a relevant characteristic of the overall behavior. This capability of the functional partitioning analysis fulfils a purpose of this dissertation, that is to capture the original intent of a design.

The hardware behaviors which are recognized as fsm modules by the functional partitioning analysis are the behaviors of a Moore or a Mealy finite state machine [Kat94]. However, a behavior is recognized if, and only if, at least one next state transition of a finite state machine has been explicitly declared in the description by means of a condition on a value of the state signal. For example, from the VHDL description in Figure 7-4 which represents a behavior, which is equivalent to the logic circuit depicted on the right side of the figure, no fsm module is identified by the functional partitioning analysis. However, from the VHDL description shown in Figure 7-5 which represents the same behavior, a fsm module is identified by the functional partitioning analysis. In this description the designer explicitly indicates that the value of *A* is relevant to determining the new value of

```
ARCHITECTURE beh OF fsm IS
SIGNAL A,B : STD_LOGIC;
BEGIN
    B <= I WHEN A='0',
        NOT I WHEN A='1';

    proc1: PROCESS(CK)
    BEGIN
        IF CK='1' AND CK'EVENT THEN
            A <= B;
        END IF;
    END PROCESS;
END beh;
```

FIGURE 7-5. Example: fsm module

B, and subsequently the next value of A. In the description of Figure 7-5, the state nature of signal *A* is stressed, therefore functional partitioning correctly recognizes a fsm module behavior. In the description of Figure 7-4, designer focus on the fact that signal *A* acts as a register which maintains a value which is obtained by the expression *A XOR I*. The fact that the new value of A depends on its previous value, because one of the two signals in the *xor* operation is *A*, it is not emphasized in the description. In this case, functional partitioning analysis recognizes a complex register module behavior. This differentiation, which is performed during the functional partitioning analysis, is important. In fact it aims to recognize what the designer had in mind when he or she was writing the VHDL specification.

7.3.2 Complex register module behavior

Definition. A *complex register module* is characterized by one or more signals called *registers*. Values of registers are kept in memory elements. Unlike the fsm module, there is no value of any register which explicitly determines a new value for itself. However, the value of a register depends on its previous value.

A complex register module is used to represent sequential behavior with one or more internal signals, *i.e.* registers, whose values depend on their previous values, but which do not control their own values. It means that a description, in which a complex register module is recognized, does not contain any explicit reference to a value of any register. Signal *A*, in the example in Figure 7-4, is identified as a register.

7.3.3 Simple register module behavior

Definition. A *simple register module* is characterized by one or more signals called *registers*. Values of registers are kept in memory elements.The value of each register does not depend in any way on its own previous values but it might depend on values of other register in the module.

A simple register module is used to represent a memory element or a set of memory elements, *i.e.* registers, whose values do not depend on their own previous values. It means

Code	Concepts

ARCHITECTURE beh OF simplereg1 IS
SIGNAL S : STD_LOGIC;
BEGIN
    O <= S WHEN B='0' ELSE 'Z';
    S <= D WHEN A='1';
END beh;
	Enabler:
	    target: S, controller: A
	    activation condition: always
	    values: D for A='1'
	Tristate Enabler:
	    target: O , controller: B
	    activation condition: always
	    values: S for B='0', 'Z' for B='1'

FIGURE 7-6. Example: simple regisetr 1

that each register is used to memorize values at certain instants and these values are then used by other module behaviors or transfered to output ports.

An example where a simple register module is recognized is reported in Figure 7-6. The enabler concept indicates that a memory element is necessary to maintain the value of target *S*. The expressions assigned to *S* depend only on values of input port *D*. A simple register module is created with *S* as a register. A slightly more complex example is reported in Figure 7-7. The enabler and the logic net concepts indicate that a memory element is respectively necessary for signal *O* and *S2*. Since values of *O* depend on values of *S2* through values of *S1*, a simple register module is identified with *O* and *S2* as registers.

Code	Concepts

ARCHITECTURE beh OF simplereg2 IS
SIGNAL S1,S2 : STD_LOGIC_VECTOR(3 downto 0);
BEGIN
    O <= S1 + D WHEN B='1';
 	
    S1 <= S2 WHEN A='0' ELSE "0000";
 	
    PROCESS (CK)
    BEGIN
        IF CK='1' AND CK'EVENT THEN
            S2 <= E AND D;
        END IF;
    END PROCESS;
END beh;
	Enabler:
	  target: O, controller: B
	  activation condition: always
	  values: S1+D for B='1'
	Selector:
	  target: S1 , controller: A
	  activation condition: always
	  values: S2 for A='0', "0000" for A='1'
	Logic Net:
	  target S2
	  activation condition rising edge of CK
	  value: E AND D

FIGURE 7-7. Example: simple register 2

7.3.4 Combinational module behavior

Definition. A *combinational module* is characterized by one or more signals which do not require any memory element. Their values are continuously determined by values of other signals.

A combinational module is equivalent to a combinational logic circuit. Abstract concepts, which have the activation condition always valid and a value defined for each possible value of their input, are recognized as combinational modules.

7.3.5 Algorithm to extract functional modules

Functional partitioning analysis uses a directed graph $G=(V,E)$. Each vertex represents a signal *tg*. All abstract concepts which have the signal *tg* in their set of output signals are collected in the vertex. Edges represent data dependencies between vertices. Particularly, an edge from vertex v_i to vertex v_j may represent:

1. a data dependence between the output *tg* of abstract concepts in v_i and a main input of a abstract concept in v_j
2. a data dependence between the output *tg* of abstract concepts in v_i and a value which is used by an abstract concept in v_j

Given an architecture and the set S of abstract concepts which have been extracted by the signal and combinational function analyses, functional partitioning analysis partitions the set S into disjoint sets $S_1, S_2, \ldots, S_n$. Each S_i corresponds to a module behavior.

The pseudo-code, in Figure 7-8, describes an algorithm which performs functional partitioning analysis. The algorithm, first proceeds by identifying all possible fsm modules, then all possible complex register modules, all possible simple register modules and finally creating combinational modules from the remaining abstract concepts which are not used in previously recognized modules. After creating all modules of types fsm and complex register, and after creating each module of type simple register or complex register, the directed graph G is updated by removing used vertices and the associated edges.

```
considering S, build the directed graph G=(V,E)
    for each v_i ∈ V in G
        try to build a fsm module f
    remove from G all vertices which are used by identified fsm modules
    for each v_i ∈ V in G
        try to build a complex register module f
    remove from G all vertices which are used by found complex register modules
    for each v_i ∈ V in G
        try to build a simple register module f
        if simple register module f has been created
            remove from G all vertices which are used in f
    for each v_i ∈ V in G
        build a combinational module f
        remove from G all vertices which are used in f
```

FIGURE 7-8. Pseudo-code for functional partitioning

As a result of the above algorithm, the original set S is partitioned into subsets which are represented by module behaviors. Each module behavior is eventually identified and built by starting from a vertex v_i, and considering other vertices and edges in the graph as follow:

- *Fsm module behavior*. Considering out-going edges from v_i, if an edge representing a data dependence with a main input exists and belongs to a loop in G which include v_i, then the output associated to v_i is recognized as a state signal and a fsm module f *is created*. All vertices, and their associated abstract concepts, which belong to the loop are collected in the fsm module. Each edge, which is used in the loop, is removed from the original G. In this way when the graph G is analyzed again to search for a loop, the edge is not anymore considered.

- *Complex register module*. Since complex register modules are recognized after all possible fsm modules have been created, all remained loops in the graph G do not contain any edge which represents a data dependence with a main input. In fact all edges which represent data dependence with a main input have been used to identify fsm modules and have been removed from the graph G. Considering out-going edges from v_i, if a

loop which includes v_i is found, then the output associated to v_i is recognized as a register and a complex register module f is created. Each output signal, which corresponds to a vertex in the loop, becomes a register in f, if it needs a memory element to maintain its value. All vertices, and their associated abstract concepts, which belong to the loop are collected in the complex register module. Each edge, which is used in the loop, is removed from the original G. In this way when the graph G is analyzed again to search for a loop, the edge is no longer considered.

- *Simple register module.* Since simple register modules are recognized after all possible complex register modules have been created and no loop is left in the graph $G=(V,E)$. In fact all edges, which belong to a loop, have been used to identify complex register modules and have been removed from the graph G. An abstract concept whose activation condition includes a clock edge, or whose values are not determined for all possible values of its input requires a memory element to keep its values. Considering abstract concepts in vertex v_i, if at least one of them requires a memory element to keep its value, then the output associated to v_i is recognized as a register and a simple register module f is created. The vertex v_i must not belong to any loop in G. Considering the set $A \subseteq V$ of vertices which are reachable from v_i, all vertices $A' \subseteq A$ which require a memory element are collected in f. The outputs associated to each vertex $v_j \in A'$ become registers in f. Considering all paths from v_i to $v_j \in A'$, all vertices in the path which belong to A-A' are also collected in f. Similarly, considering the set $B \subseteq V$ of vertices which reach the vertex v_i, all vertices $B' \subseteq B$ which require a memory element are collected in f. The outputs associated to each vertex $v_j \in B'$ become registers in f. Considering all paths from $v_j \in B'$ to v_i, all vertices in the path which belong to B-B' are also collected in f.

- *Combinational module.* Since combinational module are recognized after all possible simple register have been created, no vertices, which contains abstract concepts which require a memory element, exist. A combinational module f is built for vertex v_i. All vertices, which are reachable from v_i, are collected in f. Similarly, all vertices from which the vertex v_i can be reached are also collected in f.

Each identified module f contains a set of abstract concepts L. By examining the abstract concepts in L, each module f is completed with a set I of input signals and a set O of output signals. In particular, the set I consists of all signals which are used in expressions which are assigned to outputs of abstract concepts in L. The set O consists of all signals which are outputs of abstract concepts in L.

The procedure which finds loops in the graph $G=(V,E)$ starting from a node follows a depth-first search traversal. Therefore the complexity of the algorithm is $O(V+E)$.

7.3.6 Recent related literature

Recently, a research work by Liu and Jou [LJ00] has been published on extracting controlling finite state machines (FSMs) from RTL descriptions to facilitate the design verification process. Like the approach in this dissertation, FSMs are recognized by finding loops in a graph. In contrast to the approach in this dissertation (see Section 7.3.5), each node in the graph corresponds to an entire VHDL process. Therefore, as reported by Chien-Nan *et al*, an extra activity, called *functional dependency checking*, is needed to accept each identified loop. Particularly, the functional dependency check activity is used to verify that a loop creates a circular data dependence on a signal which is assigned in a session of the VHDL which is conditioned by a clock expression.

Chien-Nan *et al* identify two different type of finite state machines, controlling FSMs and non-controlling FSMs, based on the same principle of explicit control which is used to differentiate between fsm and complex register modules in this dissertation. However Chien-Nan *et al* use the principle that not all FSMs belong to the control part of a circuit. In this dissertation, the principle is obtained considering what a designer had in mind when he or

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY complex_reg IS
PORT (
clock,sample,sel : IN  STD_LOGIC;
data_in   : IN  STD_LOGIC_VECTOR(1 DOWNTO 0);
data_out  : OUT STD_LOGIC
     );
END complex_reg;

ARCHITECTURE beh OF complex_reg IS
SIGNAL load_data,prev_data : STD_LOGIC_VECTOR(1 DOWNTO 0);
BEGIN
latch_proc: PROCESS(sel)
BEGIN
     IF (sample = '1') THEN
          load_data <= prev_data OR data_in;
     END IF;
END PROCESS;
clock_proc: PROCESS (clock)
   BEGIN
     IF clock'event AND clock = '1' THEN
          IF (sel = '1') THEN
               prev_data <= data_in;
          ELSE
               prev_data <= load_data;
          END IF;
     END IF;
   END PROCESS;
data_out <= prev_data(0) AND prev_data(1);
END beh;
```

FIGURE 7-9. Complex register example

she described the design, and not considering the controlling nature of a finite state machine. Moreover, while the definition of controlling FSM is equivalent to the definition of a fsm module, the definition of a non-controlling FSM represents a subset of complex register modules. In fact a complex register module may have more than one register in it, while a non-controlling FSM may have only one register. The difference occurs for two reasons, that is because a node in the graph is represented by a whole VHDL process and because only loops with no more than one sequential node in it are accepted by the method in [LJ00]. Having a whole VHDL process represented by a node implies that only two types of nodes are considered: sequential and combinational nodes. A sequential node is a process with a clock expression, while a combinational node is a process with no clock expression. In this way, a signal, which requires a memory element to maintain its value but which is not controlled by a clock signal, is embedded in a combinational node and not

considered as a possible register. It is not possible to introduce an extra type of node into the approach by Chien-Nan *et al*, since in the same process both pure combinational and sequential behaviors with no clock may be described. It would be necessary to look inside the process and split it into different nodes. Since in this dissertation, nodes in the graph, where loops are identified, consist of abstract concepts, which may represent only a part of a process, signals which are not under control of a clock but which require memory elements to keep their values are also considered. Therefore, the functional partitioning analysis presented in this dissertation is capable to identify sequential behaviors (complex registers) which are not identified by the research work in the article [LJ00]. For example, based on what is described in the article, no finite state machine should be recognized from the simple VHDL description reported in Figure 7-9 by the algorithm described by Chien-Nan *et al.* However, functional partitioning recognizes a complex register with signals *load_data* and *prev_data* as registers.

7.4 Concluding remarks

Abstract concepts which are obtained by signal and combinational function analyses are artifacts which represents behaviors embedded in a component description. In particular, abstract concepts describe functional relations amongst signals under different activation conditions. In this chapter, analyses which group such abstract concepts into meaningful subsets have been described. Functional partitioning analysis groups abstract concepts in order to explicitly identifies parts of the analyzed design which perform as known sequential behaviors or as combinational behaviors. Equivalence and extra control analysis group abstract concepts in order to facilitate the investigation of the functional behavior associated to a set of related abstract concepts. Equivalence and extra control analysis are interactively used by a designer during the information exploration activity which is explained in the next chapter.

Chapter 8 Interactive exploration: VALET tool

In previous chapters, a set of analyses have been described. These analyses create artifacts which represent different aspects of the behavior of a HDL specification. In this chapter, a software tool, VHDL Assistant Low Effort Tool (*VALET*), which allows designers to extract and analyze these artifacts, is described. The tool provides an interactive environment in which designers can explore the artifacts and their relationships to recover the original intent of legacy HDL code.

In this dissertation, similarly to program understanding in the software reverse-engineering area, the process of recovering the original intent of a legacy HDL specification consists of a composition of activities supported by a reverse-engineering environment. Data gathering, knowledge management and information exploration are the activities which are used in most reverse-engineering environments [Til98]. Data gathering consists of collecting raw data from the source code of a software system. In the software domain, raw data correspond to artifacts like function calls, use-def clauses, data-flow or control flow information *etc*. Knowledge management refers to capturing, discovering and organizing past experience and domain knowledge. Abstraction mechanisms are used to create artifacts which contain some of the domain knowledge. Information exploration is the most important activity of any reverse-engineering process. During this activity, software engineers navigate through previously collected artifacts and analyze them to infer higher level concepts. Generally, automatic, semi-automatic or manual analysis supports the explora-

tion activity. Moreover, different metaphors and views may be used to assist the software engineer during inspection

In this research, in order to assist designers in recovering the original intent of legacy HDL code, a software tool, VHDL Assistant Low Effort Tool (*VALET*), was developed. In this software prototype, support for activities, which correspond to data gathering, knowledge management and information exploration activities, is provided. In particular, VHDL parsing, signal analysis, combinational analysis and functional partitioning analysis are implemented in the tool. A graphical user interface, which enables interactive navigation, analysis and exploration of identified abstract concepts and modules is also implemented.

Before describing VALET, it is important to indicate that during the information exploration activity a mental model is obtained by the person performing the exploration. A *mental model* is a mental representation of the system being analyzed. In the software reverse-engineering area, different studies [MV95] have been conducted to identify the cognitive models which are used by software engineers to form a mental model while engaged in program understanding. A *cognitive model* describes the cognitive processes and knowledge structures used to form a mental model [Til98].

In [SFM97] Storey *et al.* describe a hierarchy of cognitive design elements which should be considered during the design of a software exploration tool. These cognitive design elements are based on cognitive models and comprehension strategies which are commonly employed in program understanding.

To the best knowledge of the author, no studies exist on identifying cognitive models employed by hardware engineers in forming a mental model when analyzing an existing design. Furthermore, no collection of cognitive design elements which should be employed in designing exploration tools for understanding of hardware design exists. Consequently, based on the assumption that cognitive models similar to the one employed by software engineers are used by hardware engineers, some guidelines indicated in [SFM97] are adopted here.

Particularly in VALET:

- Semantic relationships in the form of data-flow and data dependencies graphs are indicated to the user.
- Abstraction mechanisms are supported. The user can also label and document the extracted artifacts to record results of the investigation.
- Various level of abstractions are provided.

8.1 VALET structure

The structure of VALET with all of its components is shown in Figure 8-1. The tool has access to a repository of VHDL components. During analysis and exploration activities auxiliary data are saved in the repository.

VALET consists of a *Navigator*, an *Analyzer* and three types of viewers: *Module*, *Target Group* and *Abstract Signal*. The Navigator is the primary interface to the tool. It allows designers to load a new design from the repository, to navigate through instances, architectures and processes of the design, and to invoke analyses on a selected architecture. The Analyzer is the module which actually performs all different kinds of analyses: signal, combinational function and functional partitioning analyses. These analyses together perform activities similar to the data gathering and the knowledge management activities

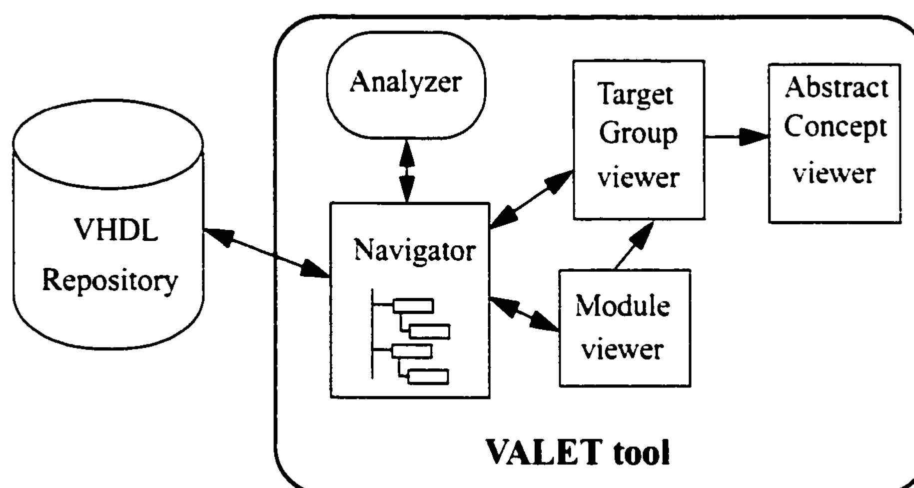


FIGURE 8-1. VALET structure

which are part of any reverse engineering process. In fact abstract concepts and modules are obtained by examining raw data, like assignments, conditional predicates, data and control dependencies, and by recognizing behaviors which belong to a specific domain, *i.e.* digital hardware domain. Information exploration activity is performed by designers using the three different type of viewers. These viewers allows designers to view and analyze the extracted concepts and modules.

During the exploration activity, the designer is the main director, and viewers aid a designer in the process of identifying the intent of a design. The following considerations have been taken into account in defining the three type of viewers and their interrelationships.

- Hardware designers focus on the behavior of a wire when analyzing an actual circuit. For example during functional testing, probes are used to checks values of specific wires. By the logic synthesis process, a signal in VHDL is always mapped to a wire in the final circuit, that is to an output or an input of a logic element (unless it was never actually used - dead code). For this reason, the *Target Group viewer* was developed. In this viewer, abstract concepts which have a specific signal *tg* amongst their outputs are shown. Using this viewer, a designer can concentrate on the understanding of functional behaviors attached to a specific signal.
- Modules and abstract concepts are hierarchically related by construction. In fact, modules are formed by aggregating a set of abstract concepts. A specific *Module viewer* was developed which shows interrelationships amongst concepts in the module. Particularly, data dependencies amongst group of concepts are shown. Based on the previous observation, this groups corresponds to the aggregate of concepts with the same signal amongst their output. Therefore, the Module viewer uses the Target Group viewer to enable designers to investigate behaviors associated to each group.
- Generally, knowing the type of an abstract concept and its associated signals should be sufficient for a designer to get a basic understanding of the intent of a design. However, there are cases, in which a designer might need detailed information about values

assigned to each signal. The Abstract Concept viewer allows a designer to interactively inspect the internal behavior of a concept by assigning values to its main inputs.

From the point of view of a user of VALET, by selecting a module, an instance of the Module viewer is invoked, while by selecting a specific signal, an instance of the Target Group viewer is invoked. Also from a Module viewer, an instance of the Target Group viewer may be invoked by selecting a group which correspond to a signal in the module. An instance of the Abstract Concept viewer may be used to investigate the behavior of a concept in more detail.

8.2 VALET components and user interface

Details on the activity performed by the component analyzer were given in chapters 5,6 and 7. In this section, details on the capabilities of the other components, which compose the VALET tool, are described.

8.2.1 Main window: Navigator

At start-up, VALET opens a main window which is shown in Figure 8-2. Using a command from the *File* menu, a design consisting of one or more components may be loaded into VALET from the repository of legacy designs. Once a design is loaded, the VHDL code and all its elements are reported in different parts of the window which represent the Navigator component. Each VHDL entity belonging to a design and its hierarchical structure is shown using a tree like representation. From this tree structure, a designer can select a specific instance or architecture to analyze. Besides listing the VHDL code of the selected component, the Navigator interface reports all input, output and inout ports which represent the interface of the component. Internal elements, that is processes, internal signals, variables and constants, which are used in describing the component are also shown.

By selecting a port or any of the internal elements, the lines of VHDL code which define and use the selected object, are highlighted. If necessary, a designer may use these elements of the Navigator interface to manually inspect the VHDL source code. However manual inspection of VHDL code is not the goal of the proposed methodology, therefore

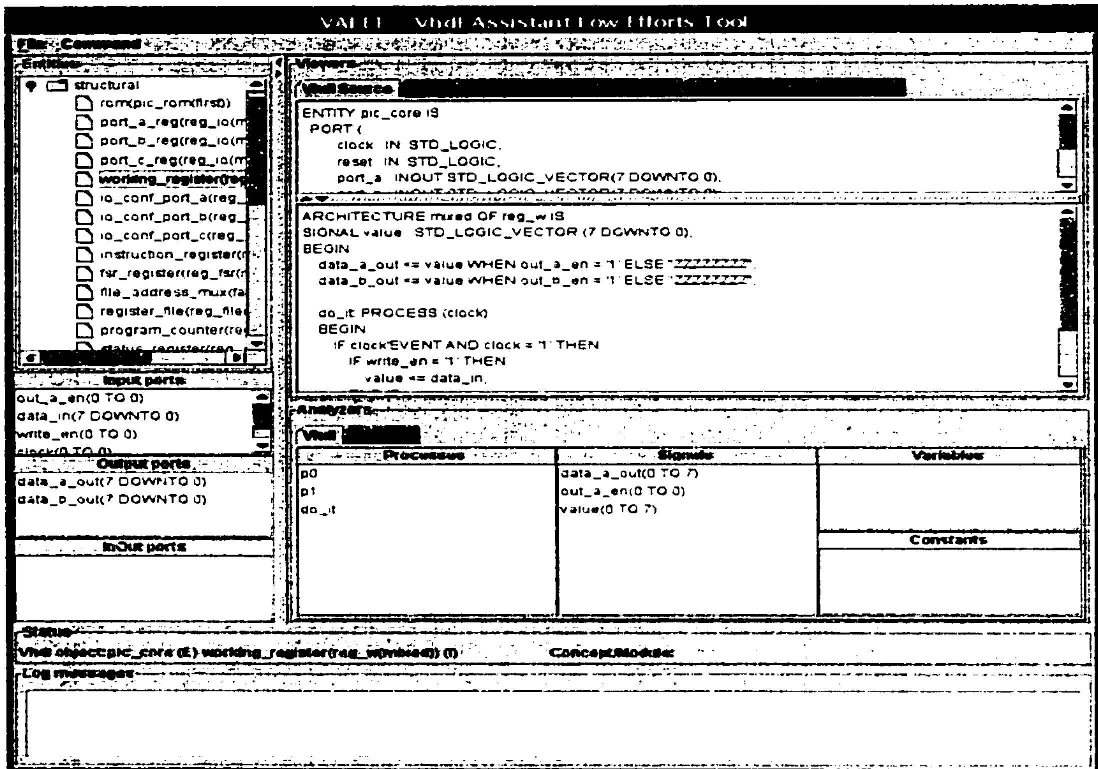


FIGURE 8-2. Navigator

after selecting a component the user generally executes the command *analysis* which is available in the *Command* menu. The command analysis automatically performs signal, combinational and functional partitioning analyses on the selected component.

As results of the analysis, a list of all extracted modules is reported in a dedicated panel named "Modules", and a directed graph representing data dependencies between those extracted modules is also depicted in a dedicated panel named "Modules data dependencies". Each node in the graph corresponds to a module or to an instance of a component which is instantiated inside the selected component. Each directed edge in the graph, from node i to j, is labelled with the name of a signal and indicates that the signal is assigned by the behavior represented by module i and the same signal is also used by the behavior represented by module j. This directed graph representing data dependencies between modules extracted by the functional partitioning analysis gives to a designer a useful global view of the hardware behaviors which compose the behavior of the component. By selecting a module, a designer can inspect the actual behavior interactively using the three viewers available in VALET.

Before describing in detail the capabilities of each viewer, it is important to mention that after loading a design, VALET produces, in a dedicated panel named "Component data-flow dependencies", a directed graph representing data-flow dependencies amongst components of the design. This graph is built if, and only if, the loaded design is composed of more than one component. Each node in the graph represents a component in the design, while each directed edge, from node i to node j, indicates that at least one output signal of the component i is used as an input by the component j. When an edge of this graph is selected, VALET reports the list of output ports and the corresponding input ports which represent all existing connections between two components.

8.2.2 Module viewer

A designer, interested in understanding the behavior represented by an extracted module, can select a module from the list of extracted modules or from the directed graph representing data dependencies between modules. For each selected module, an instance of the Module viewer is opened. A Module viewer consists of a window in which the following elements are reported:

- the type of the module;
- each input and output signal of the module;
- the name of registers embedded in the behavior of the module. No register are reported for the combinational module;
- a directed graph representing data dependencies between abstract concepts which make up the behavior of the module.

By construction, see Section 7.3, the behavior of a module is represented by a set S of related abstract concepts. To show such abstract concepts and their relations, the Module viewer builds a directed graph. For each signal tg which belongs to the set of output signals of at least one abstract concept in S, a node is created in the graph. Each node i contains all abstract concepts in S which have the signal tg_i in their set of output signals.

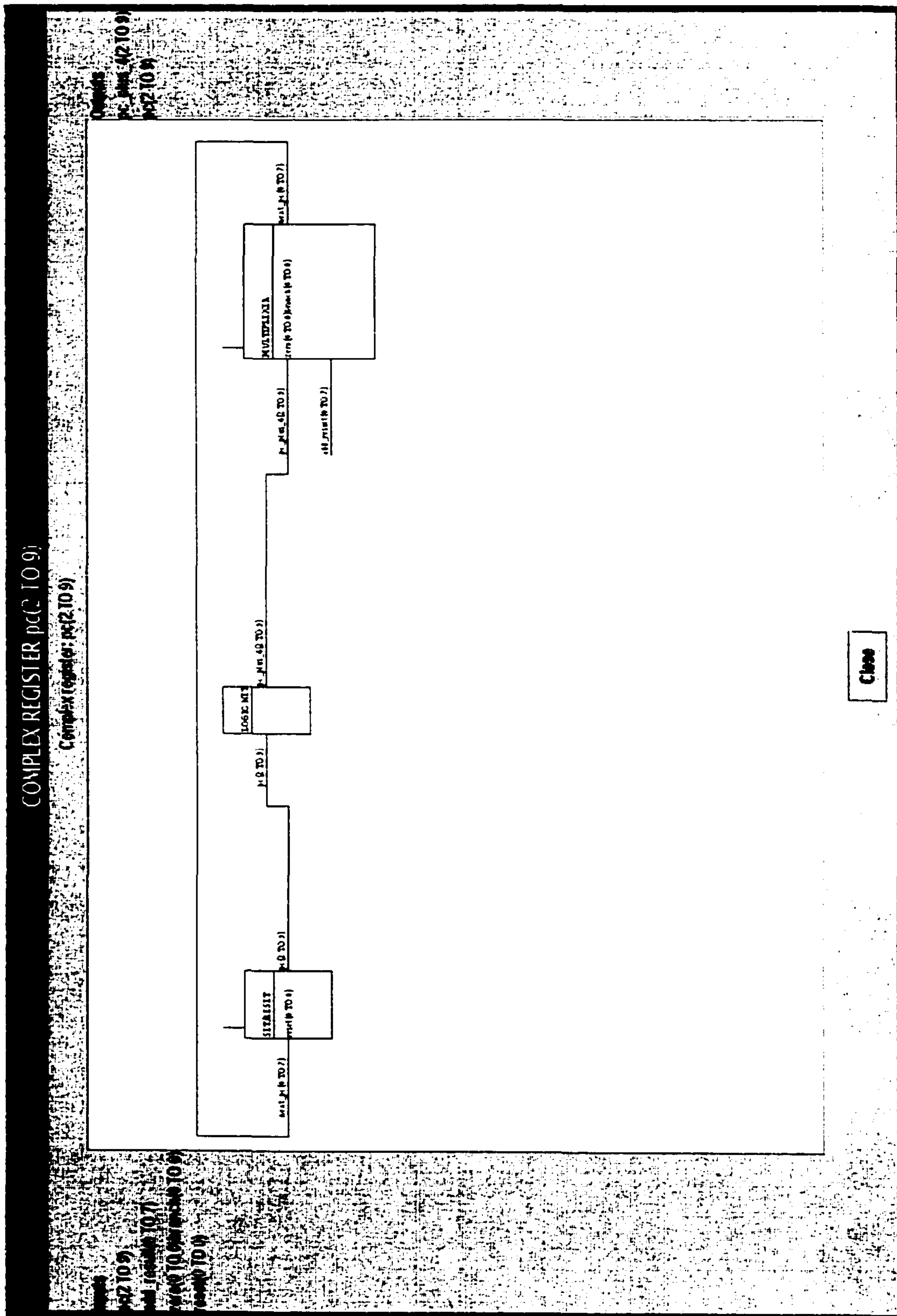


FIGURE 8-3. Example of Module viewer

Each edge, from node i to node j, indicates that the signal tg_i is used by at least a concept of node j. The edge is labelled with the name of the signal tg_i. It was decided to collect abstract concepts based on a signal, because signals are important reference points for a designer. Signals corresponds to wires or memory elements in a synthesized digital component, and generally, designers focus on those elements when examining the functional behavior of a component. For example, during functional testing of a circuit a designer checks that the implemented behavior associated to a specific wire or memory element corresponds to an expected behavior derived from the design's specifications. Since the goal of VALET is to assist the designer in understanding the functional behavior of a component, it seems reasonable to provide a designer with the set of abstract concepts which refer to the same target signal.

The directed graph shown by a Module viewer gives the designer the capability to easily identify relations between abstract concepts which refer to different target signals. If more than one abstract concept exists in a node of the graph, the node is named after the signal represented by the node. Moreover, all names of the input signals belonging to abstract concepts in the node are shown in the node. However, when a node contains only one abstract concept, the node is depicted using the abstract concept type and all its input and output signals names. When a node contains more than one abstract concept, a designer can inspect the behavior associated with the node using an instance of the Target Group viewer.

In Figure 8-3 an instance of a Module viewer is shown. This instance is used to visualize the behavior associated with a complex register module, in which the signal *pc(2* to *9)* represents the register. The graph representing data dependencies between abstract concepts which make up the behavior of the module is shown at the center of the window. A designer can explore the graph by zooming and by selecting an abstract concept to be examined in detail.

8.2.3 Target Group viewer

To inspect the behavior associated to a set of abstract concepts which have a signal *tg* in their sets of output signal, an instance of the Target Group viewer can be used. Each instance of the Target Group viewer consists of a separate window where a user can examine a set of abstract concepts using different features.

First of all, this viewer tries to reduce the number of abstract concepts, which must be examined, by removing abstract concepts whose behavior is embedded in another abstract concept. For example, consider the three abstract concepts depicted in Table 8-1. The expressions e_1 and e_2 correspond to expressions in the original VHDL code which are assigned to target *Z1*. Signal and combinational function analyses create these three concepts because the goal of these analyses is to extract all possible concepts for a given main input *ctrl* and a given target *tg*. That is, these analyses aim to recognize all possible activation conditions for which a behavioral relation exists between *ctrl* and *tg*. However, considering the three concepts in Table 8-1, clearly the concept of type multiplexer provides a more significant information regarding the controlling action of the main input *A* in relation to the target *Z1* than the other two concepts. Since the expressions in all three concepts derive from the same original expression in the VHDL code, the behaviors represented the two concepts of type enabler may be considered to be overridden by the behavior represented by the concept of type multiplexer. In cases like this, the Target Group viewer removes from the starting set of abstract concepts, all abstract concepts which are overridden by another abstract concept. Each abstract concept which covers other concepts is marked as *embedding* to indicate that its behavior embeds other behaviors. The concept which overrides other concepts must have the same main input.

Abstract concept	Target	Main input	Activation Condition	Expression	Mapping
Enabler	Z1	A	B='1' AND C='0'	e_1: D	'0' $\rightarrow e_1$: '1' $\rightarrow \phi$
Enabler	Z1	A	B='0' AND C='1'	e_2: F	'0' $\rightarrow \phi$: '1' $\rightarrow e_2$
Multiplexer	Z1	A	B='1' AND C='1'	e_1 : D : e_2: F	'0' $\rightarrow e_1$: '1' $\rightarrow e_2$

Table 8-1. Embedded abstract concept

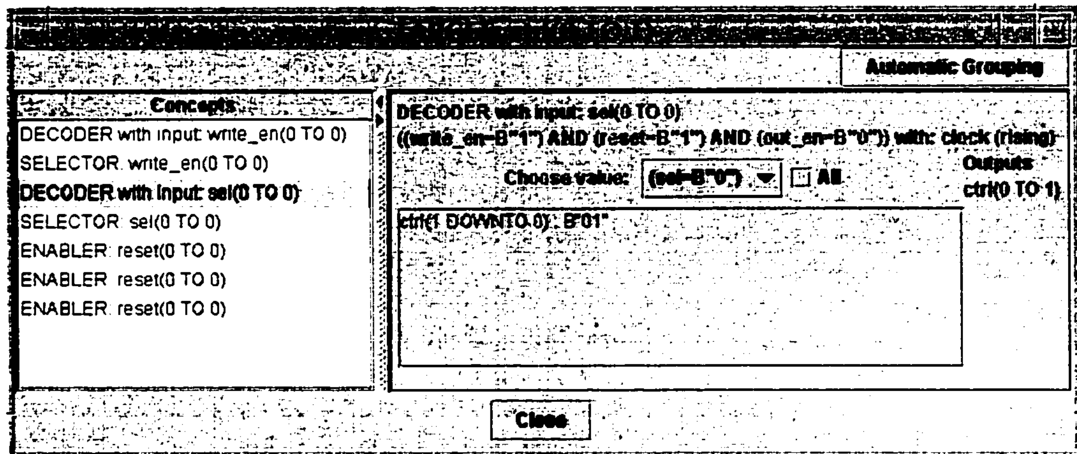


FIGURE 8-4. Example of Target Group viewer with no views

Abstract concepts which contain the same expressions of another concept but which do not have the same main input, are not considered to be overridden. If requested, a user can investigate an abstract concept marked as embedding and view all concepts which are overridden by the concept. In this way the user is able to examine special activation conditions in which the main input behaves differently. Figure 8-4 shows an instance of a Target viewer in which abstract concepts which refer to the same target *ctrl*(*0* to *1*) are listed. In this figure, an abstract concept of type decoder is selected. In the right frame of the Target Group viewer, information regarding the selected abstract concept are reported using an instance of Abstract Concept viewer.

Although concepts, whose behavior is embedded in other abstract concepts, are removed, redundant behavior information may exist in the set *S* of concepts with the same target *tg*. In fact, as reported in Section 7.1, there might exist concepts or groups of concepts which refer to the same set of expressions. These concepts differ from each other on the main input. A designer can inspect each concept in *S* and manually identify those groups of concepts which represent the same behavior. However, the Target Group viewer provides a command, called *automatic grouping*, which creates different collections of subsets of *S*. Each collection represents a different view of the behavior associated to the target signal *tg*. These views are created in the following way:

1. Abstract concepts with the same main input are collected in a view. In fact equivalent sets of concepts always have different main inputs (see Section 7.1 for the definition of equivalent set of concepts).
2. There might be expressions which are assigned to the target *tg* but whose assignment does not depend on a main input *ctrl*. In this case, the view, characterized by the main input *ctrl*, is completed by adding abstract concepts which contain expressions which are not contained in any concepts already existing in the view. In this way, each possible expression, which is assigned to the target *tg* in the examined VHDL code, is contained in at least one abstract concept of every group of concepts representing a view. As a result, each view represents the complete behavior associated to the target *tg*.
3. To ease the inspection of each view, the extra controls analysis (see Section 7.2) is executed in each view. In each view, the extra controls analysis groups the set of abstract concepts with the same main input in a collection of disjoint subsets characterized by extra selectors, extra partial selectors or extra enablers.

After the execution of the command "automatic grouping", a set of views exists. A designer can examine each view in order to understand the behavior associated a target signal. It is important that different views are obtained. In fact, it cannot be known, in advance, which view represents the behavior in terms which better interest a designer. Each view consists of a list of abstract concepts or groups of abstract concepts characterized by an extra controller. By selecting an element in the list, a designer can inspect its associated behavior using an instance of the Abstract Concept viewer. Moreover, the Target Group viewer allows a designer to identify equivalence between elements in different views. In fact, a designer can select one or more element in a view and have the Target Group viewer automatically selects, if possible, elements in other views which represent an equivalent behavior. A designer can switch between views and locate equivalent elements. In Figure 8-5, the three views obtained using the "automatic grouping" command on the example reported in Figure 8-4 are shown. In the first and the second view, the behavior is represented by a single group defined using some extra controls. In the third view the seven original abstract concepts (see Figure 8-4) have been reduced to three

abstract concepts and one group with extra controls. In Figure 8-5, information regarding the selected group or abstract concept are also depicted.

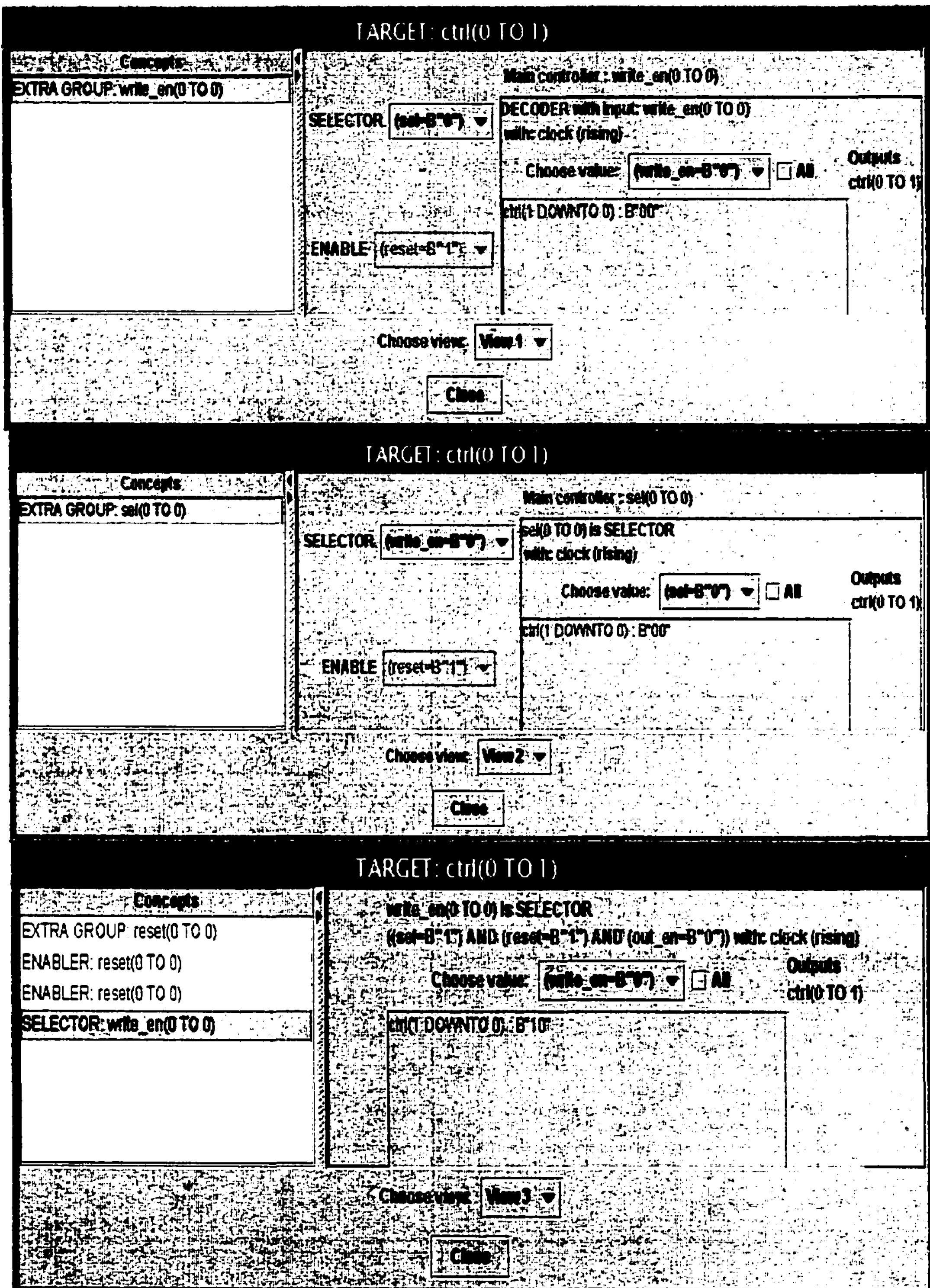


FIGURE 8-5. Example of Target Group viewer with views

8.2.4 Abstract Concept viewer

An instance of the Abstract Concept viewer is used by a designer to investigate, in more detail, the behavior of a specific abstract concept. Each instance of this viewer is specialized for each type of abstract concept. However, every instance of the Abstract Concept viewer consists of a window in which at least the following elements are reported:

- The type of the concept.
- The activation condition, in explicit form.
- All input and output signals.

For each type of abstract concepts, but for the type logic net, the Abstract Concept viewer allows users to select a specific value for the main input and to see what expression is assigned to each output of the abstract concept under consideration. That is, a designer is able to look for details of the behavior of an abstract concept. VALET, automatically, provides a user with a list of values, or range of values, to be chosen for the main input. The list consists of values for which possible different expressions are assigned to the outputs of the considered abstract concept.

An option allows designers to see all possible expressions which are assigned to the outputs of the abstract concepts independently from the value of the main input. An example of interactive use of the Abstract Concept viewer is depicted in Figure 8-6. A concept of type tristate enabler is examined by a designer using an instance of the Abstract Concept viewer. In the upper part of the figure, the designer has selected value zero for the controller *out_en*. The Abstract Concept viewer indicates that high impedance is assigned to all bits of the target signal *data_out*(*0* to *7*). When the designer selects value one for the controller out_en, the Abstract Concept viewer indicates that the expression *value*(*7* downto *0*) is assigned to *data_out*(*7* downto *0*).

When the considered abstract concept is marked as embedding (see Section 8.2.3), the Abstract Concept viewer shows the list of abstract concepts which are overridden by the concept. A designer can select one of these concepts and view its behavior by means of another instance of the Abstract Concept viewer.

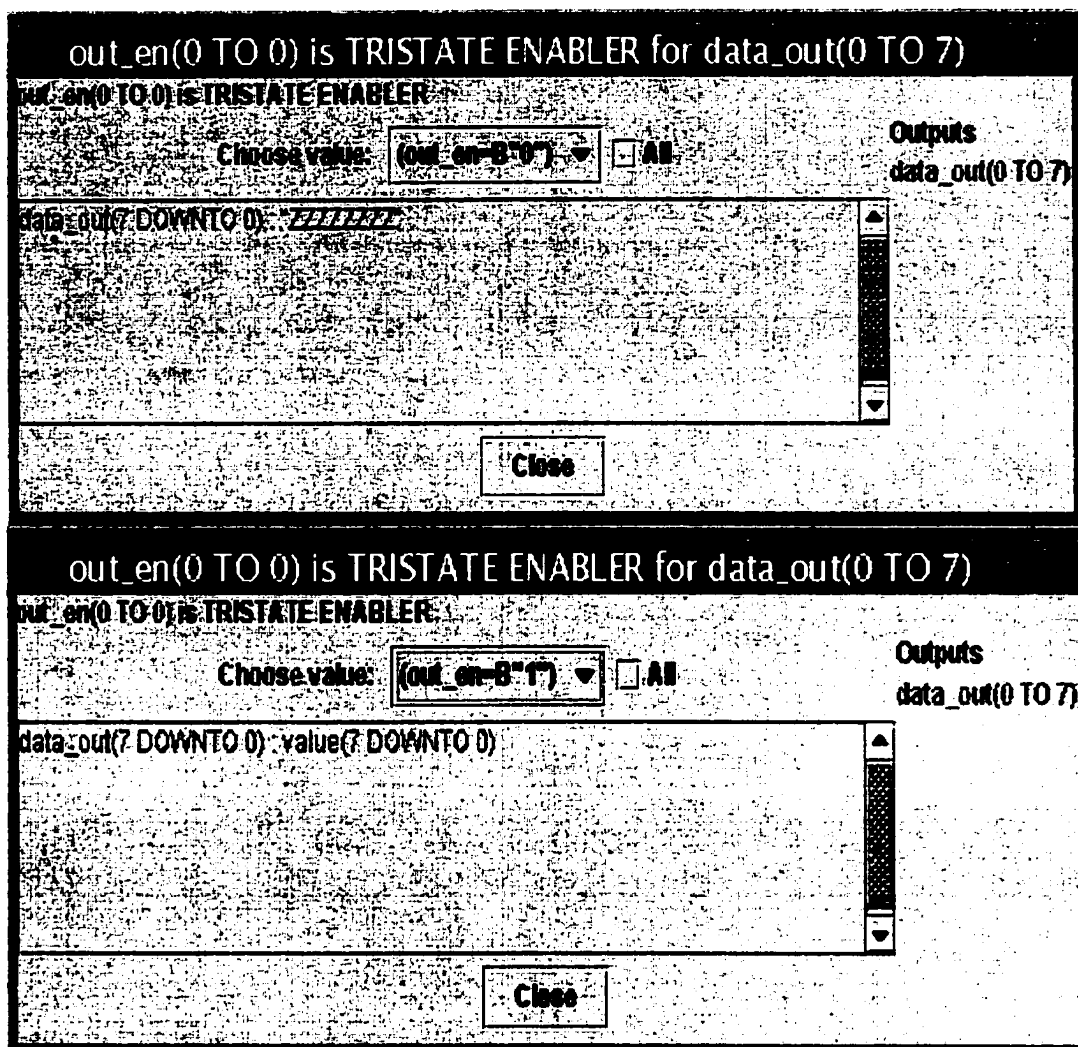


FIGURE 8-6. Example of Abstract Concept viewer

When a designer, using the Target Group viewer, selects an element which consists of a group of abstract concepts with associated extra controllers, the extra controllers are explicitly depicted in the interface of the Abstract concept viewer. A list of values, or range of values, for each extra controller is provided to the user. By choosing a value for each extra controller, a designer can identify the effect of the chosen value on the behavior associated to the outputs of the selected element in relation to values of the main input.

8.2.5 VALET implementation

VALET is a prototype designed to implement and validate the methodology proposed in this dissertation. It was developed using the Java language. VALET consists of more than

50,000 lines of Java code which creates and maintains all required data structures. For efficiency, manipulation of Boolean functions is performed using the Colorado University Decision Diagram package (CUDD) [Som97] which is written in C and was developed by Fabio Somenzi. The CUDD package is integrated into the Java code using Java Native Interface (JNI) technology.

Java Foundation Classes (JFC) and Swing components are used to build the graphical user interface of VALET. In order to avoid introducing further complexity to the already difficult task of exploring artifacts to construct a mental model of the original intent of a design, human-computer interaction (HCI) factors have been considered in designing the user interface. In particular, VALET was designed taking into account some characteristics of human memory and some of the usability paradigms described in [DFAB98].

In a human, sensory memory is used to buffer all received stimuli. By focusing on some of those stimuli, important information is transferred from the sensory memory to the short-term memory. The short-term memory is used to elaborate the information. Eventually, results from the elaboration of information in the short-term memory are transferred to the long-term memory as factual information. Based on the above characteristic of human memory, the following decisions were made in building the VALET tool:

- Instances of Module, Target Group, and Abstract Concept viewers are used to support design exploration activities performed using the short-term memory. For example, the Target Group viewer provides a designer with all information which are needed to understand the behavior associated to a specific target signal. This information is simultaneously available in the form of a list of abstract concepts and a separate instance of Abstract Concept viewer for each of the listed abstract concepts.
- User focus is maintained by visual communications such as the highlighting of the currently analyzed concept in a graph or a list.
- Support to reduce the amount of information needed in the short-term memory while investigating the detailed behavior of an abstract concept is obtained by using pull-

down menus. For example, during analysis of an abstract concept, a pull-down menu provides a list of possible input value to chose from.

- Direct support for the recovery of information from the long-term memory is achieved by means of user annotations which synthesize a previously identified inference. In fact, during the exploration activity, a designer, while building his or her mental model, infers conclusions regarding the analyzed artifact. A designer can annotate each concept, module or group of concepts using textual notes. These notes are saved and can be reviewed by a designer any time during the exploration activity.

Further decisions in designing the user interface of VALET were derived considering some usability paradigms and principles described in [DFAB98]. Each principle indicates qualities that should included in a user interface to improve its usability.

1. The *synthesizability* principle suggests that a user should be able to assess the effect of past operations on the state of the system. Two design decisions were made in VALET according to the above principle:
   - Provide immediate visual feedback on the invocation of any command. For example every item which is selected by a user is highlighted. On invocation of a command, a message is sent on a log window or a status line is updated.
   - Each viewer is immediately updated to reflect the result of previously invoked commands. Available commands are activated or deactivated based on the current state. For example, the capability to explore equivalence between abstract concepts belonging to different groups is activated only after the "automatic grouping" commands is executed.
2. The *observability* principle requires that a user should be able to analyze the state of the system by means of its perceivable representation. It includes the capability to browse the relevant information for a task without affecting the state of the system. Persistency of information is also desirable because it allows a user to manipulate information long after the act of presentation. Based on this principle, the following design decisions were made:

- Information is persistent and provided to a user in the form of visual components like lists and direct graphs. A user can navigate through the information by selecting element and investigating further details if they are available.
- Attention was taken to distribute information amongst different windows and viewers such that as many data as possible which are relevant for a specific exploration task are readily available. For example, the Navigator window is organized in tabbed sub-areas in such a way that data, which are likely to be needed at the same time, are available in different parts of the window. Information about the currently selected component and its interface is permanently available, however by selecting different combination of tabbed sub-areas a user can retrieve data for a specific task. For example if the task consists of understanding how abstract concepts are mapped to HDL code, the list of concepts and the source code can be visualized at the same time.
- In displaying a direct graph, the interface allows users to move the nodes of the graph and customize the visual layout. The customized layout is maintained even if the actual visualization is hidden due to a context switch or to the visualization of a new graph. In this way, when the user recalls the visualization of the first graph, the customized layout is displayed. The persistency of the visual layout makes it easier for users to recall into their memory the mental model associated to the graph representation.

Using an object-oriented language to develop the VALET tool was a good choice. The use of encapsulation, modularity and hierarchy principles, which are promoted by an object-oriented design, facilitate the incremental development of the tool. In fact during development, in order to incorporate new ideas various changes to data representations and analyses were required. Often, such changes merely consist of modifying methods inside a restricted group of classes or of creating a new hierarchy of classes.

Regarding computational performance, the use of Java and the CUDD package has proved to be acceptable. In fact the usage time for VALET to analyze each VHDL components

considered in this dissertation was on the order of seconds on an UltraSparc10 workstation with 128Mbyte of memory.

8.3 Concluding remarks

The capabilities and components of the interactive tool VALET have been presented in this chapter. VALET is the prototype which implements the methodology for understanding legacy HDL code which is proposed in this dissertation. In the next chapter, VALET is used for analyzing some specific test cases.

Chapter 9 Test cases

In this chapter, results obtained by using VALET in two "blind" tests are reported. Two designs are analyzed without any pre-knowledge of their behavior by the author. The results, which are obtained by an inspection of the design using VALET, are compared to the available documentation for the designs. In this way the author is able to evaluate how well the methodology implemented in VALET allows a designer to understand a design whose original intent is unknown. In the first reported test case, examples showing different features, which are available in VALET, are described. In the second test case, a design of a microprocessor is considered and analyzed.

9.1 PIC-16C5X microcontroller

This test case consists of a synthesizable VHDL description of the PIC-16C5X microcontroller developed by Microchip Technology corporation [PIC01]. The actual VHDL description is freely available [UH01]. It was written by E.Romani from the University of Ancona (Universita' degli studi di Ancona), Italy. The design consists of sixteen different components which are instantiated in a top-level component, called *pic_core*. This design has been used as a test benchmark during development of VALET. Each component has been read and analyzed using VALET. In the following sections, descriptions of the use of VALET in analyzing some of the components are presented. Several features available in VALET and its strengths in analyzing VHDL descriptions are illustrated.

9.1.1 Working register (reg_w)

In Figure 9-1, the VHDL description of a component called *reg_w* is depicted. Signal analysis recognizes three signal concepts: two signal concepts of type *tristate enabler* which are associated, respectively, to input port *out_a_en* and *out_b_en*; and one signal concept of type *enabler* which is associated to input port *write_en*. Combinational function analysis does not recognize any combinational function concepts in the design. After functional partition, VALET indicates that the component *reg_w* consists of a simple register module and two combinational modules. In VALET, data dependencies amongst these modules are represented by a graph which is reported in Figure 9-2. Thick lines are used for signals which consists of more than one bit. This graph gives a designer a simple and clear view of the hardware behaviors which made up the component. It shows that the value of the register value is used by two combinational modules.

It is possible to inspect the behavior of each module. Inside the simple register, the previously identified signal concept of type *enabler* represents its behavior. The Abstract Concept viewer reports that when the input port *write_en* is equal to value one, the register,which consists of a signal called *value*, is loaded with the value of input port *data_in*. This behavior is conditioned by the rising edge of the clock signal called *clock*. The behavior of the combinational module *comb1* is represented by one of the previously identified signal concepts of type *tristate enabler*. Using the Abstract Concept viewer, a

```
-- Copyright (C) 1998  Ernesto Romani

LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
ENTITY reg_w IS
  PORT (
      clock      : IN  std_logic;
      out_a_en   : IN  std_logic;
      out_b_en   : IN  std_logic;
      write_en   : IN  std_logic;
      data_in    : IN  std_logic_vector(7 downto 0);
      data_a_out : OUT std_logic_vector(7 downto 0);
      data_b_out : OUT std_logic_vector(7 downto 0)
     );
END reg_w;

ARCHITECTURE mixed OF reg_w IS
SIGNAL value : std_logic_vector (7 downto 0);
BEGIN
  data_a_out <= value WHEN out_a_en = '1'
             ELSE "ZZZZZZZZ";
  data_b_out <= value WHEN out_b_en = '1'
             ELSE "ZZZZZZZZ";

  do_it: PROCESS (clock)
  BEGIN
    IF clock'EVENT AND clock = '1' THEN
      IF write_en = '1' THEN
        value <= data_in;
      END IF;
    END IF;
  END PROCESS;
END mixed
```

FIGURE 9-1. VHDL of the component reg_w

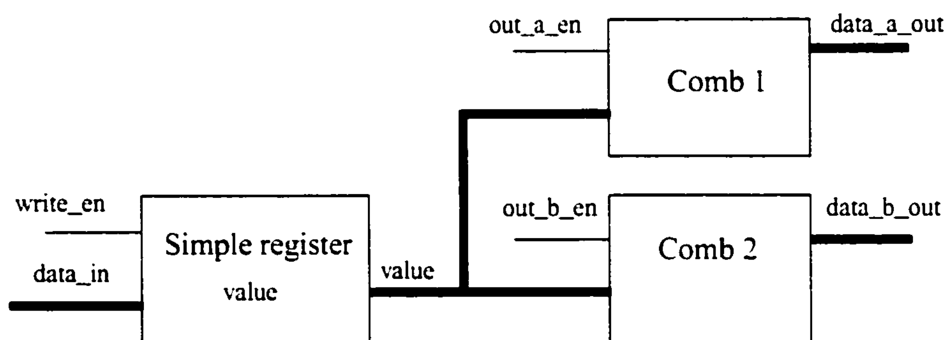


FIGURE 9-2. Data dependencies amongst modules in reg_w

designer can check that when the input port *out_a_en* assumes the value one, the value of the register *value* is assigned to the output port *data_a_out*. However, when the input port *out_a_en* assumes the value zero, high impedance is assigned to the same output port *out_a_en*. The behavior of the combinational module *comb2* is similar. The input port associated with the tristate enabler is *out_a_en* and its output port is *data_b_out*. Althought the example is simple and the behavior of the component could be easily identified by a designer by quickly reading the VHDL description, nevertheless, the results obtained by VALET on this example, indicate an important point of the proposed methodology. In fact, it shows that the methodology is capable of separating behaviors in modules and identifying their data dependencies. Generally the behavior of each module is represented by a set of abstract concepts, in this case only one abstract concept per module. Abstract concepts may be analyzed in detail to identify values of signals which affect the behaviors represented by the abstract concepts.

9.1.2 Status register (reg_s)

A slightly more complex component is *reg_s* whose VHDL description is shown in Figure 9-3. After signal, combinational function, and functional partition analyses, VALET indicates that the behavior of the component *reg_s* is represented by three simple register modules and two combinational modules. All three simple register modules have the same signal *value* as register, however, the simple register 1 indicates that the register *value* consists of 8 bits (7 downto 0), in the simple register 2, the same register *value* consists of a single bit (2), and in the simple register 3, the same register *value* consists of a single bit (0). It is useful to recall that modules represent behaviors and not physical enti-

```
-- Copyright (C) 1998 Ernesto Romani
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
ENTITY reg_s IS
  PORT (
     clock    : IN  std_logic;
     reset    : IN  std_logic;
     out_en   : IN  std_logic;
     write_en : IN  std_logic;
     data_in  : IN  std_logic_vector(7 downto 0);
     data_out : OUT std_logic_vector(7 downto 0);
     carry_out : OUT std_logic;
     carry_in : IN  std_logic;
     zero_in  : IN  std_logic;
     carry_wr : IN  std_logic;
     zero_wr  : IN  std_logic
     );
END reg_s;
```

```
ARCHITECTURE mixed OF reg_s IS
SIGNAL value : std_logic_vector (7 downto 0);
BEGIN
  data_out <= value WHEN out_en = '1'
        ELSE "ZZZZZZZZ";
  carry_out <= value(2);
  do_it: PROCESS (clock, reset)
  BEGIN
    IF reset = '0' THEN
      value <= "00000000";
    ELSIF clock'EVENT AND clock = '1' THEN
      IF write_en = '1' THEN
        value <= data_in;
      ELSE
        IF carry_wr = '1' THEN
          value(2) <= carry_in;
        END IF;
        IF zero_wr = '1' THEN
          value(0) <= zero_in;
        END IF;
      END IF;
    END IF;
  END PROCESS;
END mixed;
```

FIGURE 9-3. VHDL of the component reg_s

ties, therefore it is possible that the same signal is assigned to more than one module. In this example, it means that although all bits of the signal *value* require to be maintained in a register, different behaviors are associated to bits 0 and 2 of such a signal. In Figure 9-4 a graph, generated by VALET, depicting data dependencies amongst identified modules is reported. Clearly, the combinational modules are used to transfer values of the register to some output ports. In particular, investigation of *comb1* shows that its behavior consists of a tristate enabler concept associated to input port *out_en*. The value of register *value* is assigned to the output port *data_out* when the imput port *out_en* assumes the value one, however high impedance is assiged to the output port *data_out* when the input port *out_en* assumes the value zero. Module *comb2* is even simpler, it consists of a logic net concept representing a wiring connection. In this case, the value of bit 2 of the register *value* is always assigned to the output port *carry_out*.

Looking inside the simple register 1, two signal concepts which refer to the same target *value* represent the module behavior. The first signal concept indicates that the input port *write_en* behaves as an enabler. The second signal concept indicates that the input port reset behaves as a set/reset but only when the value of the input port *write_en* is equal to

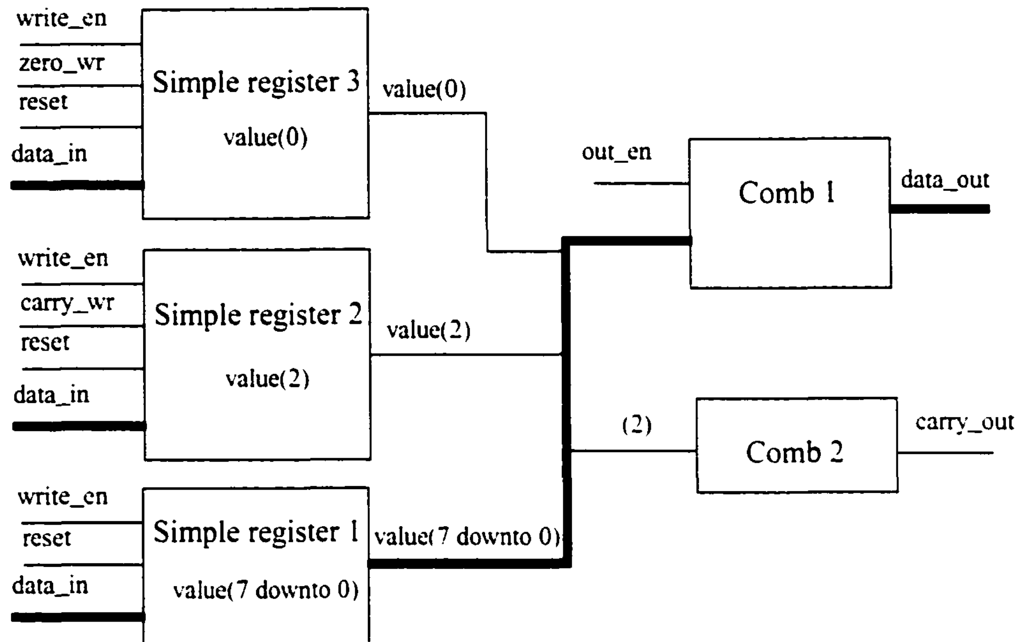


FIGURE 9-4. Data dependencies amongst modules in reg_s

one. Using the command "automatic grouping", available in the Target Group viewer, which invokes the extra controls analysis (see Section 7.2), two different views for the behavior are created. In the first view, the two signal concepts are grouped together with the input port *write_en* classified as extra enabler. In the second view, the two signal concepts are maintained as separate elements. A detailed analysis of the first view shows that the behavior associated with the register value (bits 0 to 7) is mainly controlled by the values of the input port *reset* which acts as set/reset for the register. Particularly, when *reset* is equal to zero, the register is set to value zero, while when *reset* is equal to one, the *data_in* is assigned to the register.

The behavior is subject to the input port *write_en* being equal to one. Looking inside the simple register 2, three abstract concepts which refer to the same target *value* represent the module behavior. The first concept is a combinational function concept of type complete multiplexer in which the input port *write_en* is the selector. The concept is valid for specific values of the input ports *carry_wr* and *reset*. The second concept indicates that the input port *carry_wr* behaves as an enabler under a specific condition which depends on the values of *write_en* and *reset*. The third concept indicates that the input port reset behaves

as set/reset under a specific condition which depends on the values of *write_en* and *carry_wr*.

Again, it is useful to use the Target Group viewer to generate different views which allow the designer to look at the behavior in different ways. Three different views are built. Two views represent the behavior by grouping the three concepts in two groups each, while the third view keeps the three concepts as separate elements. By inspection, one sees that the input ports *write_en* and *reset* are the main controllers in the behavior which is activated when the value of the input port *carry_wr* is equal to one. In particular, input port *write_en* acts as a selector of a multiplexer which assigns the value of the input port *carry_in* or the value of the bit five of *data_in* to bit 2 of the register *value*, depending on the values of *write_en*. When the input port *reset* is equal to zero, bit 2 of the register *value* is set to value zero. Similar results are obtained by investigating simple register 3. In this case, the behavior is activated when the input port *zero_wr* is equal to value one. The input port *write_en* is again the selector of a multiplexer and the input port *reset* acts as a set/reset. The value of the input port *zero_in* or the value of the bit seven of *data_in* is assigned to bit zero of the register *value*.

The analysis of this component shows how useful the capability of looking at a behavior using different views is. In fact examining a view, designers can identify which signals have a dominant effect on the examined behavior of a module. In this example, it is also evident that VALET is capable of identifying behaviors associated to a subset of bits of a signal. Therefore, designers can understand the specific use of a subset of bits in a component.

9.1.3 Register I/O (reg_io)

Another interesting component is *reg_io* whose VHDL description is reported in Figure 9-5. After signal, combinational function, and functional partition analyses, VALET indicates that the behavior of the component *reg_io* is represented by one simple register module and one combinational module. The graph of data dependencies amongst modules, generated by VALET, is depicted in Figure 9-6. The simple register contains two registers

```
ENTITY reg_io IS
  PORT (
      clock, out_en, write_en    : IN   std_logic;
      data_in  : IN   std_logic_vector(7 downto 0);
      data_out : OUT  std_logic_vector(7 downto 0);
      inout_sel : IN   std_logic_vector(7 downto 0);
      dataport  : INOUT std_logic_vector(7 downto 0)
     );
END reg_io;
ARCHITECTURE mixed OF reg_io IS
SIGNAL output_value : STD_LOGIC_vector (7 DOWNTO 0);
SIGNAL input_value  : STD_LOGIC_vector (7 DOWNTO 0);
BEGIN
  dataport(0) <= output_value(0) WHEN inout_sel(0) = '0' ELSE 'Z';
  dataport(1) <= output_value(1) WHEN inout_sel(1) = '0' ELSE 'Z';
  dataport(2) <= output_value(2) WHEN inout_sel(2) = '0' ELSE 'Z';
  dataport(3) <= output_value(3) WHEN inout_sel(3) = '0' ELSE 'Z';
  dataport(4) <= output_value(4) WHEN inout_sel(4) = '0' ELSE 'Z';
  dataport(5) <= output_value(5) WHEN inout_sel(5) = '0' ELSE 'Z';
  dataport(6) <= output_value(6) WHEN inout_sel(6) = '0' ELSE 'Z';
  dataport(7) <= output_value(7) WHEN inout_sel(7) = '0' ELSE 'Z';
  data_out <= input_value WHEN out_en = '1' ELSE "ZZZZZZZZ";

  do_it: PROCESS (clock)
  BEGIN
    IF clock'EVENT AND clock = '1' THEN
      IF write_en = '1' THEN
        output_value <= data_in;
      END IF;
      input_value <= dataport;
    END IF;
  END PROCESS;
END mixed;
```

FIGURE 9-5. VHDL of the component reg_io

input_value and *output_value*. The signal *dataport* is an inout port which is used and set by the simple register module. Since *dataport* is a port of the component, its use is shown outside the module. While the behavior of the combinational module *comb1* is represented by a signal component of type *tristate enabler* with the input port *out_en* as controlling

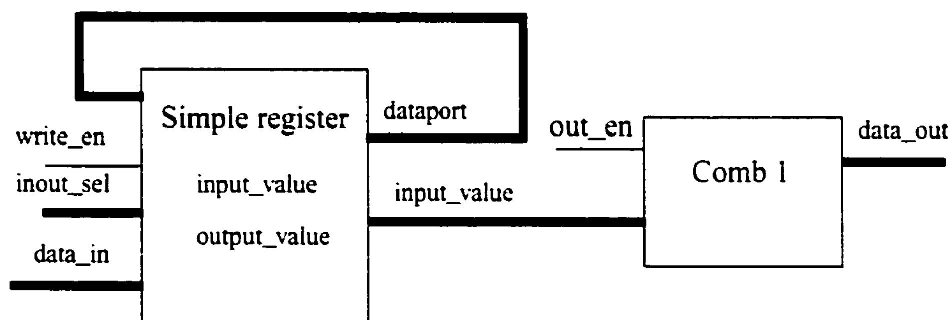


FIGURE 9-6. Data dependencies amongst modules in reg_io

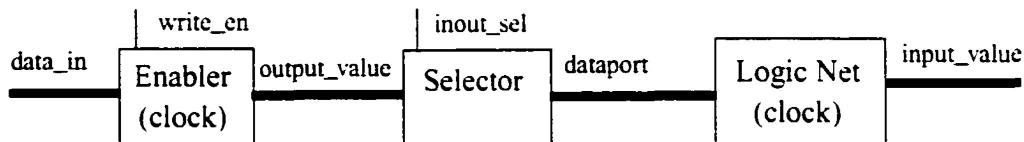


FIGURE 9-7. Data dependencies amongst abstract concept in reg_io

signal, the behavior of the simple register is represented by different abstract concepts. The Module viewer is able to depict the data dependencies amongst abstract concepts representing the behavior of the simple register using a graph (see Figure 9-7). In the graph, the validity of the two concepts *enabler* and *logic net* depends on the input port *clock* which has been identified as a clock. These concepts represent the behavior of registers *output_value* and *input_value*. For register *output_value*, the input port *write_en* acts as an enabler. When the value of *write_en* is equal to one, the value of *data_in* is loaded into the register. The register *input_value* maintains the value of the inout port *data_port* at every clock cycle. In fact the activation condition in the concept *logic net* depends only on the rising edge of the clock. Data are exchanged between the register *output_value* and the register *input_value* based on the behavior represented by the concept *selector*. Examining in detail the behavior represented by the concept selector, depending on the value of the input port *inout_sel*, different bits of the register *output_value* are assigned to the inout port *dataport* and some bits are assigned to high impedance. Using the Abstract Concept viewer, a designer can assign a value to *inout_sel* and examine what values are assigned to each bit of *dataport*.

The analysis of this component shows how the graph representing the data dependencies amongst concepts, which constitute the behavior of a module, assists designers in understanding the behavior of the module.

9.1.4 Final observations

All other components in the PIC-16C5X design have been analyzed using VALET. By inspecting graphs representing data dependencies amongst modules and concepts, and using available viewers, the behavior of each component could be understood.

9.2 MIPS test case

This test case consists of a microprocessor design (MIPS) which is published in [HF00]. MIPS is an example of a modern reduced instruction set computer (RISC). This design has been chosen because its synthesizable VHDL description is freely available, it is well documented, since described in the book, and it consists of a relatively small number of components (five).

In order to test the proposed methodology, the design is inspected using only the capabilities implemented in VALET. During analysis, it is assumed that no previous knowledge exists about the design. Moreover syntactical clues, like signal names, are not considered, except when a component with no code is instantiated (black box). After analyzing each single component of the design, an attempt to extrapolate the functional behavior of the whole design is done. At the end the analysis results are compared with the actual documentation of the design and with the VHDL description.

9.2.1 Control component

The component consists of:

- Input ports: *clock* (1 bit), *opcode* (6 bits) and *reset* (1 bit)
- Output ports: *aluop* (2 bits, bits 0 and 1), *alusrc* (1 bit), *branch* (1 bit), *memread* (1 bit), *memtoreg* (1 bit), *memwrite* (1 bit), *regdst* (1 bit) and *regwrite* (1 bit)

The architecture behavior is represented by two combinational modules and six instances of a component named *lcell*.

- *Comb1*. The first combinational module is driven by six black block instances of type *lcell*. The input of each instance of *lcell* is a bit of the input port *opcode*: its output is connected to a bit of the input *opcode_out* (6 bits) of the module. Since no code exists for the component *lcell* (black box), nothing can be said about it. However, since it has only one input and one output, both one bit wide, the output must depend directly on the input, that is no other signals influence the output value. Looking inside the module,

the behavior is represented by an encoder with input *opcode_out*. The output of the encoder is a one bit wide signal which is connected to the output port *branch* and bit zero of the output port *aluop*.

- *Comb2*. Even the second combinational module has the signal *opcode_out* (6 bits) as input which is driven by the same instances of *lcell* as in the first module. Looking inside the module, three distinct encoders exist. Each encoder has input *opcode_out* and a single bit signal as output. The output of encoder 1 is directly connected to the output ports: *memread* and *memtoreg*. The output of encoder 2 is directly connected to the output port *regdst* and to bit one of the output port *aluop*. The output of encoder 3 is directly connected to the output port *memwrite*. A logic net concept determines the value of the output port *regwrite* using the outputs of encoder 1 and encoder 2. Another logic net concept determines the value of the output port *alusrc* using the outputs of encoder 1 and encoder 3. Looking inside the logic nets, in both cases, the output correspond to the logic OR of the two inputs.

The above exploration is sufficient to declare that the component consists of logic which sets specific constant values to the output ports depending on the value of the input port *opcode*. It can also be observed that the input ports *clock* and *reset* are never used. It results in some output ports carrying the same value, particularly:

- the output port *branch* carries the same value of bit zero of the output port *aluop*
- the output port *memread* carries the same value of the output port *memtoreg*
- the output port *regdst* carries the same value of bit one of the output port *aluop*

The actual values of each output port are not known, however by inspecting each encoder concept in detail, such values could be determined.

9.2.2 Dmemory component

The component consists of:

- Input ports: *address* (8 bits, from 0 to 7), *clock* (1 bit), *memread* (1 bit), *memwrite* (1 bit), *write_data* (8 bits, from 0 to 7) and *reset* (1 bit)

- Output port: *read_data* (8 bits, from 0 to 7)

The architecture behavior is represented by one combinational module and an instance of a component named *lpm_ram_dq*.

- *Comb1*. The output of the combinational module is connected to the input *lpm_write* of the existing instance of the component *lpm_ram_dq*. Since no code exists for the component *lpm_ram_dq* (black box), nothing can be said about it. However, its name and the name of its input and output signals indicate that, probably, the component represents a memory of type RAM (Random Access Memory). Looking inside the combinational module, its behavior is represented by a logic net concept, with the input ports *memwrite* and *clock* as inputs.

Obviously the main behavior of the component is represented by the instance of the component *lpm_ram_dq*. In fact, the input ports *address*, *clock*, *memread* and *write_data* are directly used by such instance, and the output port *read_data* is connected to the output of this instance. The combinational module indicates that a specific relation between input ports *memwrite* and *clock* exists which triggers an operation of the assumed memory. It also results that the input port *reset* is never used.

9.2.3 Execute component

The component consists of:

- Input ports: *aluop* (2 bits, bit 0 and 1), *alusrc* (1 bit), clock (1 bit),*function_opcode* (6 bits, from 0 to 5), *pc_plus_4* (8 bits, from 0 to 7), *read_data_1* (8 bits, from 0 to 7), *read_data_2* (8 bits, from 0 to 7), *reset* (1 bit) and *sign_extend* (8 bits, from 0 to 7).
- Output ports: *alu_result* (8 bits, from 0 to 7), *add_result* (8 bits, from 0 to 7) and *zero* (1 bit)

The architecture behavior is represented by two combinational modules.

- *Comb1*. Looking inside the first combinational module, its behavior is represented by a logic net concept with the input port *sign_extend* and bits from 2 to 7 of input port pc_plus_4 as inputs. The output of the logic net is a nine bit signal of which only the first eight bits are used to set the output port *add_result*. In order to identify the behavior attached to output port *add_result*, it is necessary to look inside the logic net concept. It turns out that the logic net concept performs an arithmetic sum between the two inputs consisting of a different number of bits, one input is composed of 6 bits and the other is composed of 8 bits.

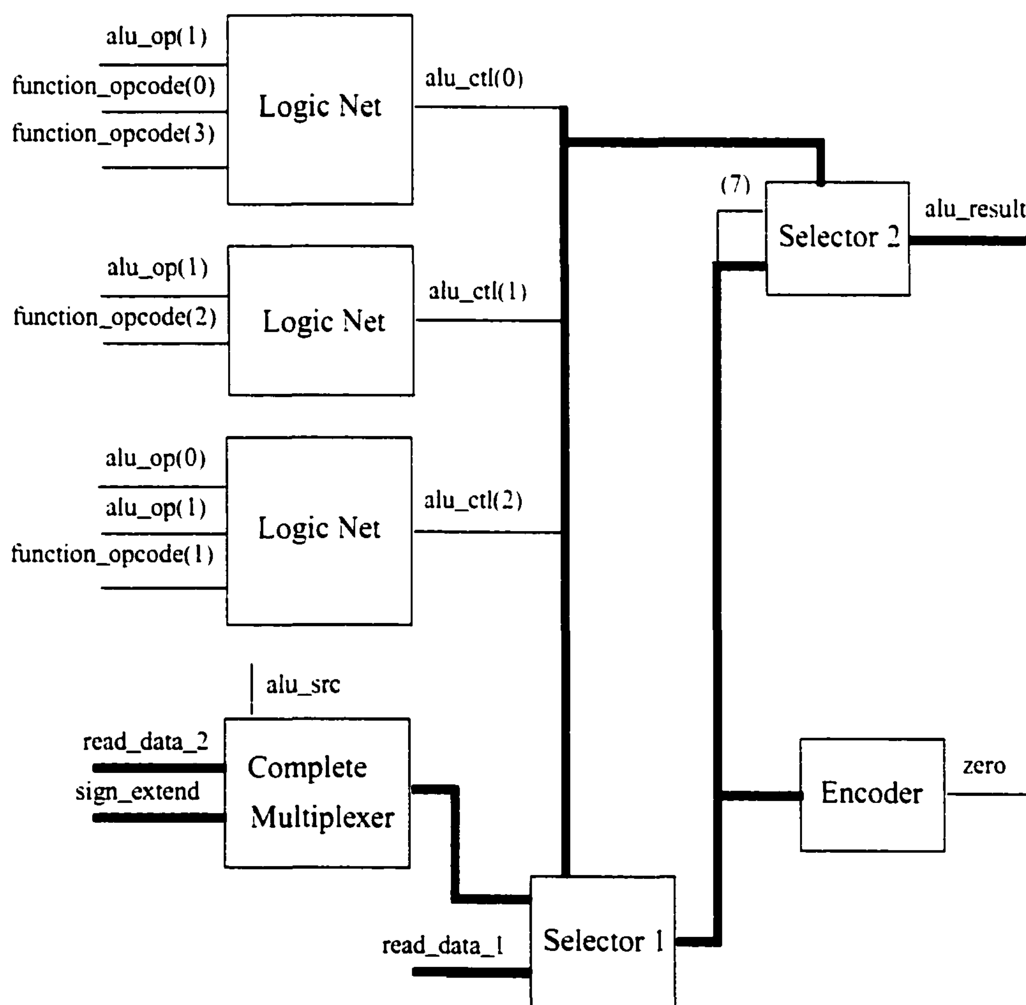


FIGURE 9-8. Comb2 combinational module of execute

- *Comb2*. Looking inside the module seven abstract concepts are used to describe the behavior. The graph, generated by VALET, representing the data dependencies amongst these concepts is shown in Figure 9-8.

By investigating the graph, the following observations can be made:

- An internal signal *alu_ctl* (3 bits) is used. Each of its bits is obtained by three different logic nets which have only input ports as inputs. In particular, different bits of input ports *aluop* and *function_opcode* are used by each of the logic net concepts.
- The input port *alusrc* is only used as a selector of a complete multiplexer. Its selects amongst the input ports *read_data_2* and *sig_extend*.
- The internal signal *alu_ctl* is used as a controller of a selector concept (*selector 1*). The selector concept has the input port *read_data_1* and the output of the complete multiplexer as inputs.
- The output of the selector 1 is a eight bits wide (from 0 to 7) signal and it becomes the input of an encoder whose output is the output port *zero*.
- The internal signal *alu_ctl* is also used as a controller of another selector concept (selector 2). It controls which values are assigned to the output port *alu_result*. The values assigned to *alu_result* depends on the input of the selector, that is on the signal which is the output of the selector, and on bit seven of the same signal.

From this preliminary investigation, it appears that the selector represents the core of the behavior. In fact, its output is used to define values for the output ports *alu_result* and *zero*. Its possible input values are, in pairs, values of input ports *read_data_1*, *read_data_2* or input ports *read_data_1*, *sig_extend* depending on the value of input port *aluscr*. Detailed investigation of the selector shows that its output is equal to the result of an arithmetic or logic operation amongst the pair of inputs depending on the value of *alu_ctl*. The value of *alu_ctl* are determined by values of input ports *alu_op* and *function_opcode*. The value of the output port *zero* is determined by an encoder whose input is the output of the selector.

Detailed investigation of the encoder shows that the output port *zero* is set to constant one when its input is equal to value zero, otherwise is set to constant zero. Looking at selector

2 which is controlled by *alu_ctl* and whose output is the output port *alu_result*, the output of the selector 1 is always assigned to the output port *alu_result*, except when *alu_ctl* is equal to "111". In this last case the output port *alu_result* has all bits equal to constant zero except for bit seven which is equal to bit seven of the output of the selector 1. It also results that the input ports, *clock* and *reset* are never used.

9.2.4 Idecode component

The component consists of:

- Input ports: *alu_result* (8 bits, from 0 and 7), *clock* (1 bit), *instruction* (32 bits, from 0 to 31), *memtoreg* (1 bit), *read_data* (8 bits, from 0 to 7), *regdst* (1 bit), *regwrite* (1 bit) and *reset* (1 bit).
- Output ports: *read_data_1* (8 bits, from 0 to 7), *read_data_2* (8 bits, from 0 to 7) and *sign_extend* (8 bits, from 0 to 7)

The architecture behavior is represented by eight simple registers and five combinational modules.

- *Simple registers*. The eight simple registers are all similar. They differ in the register signal (8 bits). In fact each simple register refers to an element of the array *register_array* which consists of eight registers. In Figure 9-9, the graph generated by VALET with data dependencies of the simple register *register_data(0)* and other modules is depicted. The simple register module has four inputs: the internal signals *write_register_address*,*write_data* and the input ports *reset* and *regwrite*. Looking inside the simple register, its behavior is represented by three abstract concepts which refer to the same target, that is *register_array(0)*. The Target Group viewer is used to explore the behavior which is represented by three concepts. The three concepts identify the internal signal *write_register_address* as an input chooser, the input port *reset* as a set/reset and the input port *regwrite* as an enabler. The internal signal *write_register_address* corresponds to the output of the combinational module *comb1*.

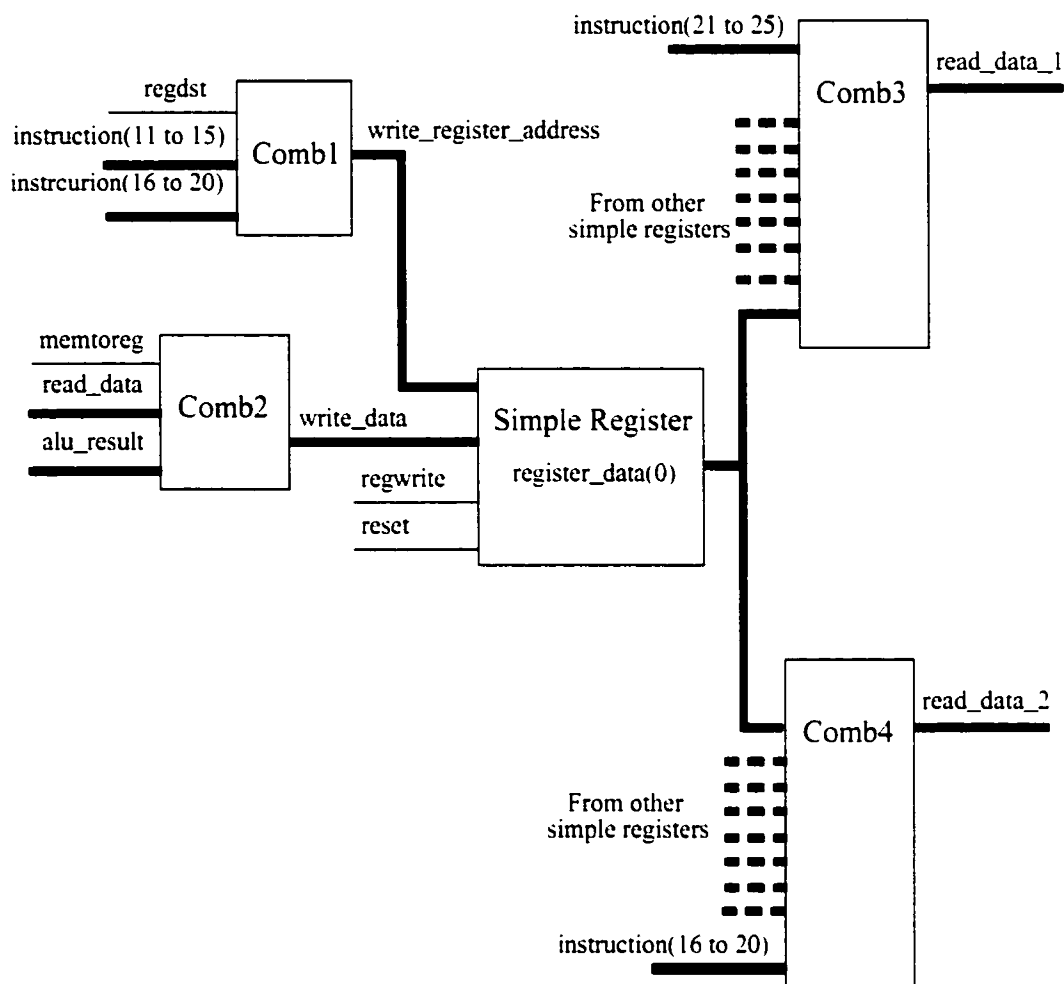


FIGURE 9-9. A simple register and its connections in the component idecode

Those concepts explicitly indicate the purpose of three out of four inputs of the simple register. In order to identify the use of the fourth input a more detailed analysis of these concepts is necessary. Before examining these concepts, the automatic grouping feature, available in the Target Group viewer, is executed. The automatic grouping creates three different views. Two of the views collect the original three concepts in two subgroups based on extra controllers, while the third view is composed by a single subgroup. By looking at the latter view, the input port *regwriter* is an extra enabler and the signal *write_register_address* is an extra partial selector. The extra enabler *regwriter* must be equal to one for the behavior to be valid. When *reset* is equal to one,

register_data(0) is set to a constant zero, while, when *reset* is equal to zero, the input *write_data* is assigned to the *register_data(0)* for three out of 32 (2^5) possible values of *write_register_address*.

For all other values of *write_register_address*, the *register_data(0)* maintains its value. From the above exploration, it can be observed that the simple register, which refers to *register_array(0)*, may be initialized to a value using the input port *reset* and may be load with the internal signal *write_data* for specific values of the internal signal *write_register_address*. However this behavior depends on the input port *regwrite* being set to one. A similar exploration of the simple register, which refers to *register_array(1)*, indicates that the simple register module differs from the previous one in the values of *write_register_address* which assign *write_data* to *register_data(1)*. In this case, four values out of 32 are used to load *write_data* into the *register register_data(1)*. As previously stated, eight simple registers have been identified. Looking at all of them, they behave in the same way except that each one uses different values of *write_register_address* to load *write_data*.

- *Comb1*. Looking inside the combinational module *comb1*, its behavior is represented by a complete multiplexer with input port *regdst* as the selector. The five bits from 11 to 15 and the five bits from 16 to 20 of the port input *instruction* are the input of the multiplexer. The output is the internal signal *write_register_data* (5 bits, from 0 to 4). Depending on the value of *regdst*, one set of bits of the input port instruction or the other set is assigned to the output.

- *Comb2*. Looking inside the combinational module *comb2*, its behavior is represented by a complete multiplexer with *memtoreg* as the selector. The input ports *alu_result*, *read_data* are the inputs of the multiplexer. The output is the internal signal *write_data* (8 bits, from 0 to 7). From the previous analysis of simple registers, it is known that the value of the internal signal *write_data* is eventually loaded into registers. Consequently, this multiplexer concept indicates that the input port *memtoreg* is used to select the input port *alu_result* or the input port *read_data* as the value to be loaded into a register.

- *Comb3*. Looking inside the combinational module *comb3*, its behavior is represented by a complete multiplexer with bits 21 to 25 of the input port *instruction* as the selector. The eight elements of the array *register_data* are the input of the multiplexer. Since the selector *instruction* is 5 bits wide (2^5 values) and there are only eight different inputs (the eight different registers), without analyzing the complete multiplexer in details, it can be said that, for different values of *instructor*, the same *register_data* is assigned to the *read_data_1*.

- *Comb4*. The behavior of the combinational module *comb4* is similar to the previous one (*comp3*). In this case, the selector consists of bits 16 to 20 of the input port *instruction*. The output of the complete multiplexer is the output port *read_data_2*. Again the size of the selector indicates that, for different values, the same *register_data* is assigned to the *read_data_2*.

- *Comb5*. The last combinational module *comb5* has no data dependencies with any of the previous modules. Looking inside the module, it consists of a logic net concept which directly connect the first eight bits (from bit 0 to 7) of input port *instruction* to the output port *sign_extend*. That is, this combinational module is nothing more than a wiring connection.

The above exploration of all existing modules is sufficient to declare that the component consists mainly of eight simple registers which are loaded with values coming from the input ports *alu_result* and *read_data*. Each register may be loaded only if specific values are assigned to input ports *reset* and *reg_write*. Which data is loaded is determined by the input port *memtoreg*. Which register is actually loaded depends on the values of the input port *instruction*. In particular, it depends on the five bits from 11 to 15 or on the five bits from 16 to 20 of it. The input port *regdst* determines which one of the two sets of bits must be used. The value of one of the *register_data* is assigned to the output port *read_data_1* depending on the value of bits 21 to 25 of the input port *instruction*. The value of one of the *register_data* is assigned to the output port *read_data_2* depending on the value of bits 16 to 20 of the input port *instruction*. Moreover the component transfers the first eight bits of the input port *instruction* to the output port *sign_extend*.

Considering the bits of the input port *instruction*, it can be observed that:

- bits nine and ten are never used in the component;
- bits 16 to 20 are used for two purposes: selecting the register to load and selecting which register must be assigned to the output port *read_data_2*. By analyzing the values of instruction in each simple register module and in the combinational module *comb4*, it is observed that for the same value of the input port *instruction* (from bit 16 to 20) the same register is selected by all modules. However value "00000" is an exception, in fact, for this specific value, *register_data(0)* is not selected and loaded with a new value.

9.2.5 Ifetch component

The component consists of:

- Input ports: *alu_result* (8 bits, from 0 and 7), *branch* (1 bit), *clock* (1 bit), *reset* (1 bit) and *zero* (1 bit).
- Output ports: *instruction* (32 bits, from 0 to 31), *pc_out* (8 bits, from 0 to 7) and *pc_plus_4_out* (8 bits, from 0 to 7)

The architecture behavior is represented by one complex register, one simple register and two independent combinational modules.

- *Complex register.* Bits 2 to 9 of an internal signal called *pc*, defined as 10 bits wide, corresponds to the register in the complex register module.This module has data dependencies with all other modules in the component, moreover the register *pc* (bit 2 to 9) is connected to an input of an instance of a component called *lpm_rom*. Since no code exists for the component *lpm_rom* (black box), nothing can be said about it. However, its name and the name of its input and output signals indicate that, probably, the component represents a memory of type ROM (Read Only Memory). Looking inside the complex register module three different abstract concepts exist. The graph representing data dependencies amongst these concepts contains a loop, see Figure 9-10. Focusing

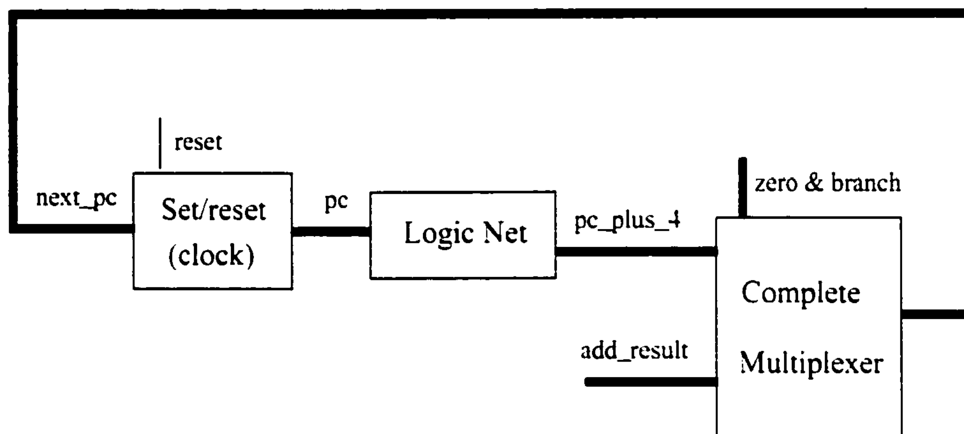


FIGURE 9-10. Complex register of ifetch

on the signal *pc*, the input port *reset* acts as a set/reset on it. In fact a set/reset concept exists which has *reset* as the main input and an internal signal *next_pc* (8 bits) as an input. Since the input port *reset* is one bit wide, it may assume only two values.

For one value, it sets the signal *pc* to a constant, for the other value it sets the signal *pc* to a new value which depend on the value of the signal *next_pc*. The value of *next_pc* is obtained through a complete multiplexer behavior with input selector equals to the concatenation of input ports *zero* and *branch*. The value of *next_pc* may be equal to the input port *add_result* or to internal signal *pc_plus_4* (10 bits) whose value depends on previous value of *pc* through a logic net. To have a better understanding, looking at the logic net concept in detail, bits 2 to 9 of the internal signal *pc_plus* are set equal to the register *pc* plus value one. From the above exploration, it can be observed that the register *pc*, which is composed of 8 bits, can be set to a constant value, to a value which is equal to its current value plus one, or to the value of the input port *add_result* depending on the values of *reset*, *zero* and *branch*. Moreover its current value and its value plus one, which corresponds to internal signal *pc_plus_4*, are used externally by some other behavior.

- *Simple register.* The internal signal *pc* is the register of the simple register module. Looking inside the module, it consists of a single concept in which the input port *reset* acts as an enabler. In details, when *reset* is equal to one, *pc* is assigned to constant value

zero, otherwise its value is not changed. The behavior represented by this module underlines that all bits of the internal signal *pc* are set to zero under control of input port *reset*.

Before proceeding to inspect the other modules, a few observations must be made. In VALET, modules represent behaviors and not physical entities, therefore it is possible that the same signal is assigned in more than one module. For example, in this component, the internal signal *pc* is set in both complex register and simple register modules. Each module depicts a specific behavior for the signal. In the complex register module, only 8 bits of signal *pc* are set, while in the simple register module all 10 bits are set. Since the synthesis process produces a single logic element for a VHDL signal, considering the two modules, it can be stated that signal *pc* represents a memory element. All bits of the register *pc* may be set to zero by the input port *reset*. Eight of its bits, from bit 2 to 9, are set based on values of input ports: *reset*, *zero*, *branch*, and *add_result*. Since the input port *reset* is an input in both modules, it is necessary to look to its use in detail. Looking back to the complex register, the input port *reset* acts as a set/reset behavior and particularly when it is equal to one it sets the eight bits of *pc* to value zero, while when it is equal to zero it assigns the current value of signal *next_pc* to *pc*. This behavior of *reset* is consistent in both modules, as it should have been expected, since the component is a working design.

- *Comb1*. The combinational module *comb1* consists of a wiring connection in which bits 0 to 7 of register *pc* are assigned to the output port *pc_out*.
- *Comb2*. Also the second combinational module *comb2* corresponds mainly to a wiring connection. In this case, bits 0 to 1 of the output port *pc_plus_4_out* receive a constant value, while bits 2 to 7 receive bits 2 to 7 of internal signal *pc_plus4*. From previous analysis, it is known that these bits of the internal signal *pc_plus_4* are set by the complex register module.

After exploring the modules, it is evident that the internal signal *pc* and *pc_plus_4* have an important role in the global behavior of the component. Moreover it seems that subsets of their bits are used in different exceptions. The first 8 bits of signal *pc*, from bit 0 to 7, are connected to the output port *pc_out*. Out of this eight bits, the first two, from bit 0 to 1, are

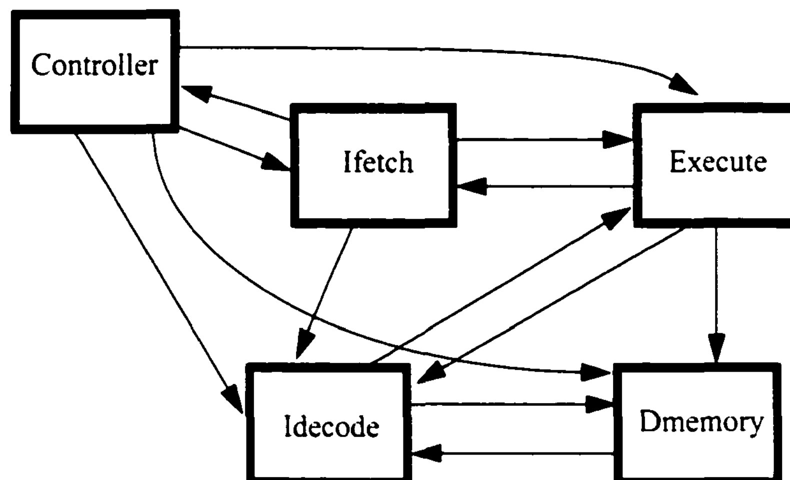


FIGURE 9-11. Data-flow dependencies amongst components of MIPS

never changed by the component except to be set to value zero when the input port *reset* is equal to one. The last 8 bits of *pc*, from bit 2 to 9, represent a register and they are connected to an input, called *address*, of the instance which represents a ROM memory. The output of the instance is connected to the output port *instruction*. Assuming a standard behavior for the ROM, each value of *pc* causes a memory value to be read and assigned to the the output port *instruction*. The last 8 bits of *pc_plus_4*, from bit 2 to 9, corresponds to the eight bits of register *pc*, from bit 2 to 9, plus one. Six of these bits, from bit 2 to 7, are concatenated to a 2 bits wide constant value. The value after the concatenation is assigned to the output port *pc_plus_4_out* with the constant values representing the least two significant bit of the output port *pc_plus_4_out*.

9.2.6 Observations about the MIPS design

VALET does not provide any novel capability to assist designers in analyzing functional relationships amongst components of a design. However, it builds a direct graph in which data-flow relationships amongst components are depicted. In Figure 9-11, the automatically generated graph of data-flow dependencies amongst components of the MIPS design is shown. Information about which signals of a component are used in another component, is associated to each edge in the graph. Considering this graph and the results of above explorations of each component, the following conclusions about relationships amongst components of the MIPS design can be drawn:

- The component *controller* generates signals which are used to activate specific activity inside each component. In fact all its output signals are mainly used as selectors or controllers in concepts recognized inside each component.
- The signal *instruction* generated by the component *ifetch* is used by all but the component *dmemory*. Sets of its bit contribute to different activities in each of the component. Bits 26 to 31 become the input signal *opcode* of the component *controller*. Bits 0 to 5 are used in the component *execute* as part of the logic that selects which arithmetic or logic operation is performed. Bits 11 to 25 are used in the component *idecode* to select which register must be loaded and also which register value must be transfered to the outputs of the component.
- The signal *pc_plus_4* generated by the component *ifetch* is used by the component *execute*. Its value is an operand of an arithmetical sum performed by the latter component.
- The results of each operation performed inside the component *execute*, are transferred to the components *ifetch*, *idecode* and *dmemory*. Both components *idecode* and *ifetch*, eventually, save these results into their registers, while component *dmemory* uses one of the result to address its internal memory.
- The outputs of the component *idecode* are used as operands of arithmetic and logic operations inside the component *execute*. The output *read_data_2* is also saved in the memory inside the component *dmemory*.
- From the above observation, one can see that data which are transferred to *idecode* from *execute* are transferred again back to *execute*. Similarly, data which are sent to *dmemory* from the component *idecode* are also reused by the component *idecode*.

Summarizing, the component *execute* seems to perform the processing activity in the design. The components *idecode* and *dmemory* are the repository of data. Both components *controller* and *idecode* generate signals which are used by other components to decide which activities to perform.

9.2.7 Comparing results of analysis with actual documentation

The results obtained while exploring each component using VALET are compared to the documentation available in the book [HF00]. The documentation consists of block diagrams for each component, of a description of the functionality of each component and of the actual VHDL code. Discrepancies as well as agreements, for each component, are reported in the following list.

1. The behavior of the component *control* is correctly recognized using VALET. Moreover, VALET identifies that some of the output signals carry the same value. Such information is not explicitly available in the documentation. Only by inspecting lines in the VHDL code, a designer could identify these relationships amongst output signals. The documentation explains the use of the instances of component *lcell*. These instances do not carry any functional behavior, they are included in the design to allow the *MAX-PLUS* tool, developed by Altera corporation, to correctly perform logic minimization and synthesis. Of course, such information could never be recovered by VALET.

2. The component *dmemory* is very simple. The documentation confirms that the conclusion that the component *lpm_ram_dq* represents a RAM, based on its name, is correct.

3. Comparing the block diagram of component *execute* in the documentation, with the data dependence graphs amongst modules and amongst abstract concepts obtained in VALET, the following observations can be made:
   - The fact that the value of the output *add_result* and the values of the outputs *zero* and *alu_result* are computed independently, is explicit in both representation.
   - The graph with data dependencies amongst abstract concepts is more detailed than the block diagram. It provides more information than the latter one. Moreover, the block diagram is slightly incorrect, in fact it does not show that the selection of the operation to perform depends also on the input *function_opcode* and not only on the input *aluop*.

- The block diagram indicates that the input *sign_extend* is shifted left before being arithmetically added to the input *plus_4*. This information is completely missed in VALET analysis. In VALET, the difference on the bit size of the two operands is identified, but no shift operation is inferred. This inference can be done by a designer using knowledge on logical and arithmetic operation in the Boolean domain.

4. Even if the VHDL description of the component *idecode* consists of few lines, for a designer it is not easy to extrapolate the precise behavior of the component by reading it. In fact, the VHDL code contains a *for* loop constructor and an array which represents memory elements. This array is indexed by another signal in the component. This component represents a good example of how VALET makes analysis easier for a designer. In fact, not only does VALET recognize the correct behavior but also, differently from the block diagram, explicitly identifies the number of registers existing in the component. Moreover, in this example, VALET locates an error in the VHDL code of the component *idecode*. On page 196 of the book, it is stated that "register 0 always contains value 0". However, VALET identifies that there are three values of signal *write_register_address* for which a new value is loaded into register zero. Direct investigation of the VHDL code indicates that the VHDL lines, reported in Figure 9-12, contains the error. The reason is that the conditional expression uses all bits of signal *write_register_address* (5 bits), while the register is indexed using only three bits, from 0 to 2, of the same signal.

   Therefore, the actual behavior is different from the intended one which is also indicated by the comment in the source code. I would like to propose an explanation for the existence of the error. In the book it is stated that, in order to make the synthesis and logic optimization processes faster, the VHDL code describes a reduced version of the original MIPS microprocessor. The modification consists of having reduced the number of bits in the data path from 32 to 8, which requires also to reduce the number

```
        -- Write back to register - don't write to register 0
ELSIF RegWrite = '1' AND write_register_address /= 0 THEN
 register_array( CONV_INTEGER( write_register_address( 2 DOWNTO 0 ))) <=write_data:
```

FIGURE 9-12. Error in VHDL code of the component idecode

of registers from 32 to 8. It is my opinion that the author of the book forgot to modify the conditional expression reported in Figure 9-12 which should look like the following:

ELSIF RegWrite='1' AND write_register_adddress(2 DOWNTO 0) /= 0 THEN

5. The basic behavior of the component *ifetch* is correctly recognized by VALET. The block diagram of this component, available in the book, and the data dependence graphs shown by VALET are equivalent. However, VALET is not able to recognize that the arithmetic operation performed when calculating the value of the output *plus_4* corresponds to the sum of value four. As for the case of the shift operation in the component *execute*, VALET assumes that a designer could make such inference based on his or her knowledge of arithmetic operation in the Boolean domain.

The results obtained in analyzing each component in the MIPS design using VALET are quite encouraging. In fact VALET correctly identifies the behavior of each component. Graphs reporting data dependencies relations amongst abstract concepts and modules are comparable to block diagrams which are often used by designers. In VALET, both high level information and detailed information are depicted using modules and abstract concepts. The detailed information is useful to explore the use of each signal. In some cases, this information might allow designer to identify incorrect behavior.

Regarding understanding of the intent of a design which is composed by multiple components, VALET gives little automated support to designers. However, using the graphs which report data flow information amongst components and results of analysis on each single component, a good general understanding of the activities performed by every component in relation to each other seems possible. In fact, the observations reported in Section 9.2.6 on the design MIPS are consistent with the documentation of the MIPS design.

Chapter 10 Conclusions

After describing an alternative approach to the problem of capturing the design intent of a HDL specification, major and minor contributions of this dissertation are reported. Future research work is described in the last section of this chapter.

10.1 An alternative approach

In December 1999, a company named Escalade released a tool called *Design Extractor* whose goal, as stated in the Design Extractor tutorial manual [Esc99], is "Apply synthesis-based technology to capture the design intent of legacy RTL code as well as raising its abstraction to a language-independent level". By contacting the company, it was found that the tutorial manual was the only available source of information regarding the tool. Not many technical details are described in this manual, however the general approach and intent of the tool are presented.

The Design Extractor tool can examine Verilog specifications. It is composed by two parts, the extractor itself and a *DesignBook* which contains visual representations, *i.e.* state diagrams, truth/state tables, flow charts, language blocks or middleware modules. Under control of the user, the extractor converts the original design into an aggregation of the visual representations available in the DesignBook. The extraction process is performed in two phases: transformation and mapping. During the transformation phase the original HDL code is analyzed and by applying available transformations it is converted to an equivalent HDL code which conforms to predefined coding styles. Design Extractor uses

coding styles which are recommended by vendors of synthesis tools, particularly Mentor Graphics Corporation and Synopsys Inc.

During analysis the code is partitioned into combinational and sequential logic, that is each concurrent statement and process is classified as belonging to one of the two types of logic. Some transformations are done automatically, while other transformations can be activated by a designer on selected parts of the HDL code. After the transformation phase has been executed, the new HDL code is mapped to visual elements of the DesignBook. Sequential statements are mapped into finite state machine representation or truth/state tables, while combinational statements are mapped to continuous assignment language blocks or middleware modules. The tutorial manual does not specify which middleware elements are recognized.

Logic gates (AND,OR,NOT,XOR ..) with different fan-in; decoders, encoders and multiplexers of different input size; adders, shifters, incrementers, decrementers, comparator modules; basic register types (D, JK, T ...); counters, parallel and serial shift registers, stacks, memory modules are available in the middleware library. As stated in the manual, not all these elements are recognized, in fact there are cases in which parts of the RTL code cannot be mapped to any elements and these parts are treated as HDL fragments. As a result of the transformation and mapping phases, a new view of the original HDL specification is provided to a designer. This view is equivalent to the synthesized design. It consists of language independent representations.

10.1.1 Comparing Design Extractor to the proposed methodology

Both the Design Extractor tool and the methodology presented in this dissertation attempt to assist designers in understanding the intent of a legacy design. However, the approach followed by Design Extractor is different from the approach presented in this dissertation. Design extractor is based on software transformations which try to convert the original HDL code to a new HDL code using predefined synthesis code templates. When successful, these transformations allow the mapping phase to uniquely identify hardware behaviors which will be synthesized from the new HDL code.

The approach in this dissertation aims to recognize functional behaviors inside the design. Those behaviors may overlap or not, and they may not directly exist as separate components in the final synthesized design. For example, in analyzing the status register component in Section 9.1.2, our methodology recognizes that the input port *write_en* behaves as a selector of a multiplexer which assigns the value of the input port *carry_in* or the value of the bit five of *data_in* to bit 2 of the register *value*. The recognized behavior is conditioned by having the input port *carry_wr*='1'. This conditioned behavior cannot be recognized by Design Extractor which attempts to map HDL into predefined modules whose behavior is not conditioned by any value of signals.

Generally, the Design Extractor performs the mapping phase on each Verilog task separately. That is, behaviors across multiple tasks are not recognized. However, designers can manually select and apply transformations on the original HDL code to remove boundaries between tasks. In this way, a designer can guide the Design Extractor to recognize behaviors across multiple tasks using the newly obtained Verilog code. In this dissertation, by merging signal concepts and by analyzing data dependencies between abstract concepts, our approach is able to automatically identify behaviors across multiple VHDL processes (or multiple Verilog tasks).

Different from Design Extractor which produces one interpretation for a given HDL code and a set of applied transformations, our proposed methodology aims to help designers in getting a wider understanding of the overall design function by providing different views of the same part of a HDL specification. Each view represents a different way to interpret the same part of a HDL code. Therefore, each interpretation could better suit specific tasks which are addressed by a designer.

In May 2000, Escalade was acquired by Mentor Graphics and Design Extractor was removed from the market.

10.2 Contributions

The main contribution of this dissertation is to show that it is possible to extract the intent of a legacy design directly from its HDL code as a collection of abstract artifacts. These abstract artifacts, representing functional behaviors in the digital logic domain, are examined by a designer to understand what a legacy component does.

Previous approaches either aim to help the designer in understanding a design by facilitating the reading of HDL code itself but not what it does, or they limit their capabilities of extracting design intent by using synthesizable objects as abstractions (see the Escalade approach Section 10.1). By extracting behaviors which may or may not exist as separate components in the final synthesized design, the methodology in this dissertation provides designers the capability of investigating the original design intent in more abstract terms and in different ways.

Summarizing, the major contributions of this dissertation are:

- A novel set of concepts, as abstract artifacts, which are characterized by an activation condition. In the software engineering area, an artifact, which is extracted by analyzing source code, represents a static property existing in the code. Signal and combinational concepts, which are artifacts extracted by the methodology described in this dissertation, represent functional behaviors in the digital logic domain which are embedded in a design description but which are valid only when specific conditions occur. These concepts are used to represent multiple interpretations of the same part of a design.
- Provide various level of abstractions which allow designers to inspect the design intent at different levels of detail. In the proposed methodology, a design is partitioned into modules which indicates to a designer the general hardware behavior of the design. By looking at the graph representing data dependencies between modules, a designer can identify the main functional parts of a design. General information regarding the behavior of a module are provided by a graph representing data dependencies between abstract concepts which make-up the behavior of the module. More detailed information is provided to a designer by the content associated to each abstract concept. The

capability of providing information at different levels of abstraction is quite important since it allows a designer to stop the investigation when he or she is satisfied with the understanding of a design.

- Extract functional behaviors across multiple processes. Previous research work analyzes HDL descriptions considering each VHDL process (or Verilog task) as a separate and independent unit. In the proposed methodology, concepts and modules are extracted looking at information and relations amongst multiple processes.
- Develop an interactive tool which supports the proposed methodology. Since information exploration is an important activity in any reverse-engineering process, the software tool VALET is an integral part of the methodology. The structure and components of VALET has been defined taking into account assumptions regarding the way a designer might proceed in understanding a design. Although, in this dissertation, no specific studies have been undertaken to validate the assumptions, the availability of VALET creates opportunities for future studies (see Section 10.3).

In the following, some further contributions of this dissertation are reported:

- Definition of a new language independent representation on which analyses are performed. By defining and using the Finer Program Dependence Graph, the proposed methodology can be applied on any synthesizable specification independently from the hardware description language which was used to describe a legacy design.
- Exploitation of a technique, described in [CK96], to represent every conditional expression existing in the original HDL code using a minimum set of Boolean variables. In [CK96] a technique to represent relational expressions, in which a signal is compared to a constant value, by means of a minimal set of Boolean variables, is described. This dissertation expands the technique to handle also comparison between signals and composite expressions.

The work in this dissertation focuses on understanding HDL specifications to help designers to reuse previously designed components. However, the VALET tool, which implements the proposed methodology, can also be used as a forward engineering tool. In fact

VALET may support designers during the design of a new component. After writing some HDL code to specify part of a new design, a designer may use the capability of VALET to extract the different functions which are embedded in the HDL code. In this way a designer can check if the code actually describes what was intended and any not intentional behavior has been erroneously introduced.

10.3 Future work

In the methodology presented in this dissertation, there is no automated support on analyzing functional relations amongst components of a design. A major future contribution would be to introduce a new analysis which aims to identify functional behaviors across borders of components. Similarly to the merging of signal concepts, this new analysis could merge abstract concepts which are identified in different components.

Sometimes, assignment expressions which are grouped into a combinational module by the methodology in this dissertation, may represent a complex combinational behavior. This complex combinational behavior may or may not be mentally mapped by a designer to a commonly known combinational behavior. Analyzing these behaviors corresponds to analyzing a network of logic gates. Since this dissertation focuses on behaviors described using high level constructors of HDL languages, no specific analysis have been developed for those cases. However, techniques to extract functional representations from a given gate-level representation exist and are surveyed in [CELV99]. It would be useful to integrate these techniques into our methodology.

In this dissertation, only techniques based on static analysis have been considered. By using techniques based on dynamic analysis, further support in understanding the behavior represented by an extracted module could be provided to designers. For example, simulation techniques could be used to inspect the behavior associated with a fsm module. A designer could assign a value to the state signal and examine the actual evolution of the behavior of the fsm module during simulation.

Lastly, but not the least important, it would be useful to test the effectiveness of the methodology and the usability of VALET with designers in a company. In fact, in this dissertation, the validity test of the proposed methodology has been performed by the author. Feedback from different designers could be particularly useful to validate the assumptions regarding the way designers proceed in understanding a design on which the VALET interface was developed.

Bibliography

[All96] Virtual Socket Interface Alliance, "Architecture document," http://www.vsi.org, Technical Report, 1996.

[AOS94] J. Altmeyer, S. Ohnsorge, and B. Schurmann, "Reuse of design objects in cad frameworks," in *International Conference on Computer Aided Design*, pp. 754-761, 1994.

[Arn93] R. S. Arnold, *Software Reengineering*. IEEE Computer Society Press, Los Alamitos, California, 1993.

[BBS96] L. Baresi, C. Bolchini, and D. Sciuto, "Software methodologies for VHDL code static analysis based on flow graphs," in *Proceedings European Design Automation Conference*, pp. 406-411,1996.

[Bha97] J. Bhasker, *A Verilog HDL Primer*. Star Galaxy Press, 1997.

[BL91] P. T. Breuer and K. Lano, "Creating specifications from code: Reverse-engineering techniques," in *Software Maintainance: Research and Practice*, vol. 3, pp. 145-162, 1991.

[BMW93] T. J. Biggerstaff, B. G. Mitbander and D. Webster, "The concept assignment problem in program understanding," in *Proceedings of the Int. Conference on Software Engineering*, pp. 482-498, Baltimore, 1993.

[Bry86] R. E. Bryant, "Graph-Based algorithms for Boolean Function Manipulation," in *IEEE Transactions on Computers*, pp. 677-690, 1986.

[BR97] I. Burnstein and K. Roberson, "Automated chunking to support program comprehension," in *International Workshop on Program Comphrension*, pp. 40-49, 1997.

[BR99] C. Barnas and W. Rosenstiel, "Object-oriented reuse methodology for VHDL," in *Proceedings Design, Automation and Test in Europe*, pp. 689-693, 1999.

[BRSW87] R. Brayton, R. Rudell, A.Sangiovanni-Vincentelli, and A. Wang, "MIS: A multiple-level logic optimization system," in *IEEE Trans. Computer-Aided Design*, pp. 1062-1081, 1987.

[CC90] E. J. Chikofsky and J. H. Cross II, "Reverse engineering and design recovery: A taxonomy," in *IEEE Software*, pp. 46-54, January, 1990.

[CK96] K.-T. Cheng and A. S. Krishnakumar, "Automatic generation of functional vectors using the extended finite state machine model," in *ACM Transaction on Design Automation of Electronic Systems*, vol. 1, no. 1, pp. 57-79, 1996.

[CELV99] G. H. Chiisholm, S. T. Eckmann, C. M. Lain, and R. L. Veroff, "Understanding integrated circuits," in *IEEE Design&Test of Computers*, pp. 26-37, 1999.

[CFR+98] E. M. Clarke, M. Fujita, S. P. Rajan, T. Reps, S. Shankar and T. Teitelbaum, "Program slicing for design automation: An automatic technique for speeding-up hardware design, simulation and verification," Computer Science Department University of Wilsconsin, Technical Report, 1998.

[COB95] P. Chou, R. B. Ortega and G. Borriello, "Interface co-synthesis techniques for embedded systems," in *Proceedings International Conference on Computer Aided Design*, pp. 280-287, November, 1995.

[CQ96] D. Chin and A. Quilici, "Decode: A cooperative program understanding environment," in *Journal Software Maintainance*, vol. 8(1), pp. 3-34, 1996.

[DFAB98] A. Dix, J. Finlay, G. Abowd, and R. Beale, *Human Computer Interaction*. Prentice Hall Europe, 1998.

[Dor98] R. C. Dorf, *The Electrical Engineering Handbook*, second edition. CRC Press, 1998.

[Esc99] "Design extractor for Verilog tutorial", Escalade Company, Technical Report, 1999.

[Gaj88] D. D. Gajski, *Silicon Compilation*. Addison Wesley, 1988.

[GC93] E. Girczyc and S. Carlson, "Increasing design quality and engineering productivity through design reuse," in *Proceedings Design Automation Conference*, pp. 48-53, 1993.

[GC99] G. C. Gannod and B. H. C. Cheng, "A framework for classifying and comparing software reverse engineering and design recovery techniques," in *Proceedings Sixth Working Conference on Reverse Engineering, Atlanta*, pp. 77-88, 1999.

[Gep99] L. Geppert, "Design tools for analog and digital ICs," in *IEEE Spectrum*, pp. 41-48, April, 1999.

[GVNG94] D. D. Gajski, F. Vahid, S. Narayan and J. Gong, *Specification and Design of Embedded Systems*. Prentice-Hall, 1994.

[HF00] J. O. Hamblen and M. D. Furman, *Rapid Prototyping of Digital Systems*. Kluwer Academic Publisher, 2000.

[HN90] M. T. Harandi and J. Q. Ning, "Knowledge-based program analysis," in *IEEE Software*, pp. 74-81, January, 1990.

[HR92] S. Horwitz and T. Reps, "The use of program dependence graphs in software engineering," in *Proceedings of the 14th International Conference on Software Engineering*, pp. 392-411, 1992.

[IEE99] *IEEE P1076.6 Standard for VHDL Register Transfer Level Synthesis*. IEEE Press, 1999.

[INIY96] M. Iwaihara, M. Nomura, S. Ichinose and H. Yasuura, "Program slicing on VHDL descriptions and its applications," in *Proceedings 3rd Asian Pacific Conference Hardware Description Languages*, pp. 132-139, Bangalore, 1996.

[Kat94] R. H. Katz, *Contemporary Logic Design*. Benjamin/Cummings Publishing Company Inc., 1994.

[KB98] M. Keating and P. Bricaud, *Reuse Methodology Manual for System-On-A-Chip Designs*. Kluwer Academic Publisher, 1998.

[KCGW98] M. Koegst and P. Conradi and D. Garte and M. Wahl, "A systematic analysis of reuse strategies for design of electronic circuits," in *Proceedings Design Automation and Test in Europe*, pp. 292-296, Paris, France, 1998.

[KNE92] W. Kozaczynski, J. Ning and A. Engberts, "Program concept recognition and transformation," in *IEEE Transaction on Software Engineering*, vol. 18, no. 12, pp. 1065-1074, 1992.

[Kos99] R. Koschke, "Atomic architectural component recovery for program understanding and evolution," Ph.D. dissertation, Institute of Computer Science, University of Stuttgart, 1999.

[Lah00] L. Lahey, "IBM rocketing forward with new ultra-fast microchip," in *ComputerWorld Canada*, pp. 6, June 30, 2000.

[LJ00] C.-N. J. Liu and J.-Y. Jou, "An automatic controller extractor for HDL description at the RTL," in *IEEE Design&Test of Computers*, pp. 72-77, July-September, 2000.

[LWMG96] G. Lehmann, B. Wunder and K. D. Muller-Glaser, "Basic concepts for a HDL reverse engineering tool-set," in *Proceedings International Conference on Computer Aided Design*, pp. 134-141, San Jose, California USA, 1996.

[Mac77] A. K. Mackworth, "Consistency in networks of relations," in *Artificial Intelligence*, vol. 8, pp. 99-118, 1977.

[Mag92] S. Maginot, "Evaluation criteria of HDLs: VHDL compared to Verilog, UDL/I and M," in *Proceedings European Design Automation Conference*, pp. 746-751, 1992.

[MH95] J. Madsen and B. Hald, "An approach to interface synthesis," in *8th International Symposium on System Synthesis*, pp. 16-20, 1995.

[MMH85] A. K. Mackworth, J. Muldter and W. Havens, "Hierarchical arc consistency: Exploiting structured domains in constraint satisfaction problems," in *Computational Intelligence*, vol. 1, pp. 188-196, 1985.

[Moo65] G.E. Moore, "Cramming more Components onto Integrated Circuits" in *Electronics* , 38(8), pp. 114-117, April 1965.

[Muc97] S. S. Muchnick, *Advanced Compiler Design Implementation*. Morgan kaufmann, 1997.

[MV95] A. von Mayrhauser and A. M. Vans, "Program comprehension during software maintainance and evolution," in *Computer*, vol. 28, no. 8, pp. 44-55, August,1995.

[Nad89] B. A. Nadel, "Constraint satisfaction algorithms," in *Computational Intelligence*, vol. 5, pp. 188-224, 1989.

[OAIP98] S. Olcoz, L. Ayuda, I. Izaguirre and O. Penalba, "VHDL teamwork, organization units and workspace management," in *Proceedings Design Automation and Test in Europe*, pp. 297-302, Paris, France, 1998.

[Per98] D. L. Perry, *VHDL, 3rd ed.*. McGraw-Hill, 1998.

[PHSM95] V. Preis, R. Henftling, M. Schutz and S. Marz-Rossel, "A reuse scenario for the VHDL-based hardware design flow," in *Proceedings European Design Automation Conference*, pp. 464-469, 1995.

[PIC01] *PIC-16C5X Family of 8-bit Microcontrollers*. http://www.microchip.com, Microchip Technology Corporation, 2001

[Pro93] P. Prosser, "Hybrid algorithms for the constraint satisfaction problem," in *Computational Intelligence*, vol. 9, pp. 268-299, 1993.

[QYW98] A. Quilici, Q. Yang, and S. Woods, "Applying plan recognition algorithms to program understanding," in *Journal of Automated Software Engineering*, pp. 1-27, 1998.

[RR99] A. Reutter and W. Rosenstiel, "An efficient reuse system for digital circuit design," in *Proceedings Design Automation and Test in Europe*, pp. 38-43, Munich, Germany, 1999.

[RW90a] C. Rich and L. M. Wills, "Recognizing a program's design: A graph-parsing approach," in *IEEE Software*, pp. 82-89, January, 1990.

[RW90b] C. Rich and R. Waters, *The Programmer's Apprentice*. Addison Wesley Publisher, 1990.

[SFM97] M.-A. D. Storey, F. D. Fracchia and H. A. Muller, "Cognitive design elements to support the construction of a mental model during software visualization," in *Proceedings of the 5th International Workshop on Program Comprehension*, pp. 17-28, Dearborn, Michigan, USA, 1997.

[SM98] J. Smith and G. De Micheli, "Automated composition of hardware components," in *Proceedings Design Automation Conference*, pp. 14-19, San Francisco, California, USA, 1998.

[Som97] F. Somenzi, "CUDD: University of Colorado decision diagram package, release 2.1.2," Dept. ECE, University of Colorado, Boulder, Technical Report, 1997

[Syn00] "Design compiler reference manual v2000.05," Synopsys Inc., Technical Report, 2000.

[Til98] S. Tilley, "A reverse-engineering environment framework," Carnegie Mellon Software Engineering Institute, Technical Report, 1998.

[TM91] D. E. Thomas and P. R. Moorby, *The Verilog Hardware Description Language*. Kluwer Academic Publisher, 1991.

[UH01] Uni Hamburg, FB Informatik, AB TECH, "Archive of VHDL models," http://tech-www.informatik.uni-hamburg.de/vhdl, 2001

[VER95] *IEEE 1364-1995 IEEE Standard Description Language Based on the Verilog Hardware Description Language*. IEEE Press, 1995.

[VHD87] *IEEE Std 1076-1987. IEEE Standard VHDL Language Reference Manual.* IEEE Press, 1987.

[VHD93] *ANSI/IEEE Std 1076-1993. IEEE Standard VHDL Language Reference Manual.* IEEE Press, 1993.

[VHD99] *IEEE Std 1076.1 IEEE Standard VHDL Analog and Mixed-Signal Extensions*. IEEE Press, 1999.

[VSI99] Verilog Synthesis Interoperability Working Group, "IEEE p1364.1 d1.4 draft standard for Verilog register transfer level synthesis," http://www.eda.org/vlog-synth, Technical Report, 1999.

[Wei84] M. Weiser, "Program slicing," in *IEEE Transactions on Software Engineering*, pp. 352-357, 1984.

[WY95] S. Woods and Q. Yang, "Program understanding as constraint satisfaction," in *Proceedings 7th international workshop of Computer-Aided Software Engineering*, pp. 318-327, 1995.

[WY96] S. Woods and Q. Yang, "The program understanding problem: Analysis and heuristic approach," in *Proceedings 8th International Conference on Software Engineering*, pp. 6-15, 1996.

Appendix A Glossary

- *Abstract concept*: a common term which is used to refer to both a signal concept or a combinational function concept.
- *Activation condition*: a Boolean expression associated to a concept and which identifies when a concept is valid.
- *Combinational function concept*: an artifact which is used to represent a basic combinational behaviors. These concepts are extracted by the combinational function analysis (see Section 6.2).
- *Conditional expression*: logic expression in a HDL code.
- *Equivalence*. Two set of abstract concepts are equivalent if they represent the same behavior (see Section 7.1 for detail).
- *Extra controller*: a predicate which has been identified by the extra controls analysis (see for Section 7.2 detail).
- *Finer Program Dependence Graph*: (*FPDG*). A graph which is used to represent an algorithmic description. It has been defined in this dissertation.
- *Intellectual Property* (*IP*): a knowledge which belongs to a company.
- *Legacy design*: a component which was implemented with no design for reuse in mind, which is often poorly documented, but which has valuable IP content. In this dissertation, legacy designs consist of synthesizable HDL description at register transfer level.

- *Main input*: the main input of an abstract concept. For signal concepts, the controller is the main input. For encoder and decoder concepts, the input i is the main input. For multiplexer and demultiplexer concepts the selector s is the main input. For comparator concepts, both input i_1 and i_2 are the main inputs.
- *Module*: an artifact which is used to represent an hardware behavior. Modules are extracted by funtional partitioning analysis (see Section 7.3).
- *Predicate*: an element which is identified by decomposition conditional expressions.There are three types of predicate: simple predicate, compare predicate and complex predicate. The values of a predicate pd, indicated by the ordered set $Values_{pd}$, are Boolean products representing all non-overlapping actual interval values of pd. Predicates are used to represent conditional expressions in a VHDL code in term of Boolean expressions.
- *Region condition*: a Boolean expression associated to each region node.
- *Restriction* (or *cofactor*): the restriction of a Boolean expression f by a Boolean product p is indicated by $f|_p$, and is obtained by replacing in the function all variable in p with their constant value. For example, assuming $p=\bar{x}_0x_2$ and $f=\bar{x}_0x_1x_4 + x_1x_2x_3$ by setting $x_0=0$ and $x_2=1$, $f|_p= x_1x_4 + x_1x_3$.
- *Signal concept*: an artifact which is used to represent a specific relation amongst signals. It consists of 5-tuple $sc = \langle tg, ctrl, cond, E, map \rangle$ (see Section 5.1 for detail).
- *Statement condition*: a Boolean expression associated to a statement node. It exists in statement nodes which have been created by the preprocessing of FPDGs. Statement conditions represent guard conditions for the execution of statements.
- *Target*: a signal or a variable to which at least one value is assigned in a VHDL description.
- $Values_{pd}$: the set of Boolean products representing all non-overlapping interval values of a predicate pd.

Appendix B Detailed Algorithms

B.1 Algorithm for signal analysis

In this appendix details of the algorithm which performs signal analysis are presented (refer to Section 5.2.2). Some useful definitions which are used in describing the algorithm, are given in the following list:

- $Values_{pd}$ is the set of Boolean products representing all non-overlapping interval values of a predicate *pd*.
- A *conditional assignment* is a 4-tuple: $ca = \langle tg, pd, co, AL \rangle$, where *tg* is a target, *pd* is a predicate, *co* is a condition represented by a Boolean expression of predicate values and *AL* is a list of assignments. Each assignment $assign(tg) \in AL$ is associated to an interval $v \in Values_{pd}$, that is *AL* is isomorphic to a subset $V \subseteq Values_{pd}$ and $|AL| \leq |Values_{pd}|$.
- *Asscond* (assignment condition) is a Boolean expression of predicate values associated with a statement (see Section 5.2.1).
- *Regcond*(R_i) (region condition) is a Boolean expression of predicate values associated with region node R_i (see Section 4.1).
- A *restriction* (or *cofactor*) of a Boolean expression *f* by a Boolean product *p* is indicated by $f|_p$, and is obtained by replacing in the function all variable in *p* with their constant value [Bry86]. For example, assuming $p=\bar{x}_0x_2$ and $f=\bar{x}_0x_1x_4 + x_1x_2x_3+x_0\bar{x}_2$ by setting $x_0=0$ and $x_2=1$, $f|_p= x_1x_4 + x_1x_3$.

Steps	
	Given a FPDG and following a postorder traversal
	at each region node R_i with *regcond*(R_i)
	create an empty set *regionCA*(R_i)
1	visit each child following the order defined by the control flow edges
1.1	if the child is a statement node with *assign*(*tg*) and *asscond* then
1.1.1	each $ca \in regionCA(R_i)$ with target *tg* is removed
1.1.2	for each predicate *pd* on which *regcond*(R_i) depends, new conditional assignments as $nca = \langle tg, pd, f, AL \rangle$ are created and inserted in set *regionCA*(R_i). *AL* contains one or more *assign*(*tg*) and condition *f* is calculated considering *regcond*(R_i), *asscond* and $Values_{pd}$.
1.1.3	if no predicate *pd* exists which depends on *regcond*(R_i) the *assign*(*tg*) is used to form a *logic net* signal concept
1.2	if the child is a predicate node P_i
1.2.1	each $pca = \langle tg, pd, pco, pAL \rangle \in predicateCA(P_i)$ is compared and eventually merged with existing conditional assignments in *regionCA*(R_i) with the same target
	at each predicate node P_i
2	initialize *predicateCA*(P_i) with the *regionCA* of one of the region's children
3	visit each remaining region child R_i with no specific order
3.1	each $rca = \langle tg, pd, rco, rAL \rangle \in regionCA(R_i)$ is compared and eventually merged with existing conditional assignments in *predicateCA*(P_i) with the same target
	after traversal, each collected condition assignment is mapped to a signal concept.

FIGURE B.1-1. Pseudo-code of signal analysis

For convenience, the pseudo-code of the algorithm is reported in Figure B.1-1.

Let us consider each of the above main steps.

- *Step1*. Each child of a region node is considered following the order indicated by the control flow edges. This order represents the original sequential dependency among statements of the VHDL code.
- *Step1.1*. A statement node may represent an assignment or a procedure call. However, before signal analysis algorithm is executed, each procedure call has been substituted by its equivalent FPDG. Therefore only an assignment exists for each statement node.

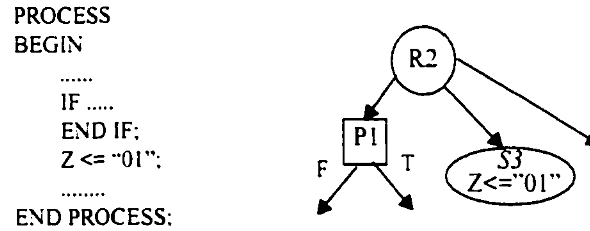


FIGURE B.1-2. Example for step 1.1.1 at region node

Note, that the statement condition *asscond* associated to a statement node is calculated during preprocessing activity (see Section 5.2.1) when variables were used in the right hand side (RHS) of the assignment. In all other cases *asscond* is equal to 1.

- *Step1.1.1.* Assuming that *tg* is the target of the assignment considered in *step1.1*, each $ca \in regionCA(R_i)$ with target *tg* is removed. This step considers the overriding property of sequential assignments, that is the fact that only the last assignment is valid in a sequence of assignments to the same target. Considering the VHDL code and FPDG in Figure B.1-2, no matter how complicated is the code which is executed before the statement node *S3*, the assignment *Z <= "01"* is always executed. Consequently, all previous assignments to the target *Z* are overwritten by this assignment. It means that all conditional assignments collected in *predicateCA(P1)* and moved to *regionCA(R2)* by step 1.2.1 are not valid anymore and should be removed.

- *Step 1.1.2.* Considering each predicate *pd* on which $regexp(R_i)$ depends, new conditional assignments as $nca = \langle tg, pd, f, AL \rangle$ are created and inserted in the set $regionCA(R_i)$. In particular, the condition $co = regcond(R_i) \cdot asscond$ is calculated and:

  for each predicate *pd* on which $regexp(R_i)$ depends
  - an empty set *CA* of conditional assignments is created
  for each $r \in Values_{pd}$
  - a condition *newco* equals to $co|_r$
  - if $\exists(ca = \langle tg, pd, f, AL \rangle \in CA)$ such that $f \cdot newco \neq 0$ then *assign(tg)* is added to set *AL* and $f \cdot newco$ becomes the new value of *f*, otherwise a new conditional assignment $nca = \langle tg, pd, newco, AL \rangle$ with $AL = \{assign(tg)\}$ is added to the set *CA*
  all conditional assignments in *CA* are added to $regionCA(R_i)$

Note that more than one conditional assignment may be inserted in the set *CA* because condition *co* may consist of an expression like (*A*='1' AND *B*='0') OR (*A*='0' AND *B*= '1'). In this case, for example, considering predicate *A*, when $r \in Values_{pd}$ corresponds to *A='1'*, *newco* equals a Boolean expression representing *B*='0'. However, when $r \in Values_{pd}$ corresponds to *A*='0', *newco* equals a Boolean expression representing *B*='1'. This latter value of *newco* is not compatible with previous values of *newco* therefore two different conditional assignments are created. One conditional assignment represents the fact that *assign*(*tg*) is executed when *B*='0' and controlled by predicate *A*='1', while the other conditional assignment represents the fact that *assign*(*tg*) is executed when *B*='1' and controlled by predicate *A*='0'.

- *Step1.1.3.* If no predicate *pd* which depends on *regcond*(R_i) exists, it means that the assignment *assign*(*tg*) does not depend on any predicate. In this case, a *logic net* signal concept for the target *tg* is identified with condition equal to *regcond*(R_i).
- *Step1.2.* Predicate node P_i is considered in which previously collected conditional assignments exist.
- *Step1.2.1.* This is a complicated step in which each conditional assignment $pca \in predicateCA(P_i)$ is compared and eventually merged with existing conditional assignments with the same target in *regionCA*(R_i). In particular:

  for each $rca = \langle rtg, rpd, rco, rAL \rangle \in regionCA(R_i)$

  for each $pca = \langle ptg, ppd, pco, pAL \rangle \in predicateCA(P_i)$ with *ptg*=*rtg*

  ppd=*rpd* and $pco \cdot rco \neq 0$

  - *case1.* $\exists(r \in Values_{rpd})$ to which an assignment $ral \in rAL$ is associated but $\neg\exists(pal \in pAL)$ associated to *r*, that is the assignments in the *pca* conditional assignment do not cover all ranges of values of predicate *ppd* which are covered by the assignments in *rca*. In this case, a new conditional assignment $nca = \langle ptg, ppd, pco \cdot rco, AL \rangle$ is created and added to *regionCA*(R_i), but it is not considered anymore in the loop since all $ca \in predicateCA(P_i)$ with the same target and predicate have conditions which are not compatible. *AL* is composed of *pAL* and all $ral \in rAL$ to which $r \in Values_{rpd}$ is associated but

$\neg\exists(pal \in pAL)$ associated to *r*. Moreover, if $rco \cdot \overline{pco} = 0$ then *rca* is removed from set *regionCA*(R_i), otherwise its condition is updated to $rco \cdot \overline{pco}$. The condition of the conditional assignment *pca* is also updated to $pco \cdot \overline{rco}$

- *case2*. For each $r \in Values_{rpd}$ associated to $ral \in rAL$,
  $\exists(pal \in pAL)$ associated to *r*, that is the assignments in the *pca* conditional assignment cover all ranges of values of predicate *ppd*. In this case, if $rco \cdot \overline{pco} = 0$ then *rca* is removed from set *regionCA*(R_i), otherwise its condition is updated to $rco \cdot \overline{pco}$

for each $pca = \langle ptg,ppd,pco,pAL\rangle \in predicateCA(P_i)$ with *ptg=rtg*, $pdd \neq rpd$ and $pco \cdot rco \neq 0$

- *case3*. Considering the Boolean sum $af = \sum_r r$ with $r \in Values_{ppd}$
  such that $\exists(al \in pAL)$ associated to *r*. If $rco \cdot \overline{pco \cdot af} = 0$ then *rca* is removed from the set *regionCA*(R_i), otherwise its condition is updated to $rco \cdot \overline{pco \cdot af}$

all *pca* conditional assignments in *predicateCA*(P_i) where the condition is still different from 0 after the above operations, are added to *regionCA*(R_i)

In order to clarify different cases of step *1.2.1*, we consider some examples.

- Operations of *case1* may be clarified by looking at the example in Figure B.1-3 and considering:

  $rca = \langle Z,A,(B = 0),[S1,S2]\rangle \in regionCA(R0)$ and

  $pca = \langle Z,A,(C = 1),[S3]\rangle \in predicateCA(P2)$.

```
PROCESS
BEGIN
    IF (A ='1') AND (B = '0') THEN
        Z <= "01";
    ELSE
        Z <= "11";
    END IF;
    IF (A='0') AND (C = '1') THEN
        Z <= G;
    END IF;
END PROCESS;
```

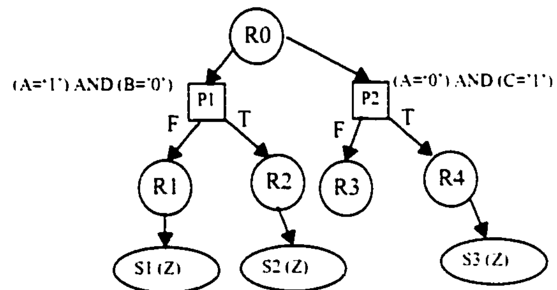


FIGURE B.1-3. Example case1 step 1.2.1

A new conditional assignment *nca=<Z,A,B='0' and C='1',[S2,S3]>* is obtained by merging the conditional assignments *rca* and *pca*. Condition of *rco* is updated such that *rca=<Z,A,B='0' and C='0',[S1,S2]>*. Condition of *pco* is updated such that *pca=<Z,A,B='1' and C='1',[S3]>*. This case delineates situations when two condition assignments are active under a same condition and the one which occurs sequentially later does not completely override the other conditional assignment.

- Operations of *case2* may be clarified by looking at the example in Figure B.1-4. At region node *R0*, we consider $rca = \langle Z,A,1,[S1]\rangle \in regionCA(R0)$, and both:
  $pca1 = \langle Z,A,(C = 1),[S2,S3]\rangle \in predicateCA(P2)$,
  $pca2 = \langle Z,A,(C = 0),[S3,S3]\rangle \in predicateCA(P2)$.
  When *pca1* is considered, the condition of *rca* is updated to *C*='0', then when *pca2* is considered the *rca* is removed from *regionCA(R0)*. This case delineates situations when two condition assignments are active under the same condition and the one which occurs sequentially later completely overrides the other conditional assignment.
- Operations of *case3* may be clarified by looking at the example in Figure B.1-5. At region node *R0*, we consider $rca = \langle Z,A,1,[S1]\rangle \in regionCA(R0)$ and $pca = \langle Z,C,1,[S2,S3]\rangle \in predicateCA(P2)$. Since *af*=1, then $rco \cdot \overline{pco \cdot af} = 0$ and the conditional assignment *rca* is removed from *regionCA(R0)*. In this example, since for any value of predicate *C* an assignment is activated for target *Z*, all previous conditional assignments, which occur sequentially earlier, are not valid, therefore *rca* is not valid.

```
PROCESS
BEGIN
    IF (A ='1') THEN
        Z <= "01";
    END IF;
    IF (A='0') AND (C = '1') THEN
        Z <= G;
    ELSE
        Z <= "11";
    END IF;
END PROCESS;
```

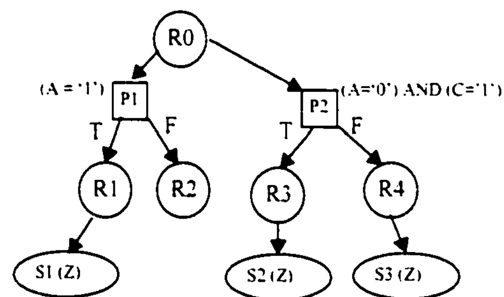


FIGURE B.1-4. Example case2 step 1.2.1

```
PROCESS
BEGIN
    IF (A ='1') THEN
        Z <= "01";
    END IF;
    IF (C = '1') THEN
        Z <= G;
    ELSE
        Z <= "11";
    END IF;
END PROCESS;
```

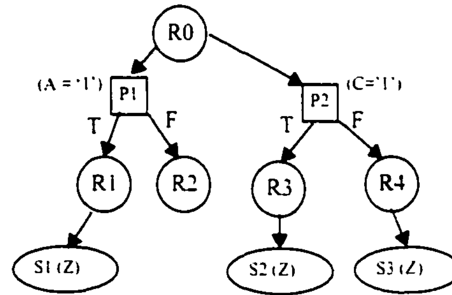


FIGURE B.1-5. Example case3 step 1.2.1

- *Step2*. Since all children of a predicate node are mutually exclusive by definition, there is no sequential relation among their statements, therefore any child may be taken as the starting node to initialize *predicateCA*(P_i).

- *Step3*. Similar to *step2*, the remaining children of predicate node P_i may be considered without any specific ordering.

- *Step3.1*. This is a complicated step in which each $rca \in regionCA(R_i)$ is compared and eventually merged with existing conditional assignments with same target in *predicateCA*(P_i). In particular:

  for each $pca = \langle ptg,ppd,pco,pAL \rangle \in predicateCA(P_i)$

    for each $rca = \langle rtg,rpd,rco,rAL \rangle \in regionCA(R_i)$ with *rtg=ptg, rpd=ppd,*

     and $pco \cdot rco \neq 0$

      - a new conditional assignment $nca = \langle ptg,ppd,pco \cdot rco,pAL \cup rAL \rangle$
      is created and added to *predicateCA*(P_i). Moreover, if $pco \cdot \overline{rco} = 0$
      then *pca* is removed from *predicateCA*(P_i), otherwise its condition is
      updated to $pco \cdot \overline{rco}$. The condition of the conditional assignment *rca* is
      also updated to $rco \cdot \overline{pco}$

  all *rca* conditional assignments in *regionCA*(R_i) where the condition is
  still different from 0 after the above operations, are added to *predicateCA*(P_i)

Note that the condition $pco \cdot rco \neq 0$ forces consideration of only conditional assignments which have a predicate which belongs to the condition in current considered predicate node P_i. In fact, any pair of conditional assignments with the same predicate which does not belong to the condition in the predicate node, must have incompatible conditions since each condition assignment belongs to a different branch of the predicate node. In

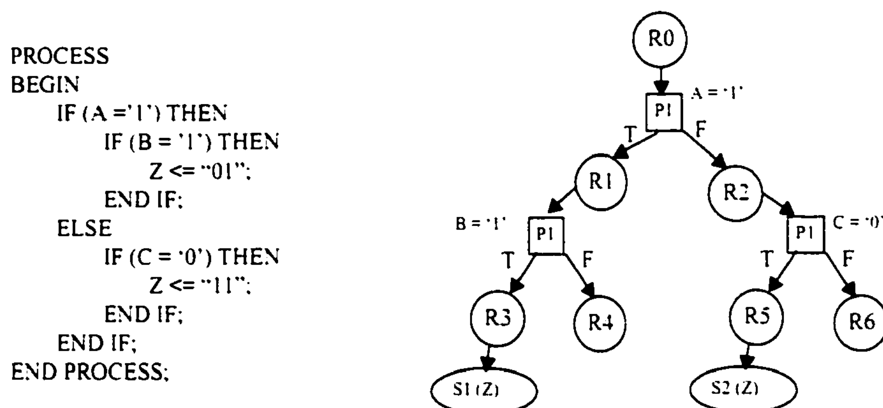


FIGURE B.1-6. Example for step 3.1

Figure B.1-6 an example which should clarify *step3.1* is depicted. At predicate node *P1*, assuming conditional assignment *pca*=<*Z*,*A*,*B*='1',[*S1*]> and *rca*=<*Z*,*A*,*C*='0',[*S2*]> the step creates a new conditional assignment *nca*=<*Z*,*A*,*B*='1' and *C*='0',[*S1*,*S2*]>. Moreover conditions of both *pca* and *rca* are updated *pca*=<*Z*,*A*,*B*='1' and *C*='1',[*S1*]> and *rca*=<*Z*,*A*,*C*='0' and *B* = '0',[*S2*]>.

B.2 Find subset of signal concept for encoder, decoder and demultiplexers

In this section, explanations and details of step 4.2 of the procedure, which identifies encoders, decoders and multiplexers (refer to Section 6.3.1), are given. The input parameters of the procedure is a set S of signal concepts $s_i = \langle tg, ctrl, cond_i, E_i, map_i \rangle$, with the same target tg and the same controller $ctrl$, and a set $C=\{ctrl_1, ctrl_2, \ldots, ctrl_n\}$ of controllers. The output consists of the largest collection $Q=\{Q_1, Q_2, \ldots, Q_m\}$ where $Q_k \subseteq S$ such that:

- each Q_k satisfies *cnd2* and *cnd3* of theorem 6.2 and represents a combinational function concept. A main input d_k is associated to Q_k and is composed by concatenating controllers of $C_k \subseteq C$. Also associated to Q is an activation condition $cond_k$ which is given by the expression b of *cnd3* in theorem 6.2.
- for each Q_k with main input d_k, there is no Q_j with main data d_j and $i \neq j$, such that d_i is contained in d_k and $cond_i \cdot cond_j \neq 0$. A main input d_k contains a main input d_j if, and only if, every controller in d_j belongs also to d_k, that is $\forall(c \in d_j) \to c \in d_k$.

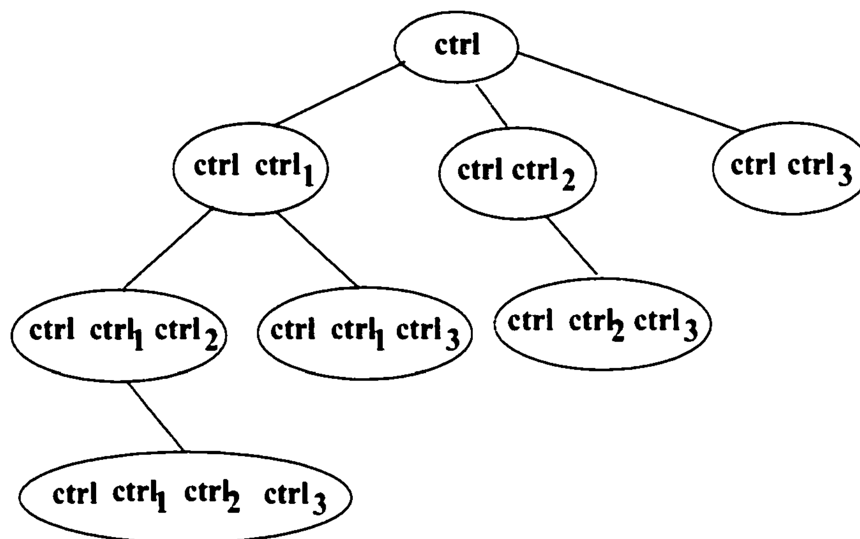


FIGURE B.2-1. Search tree

Using a branch and bound approach, the algorithm of step 4.2 creates the collection Q using a search tree which represents all possible combinations, with no repetition, of controllers in C. Each combination represents a possible main input d_k. Figure B.2-1 depicts the complete search tree for a case in which $C=\{ctrl_1, ctrl_2, ctrl_3\}$. At each node k, a set $G(k)=\{G_1, G_2, ... G_h\}$ with $G_j \subseteq S$ is built. The root node consists of a set $G(ctrl)=\{G_1, G_2, ... G_h\}$, where each G_j is a single element set consisting of a signal concept s in S. Starting from the root and using a *preorder* traversal, at each node k, the algorithm creates a set $G(k)=\{G_1, G_2, ... G_h\}$ with $G_j \subseteq S$ as described in Figure B.2-2. For a node k, it might result that $G(k)$ is empty (Φ). In this case, no further search is done along the branches starting from the node k. That is all combinations of controllers which are represented by the children of the node k are not considered.

Once all sets $G(k)$ have been built, the procedure proceeds using a *postorder* traversal of the tree. In a node k (with $G(k) \neq \Phi$), each $G_j \in G(k)$ is considered as follow:

- if G_j is a set marked as *used* or it derives from sets in parent nodes which have all been marked as *used*, then G_j is discarded, otherwise
- if G_j forms a combinational function , then G_j is included in the final collection Q. All sets in parent nodes from which G_j is derived are marked as *used*.

By discarding sets G_j which are marked as *used* or derived from previously used sets (marked as *used*), it is avoided that the algorithm builds a combinational function concept which is a sub concept of another previously identified concept. That is, it is avoided to build a concept with a main input which is contained in a main input of another concept and the two concepts have a common activation condition. In fact, considering Figure B.2-1, let us assume that a combinational function concept c from a G_j in node "ctrl $ctrl_1$ $ctrl_2$ $ctrl_3$" is found and included in collection Q. The set G_j is composed of some sets in the parent nodes "ctrl $ctrl_1 ctrl_2$", "ctrl $ctrl_1$" and "ctrl". These sets are marked as *used*. When the algorithm visits the node "ctrl $ctrl_1$ $ctrl_2$", it does not consider the marked sets beacuse they are already part of the concept c which has a larger main input. Similarly when visiting the node "ctrl $ctrl_1$ $ctrl_3$", the sets which derive from sets marked as used in node "ctrl $ctrl_1$" should not be considered. If considered, they will identify a concept which is valid at the same time as the concept c and with main input "ctrl $ctrl_1$ $ctrl_3$" which is contained in the main input of c.

To further reduce the computation of the algorithm which performs step 4.2, the initial set $C=\{ctrl_1, ctrl_2,...,ctrl_n\}$ of controllers may be reduced using a preprocessing. Considering

```
initialize G(k)=Φ
for each p ∈ Values(d_k)
    for each G'_i ∈ G'  where G' is the set built in the parent node of node k
        cond_i = ∏_{s ∈ G'_i} cond_s|_p  with cond_s is the activation condition for each s in G'_i
        if cond_i ≠ 0 then
            for each G_j ∈ G(k)
                if cond_i · cond_old ≠ 0 where cond_old = ∏_{s ∈ G_j} cond_s|_p then
                    update G_j = G_j ∪ G'_i
            if p is the first value and G'_i was not inserted in any G_j then
                create a new G_j={G'_i} and insert in G(k), keep cond_i associated to G_j
    each G_j which has not been updated for a given p is removed from G(k)
```

FIGURE B.2-2. Create the set G(k) in a node k

each controller $ctrl_j$, if there exists a value $u \in Values(ctrl_j)$ for which no signal concept s_i=<$tg,ctrl,cond_i,E_i,map_i$> exists in S with $cond_i$ depending on $ctrl_j$ and for which $cond_i \cdot u \neq 0$, then the controller $ctrl_j$ is removed from the set C. In fact in this case, by construction, any node k in which $ctrl_j$ is included in d_k results in a empty $G(k)$.

Summarizing, the basic steps of the algorithm which performs step 4.2 of the procedure described in Section 6.3.1, are the following:

Given the sets S and C and the controller *ctrl* which is the controller of all $s \in S$

- preprocess C to eventually reduce it
- build the set $G(ctrl)$ of the root node of the search tree
- build the search tree using the branch and bound approach
- traverse in *postorder* the search tree and build collection Q

Even if the algorithm is intrinsically of exponential complexity in $|C|$, considering the test cases used in this dissertation, it turns out that the computation of this algorithm is practical. In practical cases, the number of controllers in the set C is small because, generally, there are not many signals on which a specific target depends on. Moreover, during the construction of the search tree, there are nodes in which the set $G(k)$ results empty and therefore the search tree is reduced.