UNIVERSITY OF CINCINNATI

February 7, 2001

I, Nazanin Mansouri,
hereby submit this as part of the requirements for the degree of:

Doctor of Philosophy

in:

Dept. of Elec. and Comp. Eng. and Comp. Science

It is entitled:

Automated Correctness Generation for Formal Verification of Synthesized RTL Designs

Approved by:
Dr. Ranga Vemuri
Dr. Perry Alexander
Dr. Steve Pelikan
Dr. Carla Purdy
Dr. John Schlipf

Automated Correctness Condition Generation for Formal Verification of Synthesized RTL Designs

A dissertation submitted to the

Division of Research and Advanced Studies
of the University of Cincinnati

in partial fulfillment of the
requirements for the degree of

DOCTOR OF PHILOSOPHY

in the Department of
Electrical and Computer Engineering and Computer Science
of the College of Engineering

February 7, 2001

by

Nazanin Mansouri

B.E (Computer Engineering),
College of Engineering,
Iran University of Science and Technology - Tehran,
Iran, 1992

Thesis Advisor and Committee Chair: Dr. Ranga R. Vemuri

Abstract:

This work presents a formal methodology for verifying the functional correctness of synthesized register transfer level designs (RTL) generated by a high-level synthesis system. The verification is conducted by proving the observation equivalence of the RTL design with a description of its desired behavior.

High-level synthesis tools generate register transfer level designs from algorithmic behavioral specifications. The high-level synthesis process consists of dependency graph scheduling, function unit allocation, register allocation, interconnect allocation and controller generation tasks. Widely used algorithms for these tasks retain the overall control flow structure of the behavioral specification allowing code motion only within basic blocks. Further, high-level synthesis algorithms are in general oblivious to the mathematical properties of arithmetic and logic operators, selecting and sharing RTL *library modules solely based on matching uninterpreted function symbols and constants. Many researchers have noted that these features of high-level synthesis algorithms can be exploited to develop efficient verification strategies for synthesized designs. This dissertation reports a verification methodology that effectively exploits these features to achieve efficient and fully automated verification of synthesized designs.*

Contributions of this research include formalization and formulation in higher-order logic in a theorem proving environment mathematical models for the synthesized register transfer level designs and their behavioral specifications and a set of sufficient correctness conditions for these designs. It presents an in depth study of pipelining in design synthesis, and identifies the complete set of correctness conditions for RTL *designs generated through the synthesis processes that employ pipelining techniques.*

This method has been implemented in a verification tool (the correctness condition generator, CCG) and is integrated with a high-level synthesis system. CCG generates (1) formal specifications of the behavior and the RTL *design including the data path and the controller, (2) the correctness lemmas establishing equivalence between the synthesized* RTL *design and its behavioral specification, and (3) their proof scripts that can be submitted to a higher-order logic proof checker. The tool performs model extraction, correctness condition generation and proof generation automatically and without human interaction. This approach is based on the identification, by the synthesis tool during the synthesis process, of the binding between critical specification variables and critical registers in the* RTL *design and between the critical states in the behavior and the corresponding states in the* RTL *design. CCG is capable of handling a broad range of behavior constructs that may be used for specifying the behavior and a wide variety of algorithms that may be employed during the synthesis process. Also, the verification algorithms of CCG have the appealing feature of relying on symbolic analysis of the uninterpreted values of the variables and registers. This has resulted in a considerable reduction in verification time compared to other post-synthesis verification systems, that are often restrained by this factor.*

To my mother Pary,
To Adrian,
for believing in me.

Acknowledgements

I take this opportunity to express my thanks to several people who have made this dissertation possible. I would like to thank my academic advisor Dr. Ranga Vemuri for providing support, motivation and guidance throughout my work at the University of Cincinnati. I thank Dr. Perry Alexander who has been a model for me as a teacher and as a person. I thank Dr. John Schlipf who has one of the most generous hearts, and has shown me throughout the years how to be great as a scientist and as a being. I also would like to convey my gratitude to the wonderful professors at UC whom I have had the pleasure to know and work with: Dr. Karen Davis, Dr. Carla Purdy, Dr. Hal Carter, and Dr. Philip Wilsey. I thank Dr. Pelikan, and the other members of my dissertation committee. This work is sponsored in part by DARPA and monitored by US Army Ft. Huachuca under contract number DABT63-96-C-0051.

I thank my colleagues at UC and all members of the DDEL lab who made my years of studying more pleasant. I would like to thank Meenakshi, Christophe, Peter, Ramanan, Preetham, Karam, Arun, Iyad, Rajesh, Jeff, Satish, Sree, Abhijit and Priya, and specially Ela - who is the sunshine in the cloudy moments of graduate school - for their friendship. I am thankful to all the Synthesis experts at DDEL who provided many interesting design examples for this dissertation,

My life has not been a smooth journey. I have lost many people dear to my heart and I have had many gloomy days. I would have surrendered to sorrow if it have not been for the wonderfulness of the people in my life: My friend Badri who has always stood by me, my friend Mehdi who has never lost faith in me, even when I did, my friends Jorgen, Chris, Scott, Shiva, Ziba, Roxana, Shidokht, Shokoufeh, Maria, Sanghwa, June, Marc, Kataneh, Leyla, Elaheh, Mojgan, Farahnaz, Andy, Ilona and my dearest cousin Saeed who have showered my life with their astonishing goodness and made it so much richer and more delightful.

I would like to thank my father for loving me and for supporting my education in any possible way. I would like to give particular thanks to my uncle Fery who has never even owned a watch, but has gifted me with a book for every week that I have lived. For all our afternoon walks, for all the smiles he brought to my sometimes sad childhood, for our frequent trips to the bookstore, theater, movies, for all the books that he gave me, for the endless hours that he sat and read for me, for everything he has taught me, and for all the love he has given me. I thank my uncle Mehdi for treating me as an equal since my early age, for all the enlightening discussions that he gifted me with, for being a great uncle, a comrade, and a friend. I thank my grandmother Molouk-joon for her infinite love, and for showing me with her life the essence of generosity, patience and forgiveness. I like to thank my sister Behin for her friendship and love, and for all the beauty and joy she has brought to my life. I thank my mother Parvin, for being the source of inspiration in my life, for her extraordinary sacrifices that gave me a better life than what she had, for her sleepless working nights that provided me with an education, for teaching me the joy of thinking and living independently as a woman, for loving me beyond the borders of time, for always believing in me and encouraging me in achieving my goals, and for being the best mother one can ask for. Lastly, I would like to thank my partner in life, Adrian, for his genuinely loving heart, for teaching me the true meaning of giving and trust, for helping me to remain faithful to myself, to my beliefs, and to the truth, and for being the source of strength for me in struggling with the hardships, and achieving my goals.

Contents

1 Introduction **1**

1.1 Motivation 2

1.2 Objectives 4

1.3 Scope of Our Verification Approach 5

1.4 An Overview of High-Level Synthesis Process 7

1.5 Background 10

1.6 Correctness 11

1.7 Classification 11

1.8 Formal Verification Methods 11

1.9 Formal Proof Process 12

1.10 Overview 14

1.11 Outline of the Thesis 15

2 A Survey of Related Research **17**

2.1 Transformation Based or Formal Synthesis Methods 17

2.2 Post-Design Verification Methods 18

2.3 Symbolic Simulation Methods 19

2.4 Simulation-Based Methods 20

2.5 Discussion 20

3 Correctness **21**

3.1 Behavior 22

3.2 Equivalence 23

3.3 Equivalence Checking through Decomposition 25

4 Formal Specification and Modeling **33**

4.1 Modeling 33

4.2 Specification 34

4.3 Logical Framework 35

4.4 Behavior Model 37

4.5 Design Model 39

4.6 Formal Specification of the Behavior 40

4.7 Formal Specification of the Design 41

4.7.1 RTL Component Specification 42

4.7.2 Type and Interconnection of RTL Components 42

4.7.3 RTL Controller 42

4.8 Conclusion 43

5 Deploying Methodical Design Construction Knowledge for Identifying Design Properties **45**

5.1 An Overview of High-Level Synthesis Process 46

5.2 Impact of Scheduling on Verification Methodology 48

5.2.1 Scheduling of Non-Hierarchical CDFGs 52

5.2.2 Scheduling of Hierarchical CDFGs 55

5.2.3 Discussion 56

5.3 Impact of Operator Allocation on Verification Methodology 57

5.4 Impact of Register Allocation on Verification Methodology 58

5.4.1 Carrier Based Register Allocation 59

5.4.2 Value Based Register Allocation 62

5.5 Impact of Interconnect Allocation on Verification Methodology 64

5.5.1 Bit-Level Interconnect Allocation 64

5.6 Impact of Controller Generation on Verification Methodology 68

5.7 Conclusion 68

6 Formalization of the Verification Technique **69**

6.1 Critical Verification Elements 69

6.2 Formalization 71

6.3 Dynamic Register Binding and Criticality Masking Technique 77

6.4 Conclusion 79

7 Proof Construction **80**

7.1 Proof Strategy 82

7.2 Proof Tactics 83

7.3 Proof Steps 84

7.4 Inference Rules and Axioms 86

7.5 Conclusion 87

8 Correctness Condition Generation **89**

8.1 Generic Axioms 91

8.2 Behavior Axiom Generation 91

8.3 Data Path Axiom Generation 93

8.4 Controller Axiom Generation 95

8.5 Generation of Critical Path Equivalence Lemmas 96

8.6 Generation of Proof Scripts 96

9 Structural Pipelining in Design Synthesis **99**

9.1 Multi-Cycle Resources 100

9.1.1 Synthesis with Unconstrained Non-pipelined Multi-Cycle Resources 106

9.2 Pipelined Resources 123

9.3 Synthesis with Resource Constraints 138

9.4 Conclusion 140

10 Loop Winding and Functional Pipelining **141**

10.1 Verification of Non-Pipelined Loops 141

10.1.1 Pipelined Loops 155

10.1.2 Verification of Pipelined Loops 158

10.1.3 Verification of Pipelined Loops : An Alternative Approach 169

10.1.4 Discussion 181

10.1.5 Concluding Remarks 182

10.2 Functional Pipelining in Design Synthesis 183

10.2.1 Sequential Specifications 183

10.2.2 Specifications with Conditional Constructs 183

10.2.3 Specifications with Conditional and Iterative Constructs 186

10.3 Conclusion 186

11 Implementation and Results **187**

11.1 Discussion 188

12 Conclusion and Future Work **198**

12.1 Discussion 198

12.2 Future Work 199

12.3 Summary of Contributions 201

A PVS Specification and Proof of Equivalence Theorem **202**

B A Comprehensive Example **206**

C Symbolic Analysis of Specifications and Implementations **264**

C.1 Specification Variables 266

C.1.1 Mathematical and Logical Expressions 266

C.1.2 Statements in the Specifications 267

C.1.3 Specification Variables and Transition Sequences 268

C.2 Register Transfer Operations 270

C.3 Combinational and Sequential Components 271

C.3.1 Data Transfers in an RTL Design 274

C.3.2 Application of Symbolic Analysis in Verification of Synthesized RTL Designs 276

Bibliography **276**

List of Figures

1.1 Synthesis Hierarchy 6

1.2 High-Level Synthesis Process 8

3.1 Constructing G_s'' from G_s 26

3.2 Composition of S' from Communicating subsystems 28

3.3 Composition of I' from Communicating subsystems 30

4.1 A Behavior Specification 36

4.2 Example of an RTL design generated by a high-level synthesis system . . 38

4.3 Relational description of the behavior of register component 41

4.4 HDL description of the port-map of the multiplier component 42

5.1 A Sequential Basic Block 50

5.2 Scheduled Sequential Basic Block 50

5.3 Scheduled Conditional Basic Block 51

5.4 Scheduled Iterative Basic Block 52

5.5 Scheduled Procedure Basic Block 52

5.6 Control-step assignment to a sequential basic block during the global scheduling of a non-hierarchical CDFG 53

5.7 Control-step assignment to a conditional basic block during the global scheduling of a non-hierarchical CDFG 54

5.8 Control-step assignment to an iterative basic block during the global scheduling of a non-hierarchical CDFG 55

5.9 Control-step assignment to a procedure basic block as part of scheduling of a non-hierarchical CDFG 56

5.10 The Effects of Scheduling 57

5.11 Mapping relation between specification variables and design registers when no register optimization is performed 58

5.12 Specification of the Greatest Common Divisor Design 60

5.13 Example of the data-path of an RTL design generated by a high-level synthesis system after scheduling and resource optimization 61

5.14 Mapping relation between specification variables and design registers in a carrier-based register allocation scheme 62

5.15 Example of a partial specification 62

5.16 Mapping relation between specification variables and design registers in a value-based register allocation scheme 63

5.17 Virtual Components : Accounting for Bit-Level Interconnections 66

5.18 Detailed SPLIT Component 67

5.19 Detailed SPLIT Component 67

7.1 Hierarchical proof generation 81

7.2 Relational description of the multiplier component 84

7.3 Component interconnections as part of the data-path of an RTL design . 85

7.4 HDL description of the port-map of the multiplier component 86

8.1 Stages of Correctness Condition Generation 90

8.2 Some Generic Declarations 91

8.3 Some Basic Design-Specific Definitions and Axioms 92

8.4 Axioms for Some Behavior Transitions 93

8.5 Axioms for Some RTL Library Components 94

8.6 **Axioms for Some Data Path Components** 95

8.7 **RTL Controller** 95

8.8 **Some Critical Path Equivalence Lemmas** 96

8.9 **Proof Scripts for a Correctness Theorem** 98

9.1 **The stages of a multi-cycle component** 101

9.2 **Partial specification adopted from the Differential Equation benchmark** . 103

9.3 **Formal description of the partial specification of Figure 9.2** 105

9.4 **A possible unconstrained schedule for the partial specification of Figure 9.2** 106

9.5 **Data path of the design synthesized from the schedule of Figure 9.4** 107

9.6 **Controller of the design synthesized from the schedule of Figure 9.4** 108

9.7 **Description of behavior of the components in the data-path of Figure 9.5** 110

9.8 **The stages of a pipelined component** 123

9.9 **A possible unconstrained schedule with pipelined resources for the partial specification of Figure 9.2** 124

9.10 **Data path of the design synthesized from the schedule of Figure 9.9** 126

9.11 **Controller of the design synthesized from the schedule of Figure 9.9** 127

9.12 **Description of behavior of the components in the data-path of Figure 9.10** 129

9.13 **A possible schedule with constrained non-pipelined resources for the partial specification of Figure 9.2** 139

9.14 **A possible schedule with constrained pipelined resources for the partial specification of Figure 9.2** 140

10.1 **A simple loop basic block** 142

10.2 **The unrolling of a simple loop basic block** 145

10.3 **An iterative basic block** 155

10.4 **Sub-block dependencies** 156

10.5 **A possible pipelined version of the loop of Figure 10.3** 157

10.6 **A pipelined implementation of the loop** 158

10.7 **Control flow graph of the pipelined loop at the steady state** 159

10.8 **All the iterations of the pipelined loop with distinctive** CFGs 160

10.9 **The control graph of a pipelined loop** 161

10.10 **The life span of two consecutive iterations of a pipelined loop** 162

10.11 **Backward Value Propagation Algorithm** 168

10.12 **Register assignment in a pipelined implementation of a loop** 170

10.13 **The control graph of a pipelined loop revisited** 171

10.14 **Critical registers at each stage of execution of an iteration** i 173

10.15 **Balancing a Conditional Basic Block where** $(i < j)$ 184

11.1 **Proof Time as Function of the Number of Critical Paths** 189

11.2 **Proof Time as Function of the Number of Critical Registers** 190

11.3 **Proof Time as Function of the Total Number of States** 191

11.4 **Proof Time as Function of the Number of Lemmas** 192

11.5 **Average Proof Time of DECOMPRESS Equivalence Lemmas as Function of the Number of States per Path** 194

11.6 **Average Proof Time of TLC Equivalence Lemmas as Function of the Number of States per Path** 195

11.7 **Average Proof Time of MOVE Equivalence Lemmas as Function of the Number of States per Path** 196

A.1 **Execution Path Equivalence Theory** 203

A.2 **List Induction Theory** 203

A.3 **PVS Proof of Execution Path Equivalence Theorem** 205

B.1 **VHDL Behavioral Description of a Traffic Light Controller** 207

B.2 **Intermediate Behavioral HDL Description of the Traffic Light Controller** 209

B.3 **Intermediate Structural HDL Description of the Traffic Light Controller** . 218

B.4 **Register Binding Information of TLC** 221

B.5 **Critical Register Binding for TLC** 223

B.6 **TLC Behavior Axioms** 239

B.7 **TLC Data-Path Axioms** 242

B.8 **Description of the Controller of the RTL Design** 247

B.9 **Behavior Description of the Components used in Synthesis of TLC** 252

B.10 **TLC Critical Path Equivalence Lemmas** 260

B.11 **Proof Script for Lemma eq_cp0_st_lemma** 262

B.12 **Proof Script for Lemma eq_cp1_hl_lemma** 263

C.1 **Partial Behavioral Specification of Bubble-Sort** 269

C.2 **Multiplier Component** 272

C.3 **Register Component** 273

C.4 **Random Access Memory Component** 273

C.5 **Example of Interconnection of the Data-Path Components** 274

C.6 **Parts of Controller and Data-Path of an RTL Design** 277

List of Tables

4.1 **The symbolic value function δ** . . . 37

4.2 **The state transition function σ** . . . 39

11.1 **Experimental Results** . . . 188

C.1 **Symbolic Evaluation of Specification Variables after State Transitions** . . . 268

C.2 **Symbolic Evaluation of Specification Variables along A Critical Path** . . . 271

Chapter 1

Introduction

Advances in the area of digital design have provided the engineers with the potential to build systems of unprecedented size and complexity. The tremendous growth in this area has been made possible by the development of sophisticated design tools and software which automatically build implementations of the systems at different levels from their specifications at a more abstract level. The above process known as *synthesis* reduces the design time and also human design errors. Manual design processes are still superior to the synthesis processes in terms of the performance of the final product, but the time and human expertise they require are very restrictive. Synthesis processes have gained popularity in recent years since they meet the demands of modern technology to design more efficiently and more quickly.

The rapid evolution of digital technologies has resulted in a proliferation of features but a decline in reliability [31]. Automated design procedures reduce the possibility of design errors, but they don't eliminate it altogether. With exploding complexities brought about by expanding scale, it becomes increasingly difficult to ensure that digital systems will not malfunction due to design errors. Failure of the systems due to design errors may have severe ramifications. For example, failure of the class of safety-critical systems may be catastrophic. Design errors may often be associated with high costs, too. Failure of the commercial systems may lead to irrecoverable financial loss. Unfortunately, despite the impressive advances of the computer aided design technology in recent years, computer aided verification remains at its infancy.

Today, as our lives become more dependent than ever on computer systems, the urgency for development of validation methods and verification tools that can assure correctness, safety, security and reliability of digital hardware asserts itself. This has prompted the researchers to look for firm theoretical basis for correct design of hardware systems. Mathematical methods have been developed to model the functional behavior of electronic devices and to verify, by formal proof, that their designs meet rigorous specifications of intended behavior. This approach to validating the correctness of systems is known as *formal verification*. This work presents research in formal verification of digital designs and focuses on this problem at register transfer level (RTL) of abstraction.

1.1 Motivation

Our world is increasingly dependent on digital systems. These systems often have critical applications: safety critical systems (e.g. avionics and nuclear monitoring), medical monitoring, communication networks, space probes and integrated circuits (e.g. microprocessors). The criticality of the applications of these designs demands that they perform as expected i.e. be functionally correct. Despite the overwhelming advances in design of digital circuits that has resulted in the advent of systems with unprecedented size and complexity, their reliability has declined. Unfortunately, verification technology, due to its immature state, cannot deliver a countermeasure. In digital design domain, verification technology – and in particular automated verification technology – has not progressed at the same pace as synthesis technology. The conventional post-design methods of validating functional correctness, such as testing, can no longer cope with the complexity present in today's designs within feasible amounts of testing time. Therefore, the current methods and existing tools cannot deal with the design and verification problems at the same level and with the same capacity. This issue has raised a mounting sense of urgency in the search for more systematic approaches to the verification problem. In the past decade, it has become apparent that our confidence in hardware systems can only be derived from our rigorous analysis rather than by experimentation. Formal methods currently offers the most promising vehicle to achieve this goal.

Formal methods is the applied mathematics of computer system engineering [2]. Formal methods employs mathematical logic in modeling, specification, analysis and engineering of a system and in verifying by mathematical proof that the system design and implementation satisfy system functional and safety properties. Formal verification is the use of mathematical logic for the purpose of verifying various properties of the systems or the methods applied for constructing them. Formal verification is an integral part of any successful formal design discipline. In such a setting an immediate application of formal verification is validating the correctness and integrity of design construction methods, and verifying that these methods do not violate any design rules. Formal verification is used for proving the correctness (or otherwise uncovering errors) of systems that are designed through an informal process and their functional correctness is in question.

Several characteristics are desired in a verification method:

Scalability: A number of verification methods can be effectively employed to detect and locate subtle design errors in systems of relatively small size. Unfortunately, these methods fail to scale to large systems. As an example, in BDD (Binary Decision Diagram) based methods, the complexity of the verification algorithm is an exponential function of the bit size of the variables. Consequently, these methods are severely limited by the size of the designs they can verify.

Ideally, a verification algorithm deals with the designs of arbitrarily large size. However, it is reasonable to expect that a verification tool copes well with the designs of moderate to large size, encountered in industrial-scale synthesis environments.

Soundness: A primary goal of formal verification is reducing the possibility of design errors in computer systems. Some verification exports adopt conservative approaches for error detection and argue that such approaches meet this goal. When investigating if a design is correct, a conservative method does not identify the erroneous designs as correct (it doesn't generate false positives), but it may identify some of the correct designs as erroneous (it generates false negatives). Even though such methods don't address completeness issues they are sound. However, the effort in reducing the possiblity of design errors should not be interpreted in a such a way that a verification tool identifies some of the design errors, but not others (generate false positives).

A formal verification method must be sound, i.e. it must identify all the design errors that exist in a system.

Completeness: The state-of-the-art synthesis system today, employs complex techniques (in particular sophisticated optimizations) and generates designs that have equally complex and sophisticated structural and control properties. Verification engineers have often found validation of general synthesized designs a hard problem to tackle and its solution beyond immediate reach. Consequently, they have resorted to simplifying this problem by restricting the freedom of the synthesis tools in their design choices. These restrictions sometimes limit the permitted primitive constructs (structural constructs as well as control constructs) composing the design, and sometimes the construction rules adopted in generating the design. Therefore, such methods are only effective in identifying errors in design structures of a particular (usually basic) style. An ideal verification method has a broad domain of application, and copes well with general moderately to highly complex designs.

Automation: Ideally, formal verification process should be completely automatable. Unfortunately, for large classes of problems this promise has yet to be fulfilled.

The appeal of automatic formal verification is that it's automated [28]. Formal verification is often a tedious and time consuming activity that results in tremendous strain on the verification engineer. Automating this effort reduces the verification time considerably. In addition, it minimizes the expensive human labor. Close to one half of the total man-time and CPU-time in a design cycle is dedicated to verification time. This is to say that automating formal verification process is translated to increase in productivity and shortening the time-to-market cycle.

Efficiency: Even-though certain verification methods may be used theoretically to detect and locate the errors in a generic synthesized design, they cannot deliver these results within practical verification time. Also, some verification methods may require practically indefinite memory resources (state explosion problem). It is apparent that such methods are very inefficient in practice.

An ideal verification method conducts the verification task using practical time and memory resources. This ensures that simple and short specifications are verified with minimal resources, but more importantly, that the time and memory requirements for verification of the designs of substantial size is reasonable. That is to say, a design with no complex constructs, that is specified in two to three pages, should be verified in a few minutes, and the processing time of the more complex specifications increases proportional to less complex specifications [1].

Validation: During a formal verification exercise, correctness proofs are generated automatically or manually. An ideal formal verification methodology should provide a means to validate these proof sessions through a trust worthy proof checking process[1]. Each completed proof session should be validated by the proof checking mechanism. A trustworthy proof checker validates only correct proofs. The proof checking mechanism should be independent from the reasoning mechanism. This ensures that only sound proof steps were employed in the proof process.

Current methods and existing tools for formal verification of digital systems, posses some of these features and characteristics, but lack others. As example, we can mention the successful efforts made in the area of formal verification of industrial scale microprocessors [29, 57]. Such methods are complete, sound, effective, and they often scale well, but may not be very efficient. Most importantly, the degree of the verification effort that can be automated is negligible. Model checking approaches are the example of methods that are generally sound and amenable to automation, but are not scalable, complete or efficient.

1.2 Objectives

The primary objective of this dissertation is to develop a scalable, complete, sound, and effective methodology for formal verification of digital systems at register transfer level of abstraction (RTL), that is amenable to complete automation, and in addition, its correctness is provable by an autonomous reasoning system. As the first step in achieving this objective, a coherent framework for modeling, specification, analysis and verification of RTL designs has been developed. The formal modeling, specification, and analysis procedures are developed as a side product, and to serve the purpose of verification. Nonetheless, the discussions that cover these aspects present valid arguments regardless of the context. Then, a formal verification methodology, and based on that a verification system have been developed that meet the following goals:

- The verification system has a similar capacity to a typical synthesis system in terms of the size of the designs it can manage. This system can be used for verification of the designs generated by a realistic synthesis system.

- The verification method addresses the soundness and completeness requirements. A conservative notion of correctness has been adopted in verification process. However, with respect to this interpretation of correctness, the method is sound and complete.

- The method may be exploited for verification of a relatively inclusive subset of synthesized designs. In particular, we have noted that the verification of iterative constructs (loops) and pipelined synthesized designs present interesting research topic, but are not well investigated. Loop unrolling technique is used in verification of general iterative constructs, and in synthesis of pipelined iterative constructs. However, this technique cannot be applied when the number of the iterations of the loop is not known in advance. Verification of the designs with such loop constructs is very tedious and inefficient, if not impossible and these designs are often excluded from the set of verifiable synthesized designs. We believe that such loops are very real control constructs in synthesized RTL designs, and should be dealt with during the verification process. Chapters 9 and 10 present a detailed study of pipelining in design synthesis, and discuss verification issues involving non-pipelined and pipelined loops, and pipelined synthesized designs.

- Finally, verification algorithms are developed to achieve significant improvement in processing time over the existing tools and to avoid the exponential complexities of the BDD-based methods.

1.3 Scope of Our Verification Approach

The term *synthesis* refers to the automated derivation of an implementation from a pregiven specification in a top-down manner. Synthesis is a hierarchical procedure, i.e. the implementation at one level of abstraction, is the specification at immediately lower level (Figure 1.1). In a complete synthesis process, a high-level behavioral description of the design (the specification at the highest level of abstraction) is transformed into a layout (the implementation at the lowest level).

The complete synthesis of a design from the behavior description down to the layout is in general not feasible in one step, but proceeds by the solution of a number of subproblems that are combined to solve the general problem. Each subproblem is mapped to the problem of synthesis at a particular level of the hierarchy. The complete synthesis of the design typically consists of three stages *high-level synthesis*, *logic synthesis* and *layout synthesis.*

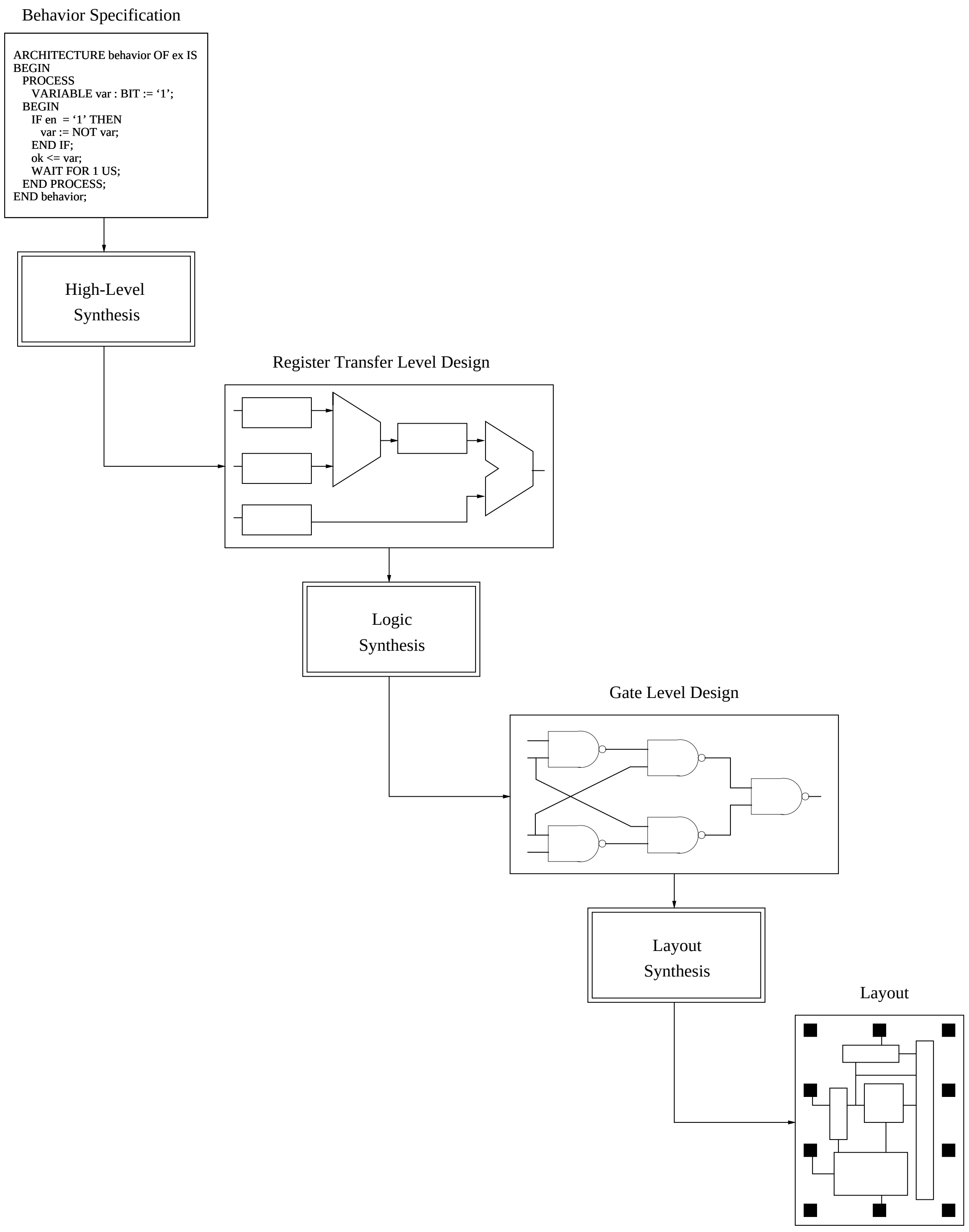


Figure 1.1: **Synthesis Hierarchy**

During the high-level synthesis process, register transfer level (RTL) implementations are derived from behavioral specifications. Logic synthesis transforms the specifications at the register transfer level into gate-level implementations. Finally, in layout synthesis, a layout implementation is extracted from a gate-level specification.

Since we are not interested in a synthesized design that does not work, it should be verified that the synthesized design implements its specification correctly. As synthesis is a hierarchical process, the verification of synthesized designs can also be performed hierarchically.

During the verification process, the implementation at each level of abstraction is compared with its specification at that level (the implementation at an immediately higher level), and is correctness is verified. If a design error at one stage of synthesis process goes undetected, the implementation in that stage as well as the implementations at all the following stages will be erroneous. Therefore, it is important to find design errors as early in the design cycle as possible. Based on this principle, in developing a formal verification methodology the logical starting point is high-level synthesis domain. This research focuses on verification of register-transfer level designs, generated through high-level synthesis. In following section an overview of this process is presented.

1.4 An Overview of High-Level Synthesis Process

High-level synthesis systems generate register-transfer level designs from algorithmic behavioral specifications (Figure 1.2) [21, 17, 61, 10, 66, 33]. The RTL design consists of a data path and a controller. The data path consists of component instances selected from an RTL component library. The controller is a finite-state machine (FSM) description subject to down-stream FSM synthesis. High-level synthesis process begins by compiling the behavior specification into a control-data flow graph (CDFG) representation. The CDFG typically consists of operator nodes representing the arithmetic and logical operators in the specification and control nodes representing the control flow operations in the specifications. The goal of the high-level synthesis system is to bind the operator nodes to arithmetic-logic units (ALUs), specification variables and data dependencies to registers and interconnect units, and control nodes to states in the controller FSM such that the user constraints on speed and cost (area) are met.

High-level synthesis process typically involves five tasks: module selection and scheduling, function unit allocation, register allocation, interconnect allocation and controller generation. Module selection and scheduling step selects a set of ALU resources from the RTL library while meeting any area constraint and schedules the CDFG across control steps such that at any control step the selected resources are sufficient to perform all the operations scheduled in that step and the clock-speed and latency constraints, if any, are met. *Binding* refers to the final assignment of behavioral operators,

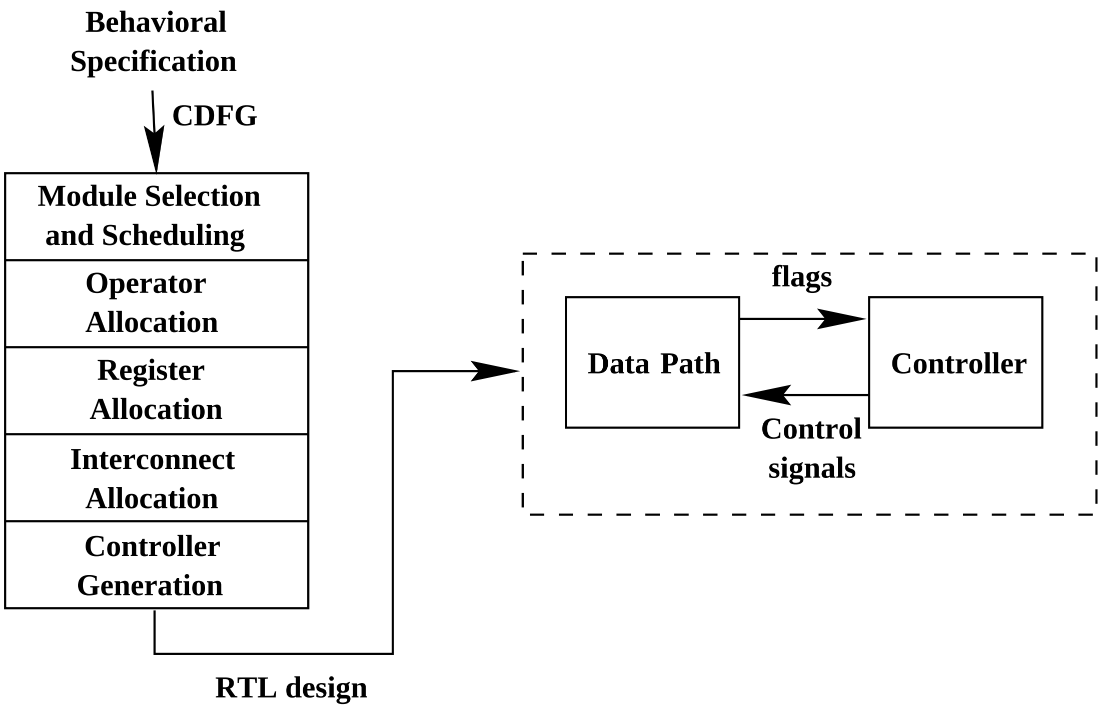


Figure 1.2: **High-Level Synthesis Process**

variables and data/control dependencies to RTL ALUs, registers and interconnect units. By maintaining links with the elements of the behavior specification throughout the high-level synthesis process (as discussed, for example, by Thomas et al. [59], the high-level synthesis tool can generate detailed binding information. As it can be seen in the following sections, this observation is the fundamental ground for our verification method.

Scheduling may be viewed as the process of code motion in compilers combined with introduction/removal of temporary variables that hold values across control steps. High-level synthesis tools in general, are oblivious to the mathematical properties of the operators used in the behavioral specification. Further, while scheduling the CDFG, high-level synthesis tools do not perform code motion across control flow operators in the CDFG such as conditional and iterative statement boundaries. Such behavioral transformations (involving transforms like constant propagation, common subexpression elimination, code motion, dead-code elimination, loop unfolding, commutative and associative rewriting of expressions, etc.) are performed prior to the commencement of scheduling in a preprocessing step called *behavior transformation* [17, 61]. Proving the correctness of such general behavioral transformations involves techniques similar to those used in program verification. McFarland studied various behavioral transformations and proposed the algebra of behavioral expressions as means to specify and verify their correctness [37].

In this research, we assume that the behavioral specification has already been transformed into the desired algorithmic form before synthesis commences. Once submitted to the synthesis system, it is not subjected to further behavioral transformation. High-level synthesis process is then predominantly concerned with resource sharing. The goal of high-level synthesis is to determine constraint-satisfying sharing among ALUs, registers and interconnections. To facilitate resource-sharing, high-level synthesis tools perform scheduling which permits time sharing of resources whose life times do not overlap across the scheduled time-scale.

During the scheduling process operations may be scheduled at any time-step as long as the data and control dependencies are not violated. That is, if an operator is data-dependent on a source operator, it can only be scheduled after the source operator is scheduled. Similarly, if an operator is control dependent on a control operator then it is not scheduled until after the control operator is scheduled. All scheduling algorithms in high-level synthesis assume that control operators introduce sequential control flow points into the CDFG being scheduled [21, 17, 61, 10, 66, 33]. For example, all operators inside a *case* statement are scheduled only after the deciding expression has been scheduled. All statements following the *case* statement are scheduled only after all the branches of the *case* statement are scheduled. All statements inside a *while* statement are schedule only after the deciding expression has been scheduled and all statements following the *while* statement would be scheduled only after the body of the *while* statement has been scheduled. This ensures that the control flow branches in the behavior specification are preserved and no new control flow branches are introduced. Scheduling, thus, is the process of implicit code motion possibly involving introduction of additional temporary variables in order to explore the design-space to determine a constraint-satisfying time-area tradeoff point.

Operator, register and interconnect allocation algorithms, which follow the scheduling step, are typically based on clique partitioning [62] or graph coloring [5]. Register allocation involves binding both the specification variables and any temporary variables introduced during scheduling to data path registers. Typically, a compatibility graph or a conflict graph is formed following a life-time analysis of the variables of the scheduled data flow graph. This graph is subjected to clique partitioning (for compatibility graphs) or graph coloring (for conflict graphs) to obtain a near-optimal register allocation subject to an interconnect cost model. Interconnect allocation is a similar procedure during which buses are formed. Operator, register and interconnect allocation algorithms perform no code motion and do not alter the control flow. Their focus is essentially on resource sharing to meet cost (area) constraints.

There are two types of register allocation schemes used in high-level synthesis: *value based* and *variable based*. In value based schemes, each instance (value) of each variable may be bound to a different register. In variable based schemes all instances of a variable are bound to the same register. The first scheme yields a mapping from values to registers and the second scheme yields a mapping from variables to registers. Value-based register allocation algorithms, such as the left-edge algorithm [36] yield optimal results, but can only be used when the behavior specification doesn't have conditional branches and loops resulting in an interval compatibility graph. In the presence of conditional and iterative constructs in the behavior specification, the life-cycles of variables form a general compatibility graph and value based register allocation cannot be used. In this case one has to use some form of clique partitioning or graph coloring depending upon whether a compatibility graph or conflict graph has been formed.

The final step in the high-level synthesis process is controller generation [17]. During this stage an FSM specification based on the CDFG schedule which determines the set of register-transfers to be activated in each control step is generated. Typically, each control step in the scheduled flow graph is turned into an FSM state, a transition is created between each control-step boundary and control signals are asserted in that state to activate all register transfers scheduled at step in the data-path. Conditional control flow nodes in the CDFG are turned into conditional states predicated on the transition conditions which are supplied from the data path in the form of condition flags.

Synthesis process and verification process are tightly connected. Each synthesis step, directly influences the elements of the verification algorithm. We will provide arguments in the following chapters to support this claim.

1.5 Background

Different approaches for verification of the RTL designs generated through high-level synthesis exist. Simulation which has been the most traditional method of verification, is being replaced with more formal methods in recent years. The rapid increase in the level of complexity of the designs, makes it impractical, if not infeasible to subject them to all possible test patterns (exhaustive testing). The use of random test patterns, on the other hand, could result in errors going undetected, thus lowering the user's confidence in the simulation exercise (random testing).

Formal verification methods rely on establishing by mathematical proof that designs are faithful to their specifications. Since the correctness proof does not depend on specific input/output values, it is not necessary to investigate all the input/output patterns, therefore, the problems with exhaustive testing does not arise in this case.

1.6 Correctness

Verification in this class of design validation methods involves furnishing a proof that an implementation 'satisfies' a specification. The notion of satisfaction has to be formalized, typically in the form of requiring that a certain formal relationship hold between the descriptions of the implementation and the specification. Various notions have been used by researchers, the semantics for each of these ensuring that the intended satisfaction relation is met:

- Implementation is equivalent to specification ($\mathsf{Imp} \equiv \mathsf{Spec}$),
- Implementation logically implies specification ($\mathsf{Imp} \Rightarrow \mathsf{Spec}$),
- Implementation provides a semantic model with respect to which specification is true ($\mathsf{Imp} \models \mathsf{Spec}$).

1.7 Classification

Research in formal verification of synthesized designs can be classified as *transformation based synthesis* (formal synthesis) and *post-synthesis verification.*

In transformation based synthesis, all the synthesis steps are formally proved by verifying the correctness of each and every transformation involved in that step. Unfortunately, even though it guarantees designs that are correct by construction, transformation synthesis is largely interactive. It requires a large degree of human expertise on the part of the verifier.

In post synthesis verification the correctness of a synthesized design is mathematically established. The verification process can be partially or fully automated and hence is usually transparent to the designer. The disadvantage of this approach is that it is severely limited by design size and often, demand computing resources that increase exponentially with design size. Model checking methods in general suffer from this disadvantage and are usually not applicable to the real size industrial designs.

1.8 Formal Verification Methods

Various proof methods can be used to formally establish the formal relationship between the specification and implementation:

Theorem Proving - Relationship between a specification and an implementation is regarded as a theorem in logic, to be proved within the context of a proof calculus, where the implementation provides axioms and assumptions from which the proof can be deduced.

Model Checking - Specification is in the form of a logic formula, the truth of which is determined with respect to a semantic model provided by an implementation.

Equivalence Checking - Equivalence of a specification and an implementation is checked, *e.g.* equivalence of functions, equivalence of finite state automata, *etc.*

Language Containment - The language representing an implementation is shown to be contained in the language representing a specification.

Any correctness relation between the specification and implementation can be established using each of the proof methods, e.g. it can be proved by model checking that an implementation logically implies the specification.

In this work, we propose a post synthesis verification methodology, which establishes the correctness of synthesized designs, by verifying that the specification and implementation of design are equivalent, through theorem proving.

1.9 Formal Proof Process

A formal proof process typically consists of the following steps:

- Adopting a general proof method and an appropriate logic framework.
- Stating the premises or the known facts about the situation in hand using the formal language of the adopted logic.
- Stating the property (or properties) to be proven using the formal language of the adopted logic (e.g. as a theorem of logic or as as boolean formula).
- Decomposition of the problem in case of a property requiring a complex proof. A complex problem may usually be split into several simpler and more manageable problems. Then the proof of each sub-problem may be independently derived. In this case an additional proof step is required to show that the proof of the sub-problems collectively imply the proof of the original problem.
- scheming a proof strategy and accordingly constructing a proof using the premises and the deduction rules of the adopted logic.

- Validating the proof by an independent proof checker to ensure the soundness of the deductive steps.

Validating a digital design by constructing a formal proof of its correctness is similar to any other formal proof procedure:

At the first step, a general proof method is adopted. The choice of a proof method to a great extent depends on the type of the design properties that need to be validated. The choice of the method in question in turn influences the choice of the logic framework. For example, in a model-checking approach temporal logic, and in a theorem-proving approach propositional logic, or higher-order logic can be the logic of choice.

The next step is to is construct a formal model of the system. This is done by converting the design into a formal description using the language of the adopted logic. This procedure is usually a compilation task. The model of the system captures those properties that are essential for establishing the correctness of the design. At the same time, this model should abstract away those details of the system that do not affect the correctness proof, but complicate the proof effort. Abstraction mechanisms may be used to eliminate unimportant or irrelevant details of the design during the modeling process. In addition, the known properties of the design may be expressed as premises, using the formal language of the adopted logic. These premises may be utilized during the proof construction process.

At the third step the property (properties) that the design must satisfy are stated in some logical formalism. The remaining steps are as sketched above. If necessary, the problem is decomposed, a strategy is schemed, and a logical proof is constructed. This proof is then validated by a proof checker.

When planning a proof exercise in general, and when planning a design validation proof procedure in particular, the possibility of automating the effort partially or completely shall be investigated. It is not yet possible to generate arbitrary complex proofs automatically. However, for specific classes of problems, where proofs are long but simple, and development of precise proof strategies is possible, the proof process may be automated.

A particular formal verification exercise usually entails tedious repetition of similar proof steps. A deep study of specific strategies used during this exercise, and extensive experimentation is required for precise identification of these similar proof steps. This also serves as an accurate assessment of the amount of reasoning effort that may be automated. One goal of this research plan is development of automated proof strategies to be used in formal verification process.

The formal verification methodology introduced in the dissertation follows these steps to prove the correctness of the designs. Chapters 3, 4, 6, 7 and 8 discuss each step in our proof process in detail.

1.10 Overview

This research introduces the notion of synthesis aware post-design verification and presents a formal approach for functional verification of synthesized RTL designs. Contributions of the research include formalization and formulation in higher-order logic in a theorem proving environment mathematical models for the register transfer level synthesized designs and their behavioral specifications and a set of sufficient correctness conditions for these designs. It presents an in depth study of pipelining in design synthesis, and identifies the complete set of correctness conditions for RTL designs generated through the synthesis processes that employ pipelining techniques. This approach has been implemented in a verification tool integrated with a high-level synthesis system. The tool performs model extraction, correctness condition generation and proof generation automatically and without user interaction.

We focus on register transfer level designs that are generated from algorithmic behavioral specifications by a high-level synthesis tool. The high-level synthesis process consists of dependency graph scheduling, function unit allocation, register allocation, interconnect allocation and controller generation tasks. Widely used algorithms for these tasks retain the overall control flow structure of the behavioral specification allowing code motion only within basic blocks. Further, high-level synthesis algorithms are in general oblivious to the mathematical properties of arithmetic and logic operators, selecting and sharing RTL library modules solely based on matching uninterpreted function symbols and constants. Many researchers have noted that these features of high-level synthesis algorithms can be exploited to develop efficient verification strategies for synthesized designs. We present discussions on how to apply the knowledge of methodical design to identify system properties that are the consequence of the construction methods, and report a verification methodology that effectively exploits these properties to achieve efficient and fully automated verification of synthesized designs.

The dissertation presents a notion of correctness of the systems in general, and RTL designs in particular based on observation equivalence of the synthesized design and its behavioral specification. We propose a decomposition for the verification problem and through a formal proof show that a set of smaller and simpler design correctness lemmas can collectively capture the correctness condition of an RTL design. Based on this decomposition the proof of correctness of a design is reduced to the proof of design correctness lemmas.

To be able to reason about the design formally, and generate the proof of correctness of the design, we model the behavior specification and its RTL implementation as extended finite state machines, then formulate these models as axioms of higher order logic. The proof of correctness of design lemmas can then be generated by simplifying it into subgoals, then proving each subgoal using the inference rules of higher order logic and design axioms.

The work introduces a verification tool developed on this basis. This tool has multiple engines to extract formal models of behavior description and its RTL implementation, and the design correctness condition lemmas. Also, as the proof of the design correctness lemmas consist of many repetitive steps, as part of the verification tool a completely automated engine for generating these proofs has been developed. This engine generates the proofs by constructing proof strategies consisting of repetitive proof steps.

The dissertation also focuses on pipelining in design synthesis. In synthesis of digital designs three different types of pipelining techniques may be used: structural pipelining, functional pipelining and loop winding. We show the usefulness of our approach in presence of structural pipelining. We show that our verification method can be extended to accommodate verification of RTL designs that are synthesized from multi-cycle pipelined or non-pipelined resources by developing the library of formal descriptions of synthesis resources in such a way that the formal descriptions of these components capture their particular behavior. This particular behavior is reflected during the verification exercise when the formal descriptions of these components are instantiated.

Also, we discuss the verification of synthesized implementations of iterative constructs in general, and pipelined iterative constructs, in particular. We discuss loop-winding, a design technique used to optimize the execution delay of a loop, and on that basis functional pipelining, a design technique used to optimize the overall execution of a general design. We show that synthesis and therefore verification of pipelined designs are different from synthesis and verification of non-pipelined designs and point out these differences through a detailed analysis. Then, we present extensions of our verification method to account for loop-winding and functional pipelining in a synthesized designs. We present a set of three correctness conditions for pipelined synthesized designs, and prove that these three correctness conditions are sufficient for verifying these deigns.

1.11 Outline of the Thesis

This dissertation thesis is organized in two parts. The first part discusses the verification of synthesized RTL designs in general. The second part of the dissertation focuses on verification of loop implementations, and pipelined synthesized RTL designs. The chapters are organized as follows: Chapter 2 presents a survey of related research. A general notion of correctness based on the behavior of the designs is presented in Chapter 3. The discussions in Chapter 4 are concentrated on modeling and formal specification of the behavior specifications that are the input of the synthesis system and the synthesized RTL implementations. In Chapter 5 we present a detailed study of high-level synthesis process, and show how the synthesis knowledge can be deployed for developing effective verification methods. Chapter 6 presents a formalization of our verification technique. Our hierarchical proof construction process is described in detail in Chapter 7. Chapter 8 introduces a formal verification tool based on the concepts presented in this dissertation. The issues in verification of special synthesized designs such as the implementation of loops or implementation of pipelined synthesized designs are discussed in Chapter 9 and Chapter 10. Chapter 11 is a presentation of the implementation issues and results, and finally, concluding remarks, some avenues for future research, and a summary of contributions are presented in Chapter 12.

Chapter 2

A Survey of Related Research

Several authors have devised techniques for verification of synthesized designs. Verification methods commonly used for verification of digital designs may be categorized as follows:

2.1 Transformation Based or Formal Synthesis Methods

Transformational approach to hardware synthesis has been pioneered by Johnson [30]. Vemuri [64] and Feldbusch and Kumar [20] proposed converting RTL implementation into a normal form where it can be compared with the behavior specification in a straight-forward manner. However, transformation into normal form seems to be possible only for restricted classes of designs. Rajan [51] addressed the same question using theorem proving by formalizing both the transformations and their correctness in the PVS theorem prover. Eisnbiegler and Kumar [19] used tight integration of high-level synthesis with theorem proving to perform synthesis and verification hand-in-hand. McFarland [37] developed *behavior expressions* for abstract specification of behavior and described how such expressions can be used to examine the correctness of behavioral transformations used in the high-level synthesis system SAW [61]. Aagaard and Leeser [3] reported a formally verified logic synthesis tool. Kropf et al. [35] presented a formal high-level synthesis approach in which the target architecture, based on handshake processes, is produced using a few basic primitives. Proof obligations are automatically generated during the synthesis process. The approach presented in this paper has a similar goal, but aims to incorporate proof obligation generation within the context of conventional target architectures used by traditional high-level synthesis tools. Narasimhan and Vemuri [46, 45] systematically formulated the correctness properties for certain high-level synthesis stages. They identified a set of assertions and invariants that should hold at various steps of high-level synthesis process. These invariants were inserted in the high-level synthesis tool DSS [53] to detect and isolate the errors in a specific run of the tool. Our verification technique follows the same goal in a different approach and is also integrated with DSS.

2.2 Post-Design Verification Methods

Hunt [29] presented a formal specification and verification of the FM8501, a simple general purpose microprocessor. In his verification exercise, a microcoded architecture is proved to implement an instruction-level description of machine behavior. Srivas and Miller [57] reported the verification of AAMP5, a pipelined commercial microprocessor. They have formalized the macro-architecture and micro-architecture in the logic of the PVS theorem prover [55]. Their correctness conditions are based on comparing the micro-architecture with the macro-architecture at *visible* states, an approach typical of processor verification efforts based on theorem proving [68]. The comparison is carried out using the rewriting strategies in PVS. An *abstraction function* that returns the macro-state corresponding to a micro-state is defined to aid the process of comparison. In our method, the critical state binding, which may be viewed as the inverse of the abstraction function, is automatically generated during synthesis. However, we have not yet applied our method to pipelined structure synthesis. These two verification exercises are conducted manually. The verified designs are far superior in terms of size and complexity to the automatically verifiable designs.

Devadas et al. [54] developed methods to verify equivalence between sequential machines at RTL and logic levels. Their method depends on extracting state transition graphs from the two finite automata exploiting *don't care* information. Corella et al. [16] verified synthesized designs by back-annotating the specification with *clock* statements according to the schedule. Claesen et al. [11, 12] developed a method to compare implementations at register-transfer and switch levels against signal flow graph specifications. Their method is based on partitioning the signal flow graph into disjoint acyclic subgraphs formed by reference signal boundaries. Then both the subgraph specification and the corresponding implementation fragment are symbolically simulated [7]. The resulting symbolic expressions are compared using standard OBDD [8] techniques. The method of identifying critical states used in this paper is comparable to SFG partitioning. However, our verification strategy itself is based on automatically generating axiomatic definitions of the behavior specification and RTL implementation and automatically generating proof scripts for theorem prover. Corella [15, 14] focussed on control flow properties by using uninterpreted functions to represent data path elements. Burch and Dill [9] used similar ideas to develop an efficient validity checker for a logic of uninterpreted functions with equality and used it to verify the control logic in pipelined microprocessors. The method discussed in this paper also uses uninterpreted functions and constants but uses standard decision procedures in a high-order logic theorem prover. Our method is developed in the context of and is fully integrated with a high-level synthesis system so that all formal models, correctness conditions and proof scripts are automatically generated as a byproduct of the synthesis process. Aagaard et al. [4] attempted hardware verification with smart tactics. They discovered that in doing hardware proofs, the user follows the same reasoning repeatedly and aimed to capture this reasoning. The verification system discussed in this paper uses the information generated by synthesis tool for generating the verification tactics. It is similar to the smart tactics method in that (1) it supports hardware verification that is used in higher order logic theorem proving, and (2) automates the repetitive reasoning found in hardware proofs. It is different from smart tactics in that (1) verification using smart tactics requires some interaction with the user, while our proofs are carried out with no user interaction; (2) as part of the proof, smart tactics method maps the design correctness goals into a set of simpler subgoals, but we decompose the main correctness goal into a set of sub-lemmas that are proved independently; and (3) our verification system tailors the proof for each individual design and can always prove the correctness of a design (if a proof exist), but this is not the case with smart tactics. Windley [67] presented an engineering methodology for verifying microprocessor designs and proposed the integration of verification tools with CAD tools as the next step in research. The work presented in this paper conforms with this idea.

2.3 Symbolic Simulation Methods

Moore [42] discussed symbolic simulation of formally specified systems. He suggested that with a symbolic simulation capability, an engineer can 'run' a design on certain kinds of in-determinant data, thereby covering more cases with one test. He used the ACL2 theorem prover for formally specifying the designs in his experiments. Also, in [25] Greve described how symbolic simulation was applied in the development of JEM1, the world's first JAVA processor. In his experiments on the application of formal methods to the verification of microcoded microprocessors he discovered that many of the errors that were ultimately detected in formal models were revealed during the symbolic simulation of the microcode, rather than during equivalence checking with an abstract specification. PVS is used for the specification of JEM1. Rajan [52] et al. studied various issues in combining theorem proving and symbolic simulation. They experimented with specification and verification of a bounded stack at differing levels of abstraction. They used HOL theorem proving at higher abstraction levels, and VOSS symbolic trajectory evaluation at the switch level. Symbolic simulation is a post-design verification approach. We categorized these methods separately since they have common features. All these methods benefit from the expressive power of theorem proving for formally specifying the designs, and by taking advantage of symbolic simulation, they automate the proof effort.

2.4 Simulation-Based Methods

Bergamaschi and Raje [6] defined a notion of behavior-RTL equivalence compatible with the specification's simulation behavior. Their technique depends on generating additional hardware to raise a synchronization signal at the points of comparison. These synchronization points, called *observable time windows*, are identified based on a notion of equivalent states which is very similar to the one used in this paper. However, in contrast to Bergamaschi's approach targeted to simulation, our method is targeted to fully-automated formal verification. No support hardware is necessary in our method.

2.5 Discussion

In general, all the developed methods for formal verification of correctness of synthesized designs take advantage of the features and limitations of the synthesis and optimization algorithms that generate design structures in a particular style. These methods mostly resort to syntactic or symbolic comparisons avoiding the combinatorial explosion problem involved in introducing more general notion of correctness based on, for example, Boolean equivalence. Our method also takes advantage of the nature of the high-level synthesis algorithms. In addition, our method is fully integrated with a high-level synthesis tool such that both the correctness theorems and their proofs are automatically produced as auxiliary outcomes of the synthesis process. These proofs are then readily processed by a higher-order logic proof checker.

Chapter 3

Correctness

The design of a digital system starts from an abstract description of its behavior (*specification*). This description specifies the operations that the system to be designed is required to perform and a set of constraints that it should satisfy. The designer or the synthesis tool then generates an architecture which realizes that behavior and meets the constraints (*implementation*). More than one structure may realize a specification, therefore a specification may have more than one implementation. The correctness of an implementation designed by a designer or generated through synthesis is usually suspect and hence should be verified. The notion of correctness of designs is a relative concept; a design is correct if it is faithful to its specification.

Various criteria for defining the correctness of a design exist, and consequently, various aspects of a design may be verified. The efforts in verification of synthesized designs are mainly categorized into two classes: performance verification efforts and functional verification efforts. While the correctness of performance of a design is verified with respect to its specified constraints, its functional correctness is verified with respect to its specified behavior.

This work is focused on functional verification of digital designs. Functional verification algorithms inspect if a design is faithful to its specified *behavior*. If the operation of a system *satisfies* its specified behavior, it is considered to be *functionally* correct. Before discussing the subject of functional correctness of the systems any further, we should precisely define the two terms “satisfy” and “behavior”.

In Section 1.5 we mentioned that the notion of satisfaction is formalized by requiring that a certain formal mathematical relation between the specification of a system and its implementation hold. One such satisfaction relation is equivalence, i.e. the implementation of a system is considered correct if it is equivalent to its behavioral specification: $(\mathsf{Imp} \equiv \mathsf{Spec})$. We have adopted this interpretation of the correctness as the basis of our formal verification approach. The equivalence relations may be defined between any two designs, whether they are at the same or different levels of abstraction, or whether they are both specifications, both implementations or else. However, in this discussion we concentrate on defining the equivalence between specification of a design and its implementation at a lower level of abstraction (even though the discussion may be easily extended to the cases mentioned above).

In this chapter we introduce two different interpretations of equivalence: *observation equivalence* and *strong equivalence.* Observation equivalence is defined based on the *observable behavior* or for short *behavior* of a design. As opposed to that, strong equivalence of the designs is defined in terms of their observable behavior as well as their internal function or *internal behavior.* Therefore, to discuss various interpretations of equivalence, first the notion of behavior of a design should be precisely defined.

3.1 Behavior

A precise notion of behavior is an essential part of a theory of complex systems. Behavior of a system is nothing more or less than its entire capability of communication [41]. "The behavior of a system is exactly what is observable, and to observe a system is exactly to communicate with it" [41]. This is the central idea of *observation equivalence.*

In contrast to observable behavior or simply the behavior of a system which is defined entirely in terms of its communication with environment, and independent from its internal structure and operation, its *internal behavior* precisely involves those aspects. The internal behavior of a system is defined by its temporal domain and its value holding elements (such as variables and registers). We can consider an abstract notion of time for a system based on the changes in its internal state. A change on the internal state of the system occurs when a new value is assigned to a value holding element or when a change in flow of control takes place (e.g. a procedure is called, or a based on the value of a variable a decision is made).

Each specification is viewed as a system. This system is internally defined by its set of value holding elements (variables). The temporal domain of the specification system consists of all the various states that the system goes through due to the operations on its variables. The internal state transitions of the specification are transparent to an observer of its behavior. In fact only those points of its temporal domain and those variables have significance in defining its behavior that at those points and through those variables it communicates with its environment.

The implementations are also viewed as systems. The structural implementations generally consist of a set of components and a control unit that determines the intercommunication of the components. The implementations temporal domain consists of all distinct states of its controller. The implementation is internally defined by its temporal domain and its components. The internal details of the implementation are transparent to an observer of its behavior. Only those points of its temporal domain and those value holding components have significance in defining its behavior that at those points and through those components it communicates with its environment.

3.2 Equivalence

Informally, two systems are equivalent, if they have equivalent behaviors. If two systems have equivalent observable behaviors they are said to be observation equivalent. If two systems have equivalent internal behaviors they are strongly equivalent. The following defines the equivalence between the two systems under the assumption that they start operation from equivalent internal states:

Definition 3.1 *Two systems are **observation equivalent** iff their communication with their environment is in the same order and with the same values.*

Definition 3.2 *Two systems are **strongly equivalent** iff their cycle to cycle internal behaviors are equivalent.*

where the internal behavior of a system at each cycle involves its internal value holding elements in addition to its communication with environment.

In high level synthesis domain the implementations are register transfer level architectures that are generated from the algorithmic specifications. There is no assurance in correct function of synthesis tool, therefore, the correctness of the implementations generated by this tool is also under question. A synthesized RTL design is considered correct if its behavior is equivalent to its specification's behavior.

Consider a specification system S. The specification S may have multiple implementations. In order to accept a system I as an implementation of the system S, it should be established that they have the same behavior. An RTL implementation I consists of a control unit and a data-path which is an interconnection of components. The components can be divided into value holding components (registers), value passing components (multiplexers and buses) and functional units. The registers are of two types *architectural registers* and *temporary registers*. An architectural register physically represents a variable in specification. The temporary registers, on the other hand, hold the intermediate results of register transfer operations and do not correspond to any variable of specification.

We assume that a graph G_s represents the operation flow of S and a graph G_i represents the control flow of the operations specified by the controller of I. Also we assume that the behaviors of S and I are mathematically formulated by the two tuples $\langle E_s, \lambda_s \rangle$ and $\langle E_i, \lambda_i \rangle$, respectively. E_s is a regular expression defined as $E_s = (in + out)^*$. λ_s is a string of values that the variables in the specification can assume, and is defined by $\lambda_s = \{val \mid val \in D_v\}^*$ and $length(E_s) = length(\lambda_s)$. The expression E_i and the string of values λ_i, describing the behavior of I are similarly defined. The elements of the expression E_s (E_i) and the string λ_s (λ_i) respectively denote the direction (input or output) of the communication of the system S (system I) with the environment and the exchanged values in each communication. For example, given the k-th element of E_i to be *out* and the k-th element of λ_i to be val_k then the k-th communication of the system I with its environment is a write operation in which it outputs the value val_k. Then under the Assumptions 3.1 and 3.2 given below the equivalence of the two systems S and I may be defined.

Assumption 3.1 *The operation environments of S and I assure that the two systems have equivalent initial internal states, i.e. the initial value of each variable in S is the same as its corresponding architectural register in I.*

Assumption 3.2 *All the states of G_s, the operation flow graph of S, are reachable from its initial state, and likewise, all the states of G_i, the operation control flow graph of I, are reachable from its initial state.*

Definition 3.3 *An implementation design I is observation equivalent to a specification S if the terms $\langle E_s, \lambda_s \rangle$ and $\langle E_i, \lambda_i \rangle$ describing their behavior are syntactically equivalent.*

The internal behavior of a system is usually defined in terms of the contents of its value holding elements (e.g. variables, registers, ...) at each state and the effects of the operations and value transfers on these values. In this discussion, we define the internal behavior of specification S at each state, based on the values of the variables present in this description. Similarly, the contents of the architectural registers of implementation I at each state, define its internal behavior. Then the strong equivalence between S and I may be equivalently phrased as:

Definition 3.4 *An implementation design I is strongly equivalent to a specification S, iff (1) I and S are observation equivalent, and (2) at the end of each operation cycle, each variable in S and its representative architectural register in I hold equivalent values.*

Discussion

When G_s and G_i are purely sequential graphs, verifying the observation equivalence of the specification S and its implementation I consists of a syntactic comparison of the regular expressions E_s and E_i and the strings of values λ_s and λ_i. However, when iterative or conditional constructs are present in G_s and/or G_i, extracting the regular expressions E_s and E_i is often difficult or infeasible. Consequently, the verification of observation equivalence between a behavioral specification and an RTL design in general is considered an intractable problem. Verification of strong equivalence between a behavioral specification and an RTL design is at least as difficult as verification of observation equivalence between them. Therefore, this problem is in general intractable also. However, for particular classes, of designs, or for designs with certain properties specific practical solutions to the problems of observation equivalence verification or strong equivalence verification may be found. In the following section we will discuss a specific case when the equivalence of the two systems may be verified.

3.3 Equivalence Checking through Decomposition

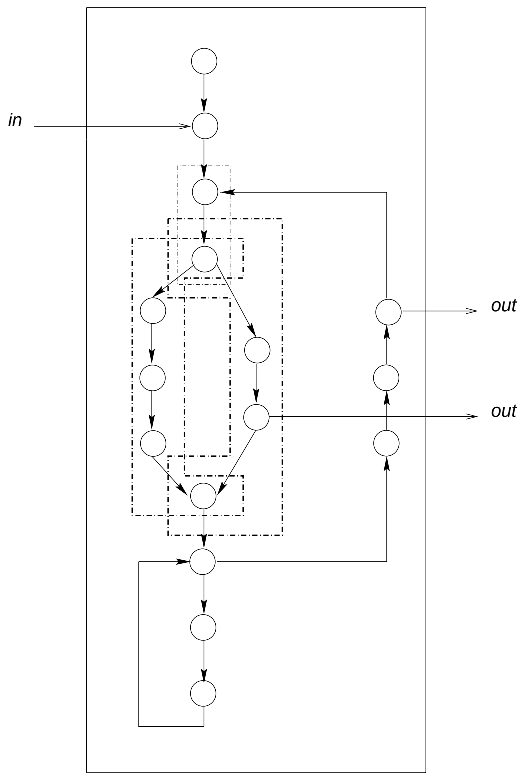
Consider a system S a single process, asynchronous specification of the behavior of an RTL design. We assume that S has m distinct variables, and its operation flow is defined by the operation flow graph G_s. G_s may be decomposed into directed acyclic subgraphs (DAGs) g_{s_1}, g_{s_2}, $\ldots$, g_{s_n} with overlapping initial and final states. The states across which G_s is divided consist of: (1) split-merge points of a conditional construct, (2) enter-exit point of an iterative construct, or (3) transfer points of a procedure. Such states only represent transfer of the control flow, and in a behavior specification no operation occurs in these states (the values of the variables do not change in such states). We can reconstruct a graph G''_s from G_s by replacing each directed acyclic subgraph g_{s_k} with a box labeled g_{s_k}. Then, each box in the graph represents a hyper-state (Figure 3.1).

Now, consider a system S' composed of n subsystems s_1, s_2, $\ldots$, s_n (Figure 3.2) such that:

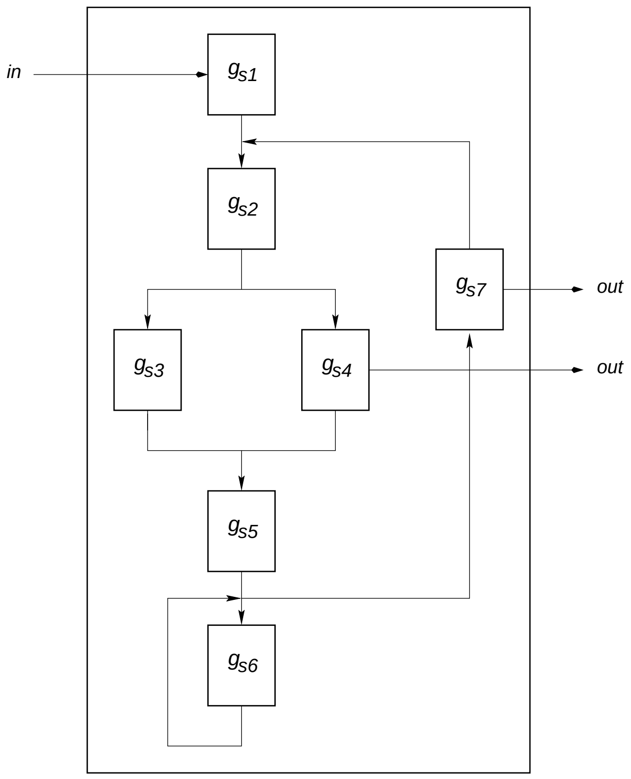
(1) Each sub-system s_k inherits the m variables of the specification S.

(2) The interface of each sub-system s_k is defined as follows: (a) in operation flow graph G_s, if at some state of the sub-graph g_{s_k} the system S reads/writes a value from/to an input/output, then the main input/output of the sub-system s_k is directly connected to the input/output of S'; (b) if in G''_s a subgraph g_{s_i} exist such that there is a directed edge from g_{s_i} to g_{s_k}, then s_k has an array of m additional inputs $in_k[m]$; (c) if in G''_s a subgraph g_{s_j} exist such that there is a directed edge from g_{s_k} to g_{s_j}, then s_k has an array of m additional outputs $out_k[m]$.

(3) The interconnection of the sub-systems of S' is defined by the interconnection of the hyper-



(a) G_s



(b) G''_s

Figure 3.1: **Constructing $\mathbf{G''_s}$ from $\mathbf{G_s}$**

states of the operation flow graph G''_s. If in the operation flow graph G''_s a directed edge from g_{s_i} to g_{s_j} exist, then the outputs $out_i[m]$ of sub-system s_i are connected to the inputs $in_j[m]$ of the sub-system s_j.

(4) At the sub-system level, the operation flow of each sub-system s_k is defined by a directed acyclic sub-graph g'_{s_k}. g'_{s_k} is identical to the directed acyclic subgraph g_{s_k} in G''_s except in two ways: (1) if in G''_s a subgraph g_{s_i} exist such that there is a directed edge from g_{s_i} to g_{s_k}, then g'_{s_k} is modified so that at its initial state, the values of all the m variables of s_k are read from the inputs $in_k[m]$, and (2) if in G''_s a subgraph g_{s_j} exist such that there is a directed edge from g_{s_k} to g_{s_j}, then g'_{s_k} is modified so that at its final state, the values of all the m variables of s_k are written to the outputs $out_k[m]$.

(5) At the system level the operation flow of S' is defined by an operation flow graph G'_s. G'_s is identical to G''_s except that each subgraph g_{s_k} of G''_s is replaced by the slightly modified sub-graph g'_{s_k} described above.

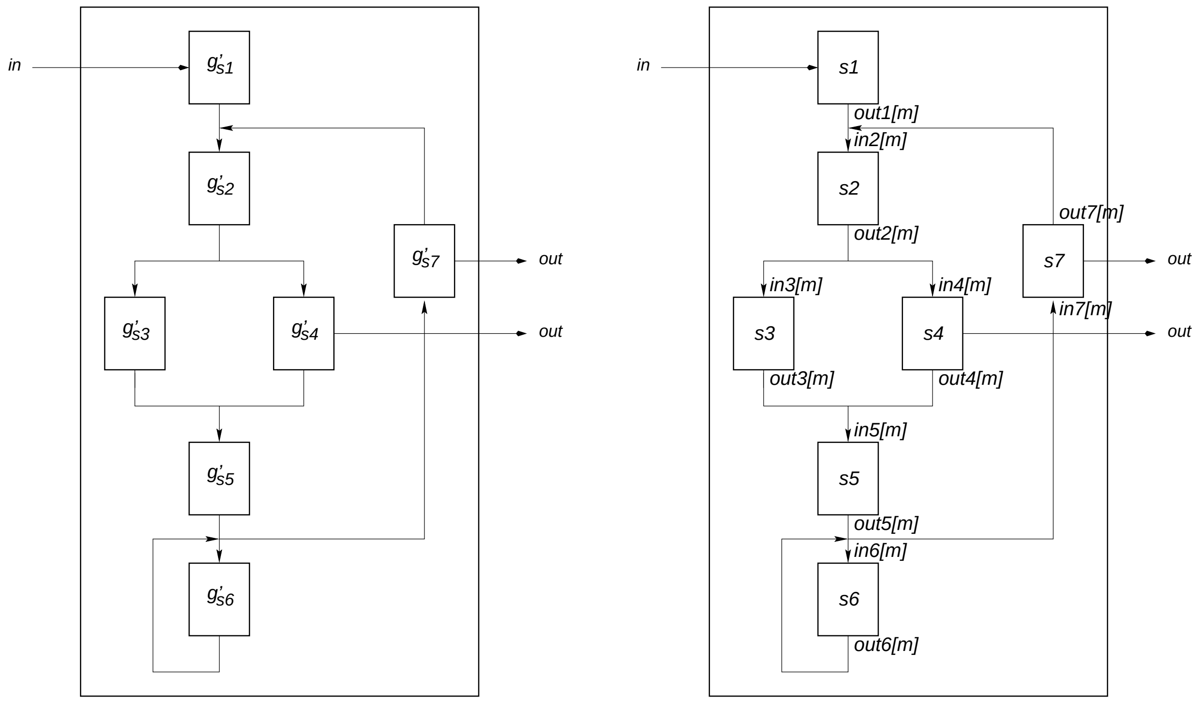
It is straightforward to prove by construction that the two systems S and S' are strongly equivalent. Since equivalence is a transitive relation, for verifying the equivalence of a system S with a system I, it is sufficient that the equivalence of S' and I be verified. Following a similar reasoning a system I' equivalent to I, the RTL implementation of S may be constructed.

We assume that I has m' distinct architectural registers, and its operation flow is defined by an operation flow graph G_i. G_i may be decomposed into directed acyclic subgraphs g_{i_1}, g_{i_2}, …, $g_{i_{n'}}$ with overlapping initial and final states. Similar to the previous case, the states across which G_i is divided consist of: (1) split-merge points of a conditional construct, (2) enter-exit point of an iterative construct, or (3) transfer points of a procedure. Such states only represent transfer of the control flow, and in a behavior specification no operation occurs in these states (the values of the variables do not change in such states). G''_i is constructed by replacing directed acyclic sub-graphs of G_i, exactly in the same way that G''_s was constructed by replacing directed acyclic sub-graphs of G_s.

Now, consider a system I' composed of n' communicating subsystems i_1, i_2, …, $i_{n'}$ (Figure 3.3) such that:

(1) Each sub-system i_k inherits a replica of the data-path of I. Then assuming that the data-path of I has m' distinct architectural registers, each subsystem i_k has also m' distinct architectural registers.

(2) The interface of each sub-system i_k is defined as follows: (a) in operation flow graph G_i, if at some state of the sub-graph g_{i_k} the system I reads/writes a value from/to an input/output,



(a) G'_s (b) S'

Figure 3.2: **Composition of S′ from Communicating subsystems**

then the main input/output of the sub-system i_k is directly connected to the input/output of I'; (b) if in G''_i a subgraph g_{i_i} exist such that there is a directed edge from g_{i_i} to g_{i_k}, then i_k has an array of m' additional inputs $in_k[m']$; (c) if in G''_i a subgraph g_{i_j} exist such that there is a directed edge from g_{i_k} to g_{i_j}, then i_k has an array of m' additional outputs $out_k[m']$.

(3) The interconnection of the sub-systems of I' is defined by the interconnection of the hyper-states of the operation flow graph G''_i. If in the operation flow graph G''_i a directed edge from g_{i_i} to g_{i_j} exist, then the outputs $out_i[m']$ of sub-system i_i are connected to the inputs $in_j[m']$ of the sub-system i_j.

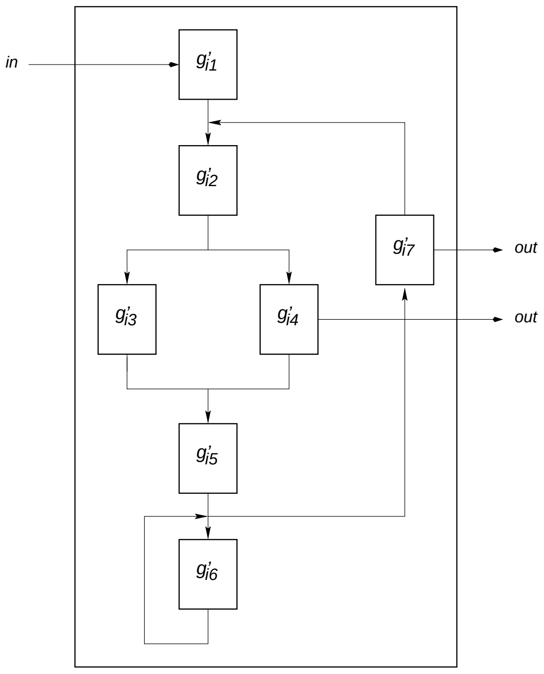
(4) At the sub-system level, the operation flow of each sub-system i_k is defined by a directed acyclic sub-graph g'_{i_k}. g'_{i_k} is identical to the directed acyclic subgraph g_{i_k} in G''_i except in two ways: (1) if in G''_i a subgraph g_{s_i} exist such that there is a directed edge from g_{s_i} to g_{i_k}, then g'_{i_k} is modified so that at its initial state, the contents of all the m' architectural registers of i_k are read from the inputs $in_k[m']$, and (2) if in G''_i a subgraph g_{s_j} exist such that there is a directed edge from g_{i_k} to g_{s_j}, then g'_{i_k} is modified so that at its final state, the contents of all m' architectural registers of i_k are written to the outputs $out_k[m']$.

(5) At the system level the operation flow of I' is defined by an operation flow graph G'_i. G'_i is identical to G''_i except that each subgraph g_{i_k} of G''_i is replaced by the slightly modified sub-graph g'_{i_k} described above.

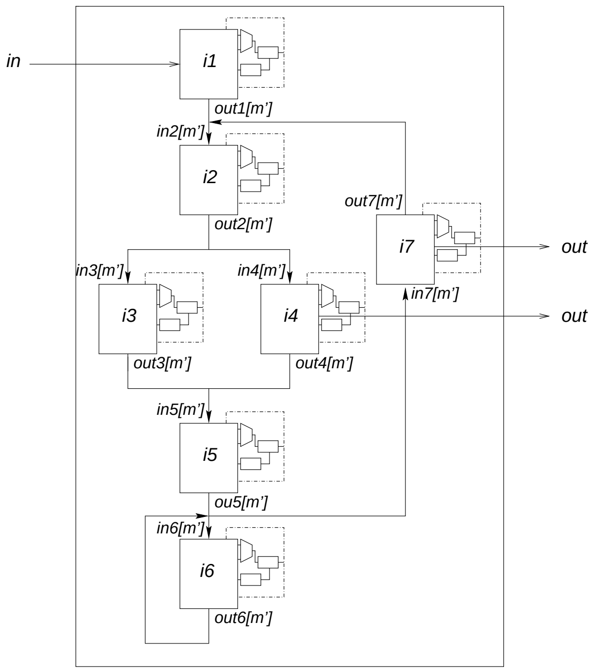
Like in the previous case, the two systems I and I' are strongly equivalent. Due to transitivity of the equivalence relation, instead of verifying the equivalence of a behavior specification S with its RTL implementation I, it is sufficient that the equivalence of the systems S' and I', constructed as described above, be verified.

As we mentioned before due to the presence of non-sequential constructs in G'_s and G'_i, extracting the regular expressions representing the behavior of these systems, and therefore, verifying the observation equivalence of S' and I' in most cases is an intractable problem. However, this problem is solvable in special cases.

Consider the case where two following verification preconditions are met: (1) $n = n'$, i.e. the mapping relation between the sub-systems of S' and sub-systems of I' is a bijection. (2) I' is implemented such that each sub-graph g_{i_k} of G'_i uniquely represent the operations described by g_{s_k} of G'_s. This means that each sub-system i_k of I' is the implementation of the sub-system s_k of S. Then by verifying the observation equivalence of the pairs of sub-systems s_k and i_k, the equivalence of S' and I' may be verified. This notion of the equivalence between S' and I' is stronger than the observation equivalence, but weaker then strong equivalence.



(a) G'_i



(b) I'

Figure 3.3: **Composition of $\mathbf{I'}$ from Communicating subsystems**

Let's assume that we can show that each pair of the sub-systems s_k and i_k are observation equivalent. If we denote the observation equivalence of two sub-systems by $\equiv$, then:

$$\forall k,\ 1 \leq k \leq n\ :\ s_k\ \equiv\ i_k$$

That is to say:

(*a*) S' and I' communicate with the environment in the same order and with the same values, then S' is at least observation equivalent to I':

$$S'\ \equiv\ I'$$

(*b*) At the final states of each pair of operation graphs g_{s_k} and g_{i_k} the values at the outputs $out_k[m]$ of s_k are the same as the values at the outputs $out_k[m]$ of i_k. This means that at the final states of each pair of operation graphs g_{s_k} and g_{i_k} the corresponding pairs of the variables of S' and the architectural registers of I' have the same values. In other words, when the operation cycle of each corresponding pair of sub-systems s_k of S' and i_k of I' terminate, S' and I' have equivalent internal states.

Now, it becomes apparent that the equivalence between S' and I' is stronger than observation equivalence. Therefore, given the mentioned verification preconditions, and the decompositions discussed by verifying observation equivalence at the sub-system level, a stronger equivalence relation at the system level may be verified.

It should be noted that in the proposed decomposition, each operation flow graph g_{s_k} and similarly each operation flow graph g_{i_k} is a directed acyclic graph, therefore, the operation flow of each sub-system of S' or I' is purely sequential. Then if a system S and its RTL implementation I satisfy the mentioned verification preconditions, by decomposing the system as suggested, extracting regular expressions describing the behavior of each sub-system s_k or i_k, and therefore verification of equivalence between S' and I' (and consequently between S and I) is always feasible.

We consider observation equivalence checking under the above mentioned preconditions an invaluable verification exercise. In the following chapters we will show that even though the required verification preconditions are very restrictive in general, they cover the verification of a broad class of synthesized implementation.

It should be noted that by analyzing the properties of S and I it is not possible to determine if a bijective mapping between the sequential sub-systems of S and I exist or not. The design properties that may effectively exploited for verification purposes are usually a consequence of the construction method, and not inherent in the design. Therefore, the study of the design construction methods is more effective than the analyzing a design in identifying such properties.

The observation equivalence checking based on the proposed decomposition and under the mentioned verification preconditions is the basis of the verification methodology presented in this dissertation. We will discuss the validity of this decomposition in a more formal setting and in more detail in the following chapters.

Chapter 4

Formal Specification and Modeling

This chapter discusses formal modeling and formal specification of the behavior specification and its RTL implementation. In previous chapters we introduced a notion of correctness for the implementation of an RTL design that is based on the observation equivalence of this implementation with its behavior specification. In this process the (observable) behaviors of the algorithmic specification and the RTL implementation are compared. In order for this comparison to be meaningful, their behavior should be mathematically formulated.

This process involves (1) constructing suitable models of the behavior specification and RTL implementation that appropriately describe their behavior, (2) selecting a logic that is expressive enough to mathematically specify the behavior of the designs, and (3) formulating the models of design in the language of selected logic. The following sections discuss these steps in detail.

4.1 Modeling

In verification domain, the model of the system should capture those properties that are essential for establishing the correctness of the design. At the same time, this model should abstract away those details of the system that do not affect the correctness proof, and at the same add to the complexity of proof generation task. Abstraction mechanisms may be used to eliminate unimportant or irrelevant details of the design during the modeling process. In addition, the known properties of the design may be expressed as premises, using the formal language of the adopted logic. These premises may be utilized during the proof construction process.

The nature of abstract specification - that is a description of behavior - and the structural implementation - that is a physical realization of that behavior, are inherently different. The specification of an RTL design is an algorithm-like description. Its implementation, on the other hand, is a description of an architecture. During the synthesis process, the behavioral specification goes through many transformations, so that its RTL implementation that is the final result of synthesis, bears little resemblance to it. In order to compare these two completely different entities, a common ground for their comparison should be found. It is part of the modeling task to identify common properties of these entities and provide the required common ground.

We have aimed at constructing models that specify a system as a relation between the sequence of values on its inputs and outputs. This modeling is suitable for our verification purpose, since it contains properties of the design that are essential for observation equivalence checking, while it abstracts away those aspects of the design that bear no relevance to the verification exercise.

The behavior specification is actually nothing but a description of the sequence of the operations that a correct design performs in their particular order. Some of these operations are conditional, repetitive or $\cdots$ which are specified using various behavior constructs.

The RTL design consists of a data-path and a controller. The register transfer operations performed in the RTL implementation, as well as the order in which they are performed, are determined by the controller. The controller of an RTL design is usually modeled by a finite state machine (FSM).

In Chapter 3 we mentioned that the flow of operations in a behavior specification may be captured by an operation flow graph. Also, we mentioned that the control flow of the operations in an RTL design may be captured by an operation control flow graph. This suggests that extended finite state machines are good candidates for modeling the operation flow of these designs. In the specification model, the finite state machine in conjunction with the set of statements define the behavior. In the implementation model, the finite state machine in conjunction with the data-path define the behavior. This is the reason they are referred to as extended finite state machines.

4.2 Specification

In high-level synthesis framework the behavior description of the design as well as the synthesized RTL design are usually specified using a hardware description language (HDL) like Verilog or VHDL. Also, the behavior of each component of high-level synthesis resource library is described in a hardware description language. It is often not possible provide comprehensive formal semantics definitions of the HDLs. Besides, it is often not possible to formally reason about the designs specified in such languages. Therefore, to develop a successful verification exercise, it is necessary to translate the HDL description of the design into a formal specification medium.

Formal specification is the use of mathematical logic as a notation for describing the systems precisely. The main benefit of formally specifying a system is intangible - gaining a deeper understanding of the system being specified. It is only through this specification process that the design engineers can evaluate and predict the behavior and performance of a system prior to its implementation and/or automatically synthesize implementations from those models. They also serve as a tool for uncovering design flaws, inconsistencies, ambiguities and incompletenesses. Industrial scale experiments have shown that formal specifications can aid in the overall improvement in the quality of the products, a reduction in the number of design errors discovered, and earlier detection of errors found in the design process.

A rigorous and descriptive formal language that can mathematically describe the behavior specification and RTL design is a key element in development of a formal verification framework. In addition to modeling the digital components, this language is used to specify various design properties of hardware descriptions, and facilitates the synthesis and verification process.

4.3 Logical Framework

One of the first steps in a formal verification exercise is selecting a logical framework. The choice of the logical framework to a great extent depends on the proof method, as well as design properties that need to be verified. The logic of choice should accommodate formal specification of the design as described above. This logic should have the expressive power to mathematically model the behavior description and RTL design, and to correctly formulate the design properties that are essential for proving the correctness. In addition, the set of inference rules of this logic should be rich enough to allow construction of the proof of correctness, from the known facts about the design, if one such proof exist.

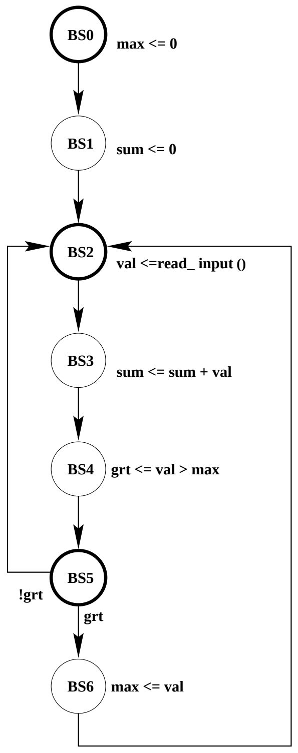
In this work we have used *higher order logic* for modeling the RTL design and its behavior specification and for proving that they are observation equivalent. Higher order logic can be used both as a hardware description language and as a formalism for proving that designs meet their specifications [22]. Is has been demonstrated that higher-order logic is a formalism in which a wide variety of behavior and structure can be specified [22]. Higher order logic makes the results of general mathematics available, and this allows one to construct any mathematical tool needed for the verification task in hand [39]. Its expressive power permits hardware behavior to be described directly in logic; a specialized hardware description language is not needed. Besides, the inference rules of the logic provide a secure basis for proofs of correctness; a specialized deductive calculus for reasoning about hardware behavior is not required [39].

```
variables max, sum, val, grt;

max := 0;
sum := 0;

repeat
    val := read_input();
    sum := sum + val;
    grt := val > max;
    if grt then max := val;
forever;
```

(a) **Algorithmic Specification**



(b) **Behavioral Automaton**

Figure 4.1: **A Behavior Specification**

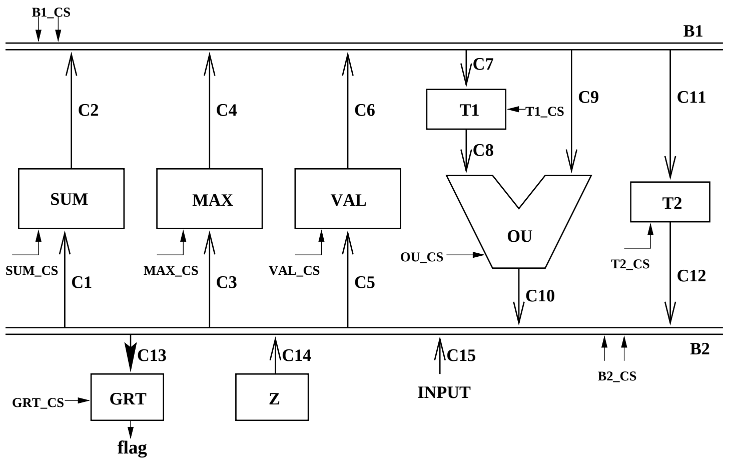
s_b	Next s_b	max	sum	val	grt
bs_0	bs_1	$'ZERO'$	$sum(bs_0)$	$val(bs_0)$	$grt(bs_0)$
bs_1	bs_2	$max(bs_1)$	$'ZERO'$	$val(bs_1)$	$grt(bs_1)$
bs_2	bs_3	$max(bs_2)$	$sum(bs_2)$	$input(bs_2)$	$grt(bs_2)$
bs_3	bs_4	$max(bs_3)$	$sum(bs_3) + val(bs_3)$	$val(bs_3)$	$grt(bs_3)$
bs_4	bs_5	$max(bs_4)$	$sum(bs_4)$	$val(bs_4)$	$(val(bs_4) > max(bs_4))$
bs_5	bs_6	$max(bs_5)$	$sum(bs_5)$	$val(bs_5)$	$grt(bs_5)$
bs_5	bs_2	$max(bs_5)$	$sum(bs_5)$	$val(bs_5)$	$grt(bs_5)$
bs_6	bs_2	$val(bs_6)$	$sum(bs_6)$	$val(bs_6)$	$grt(bs_6)$

Table 4.1: **The symbolic value function δ**

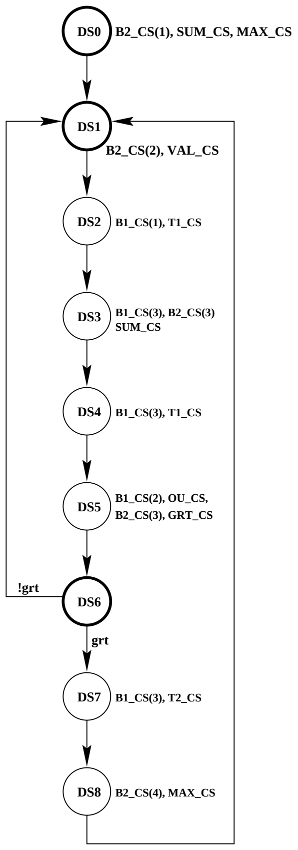
4.4 Behavior Model

A *behavior automaton* models the behavioral specification. This automaton is an extended finite state machine that is represented as a five-tuple $(S_b, S0_b, R_b, \delta_b, \sigma_b)$. In this model S_b is the set of the states of the automaton and $S0_b$ is the initial state of the automaton. A state may be an *assignment* state or a *conditional* state. An assignment state is annotated with one or more assignment statements and has exactly one outgoing transition to a *next* state. A conditional state is not annotated with any assignment statements and has exactly two outgoing transitions, one of which is labeled with the condition v and the other is labeled with the condition $\neg v$, where v is a specification variable. $R_b = \{r_{b_1}, r_{b_2}, \cdots, r_{b_m}\}$ is the set of the behavior registers or specification variables. σ_b is the next state or state transition function, that defines the next state in terms of current state and condition flag. δ_b is the value function, which symbolically defines the value of each behavior register after each state transition. Each state of the behavior automaton is labeled with a statement of the behavioral specification. The initial state has a particular significance in verification, since the observation equivalence preconditions are defined in terms of the initial values of variables at this state.

A simple behavioral specification and its representation as a *behavioral automaton* are shown in Figures 4.1.a and 4.1.b respectively. In this example $S_b = \{bs_0, bs_1, bs_2, bs_3, bs_4, bs_5, bs_6\}$, $S0_b = bs_0$, $R_b = \{max, sum, val, grt\}$, and the functions δ_b and σ_b for this example, are shown in Tables 4.1 and 4.2 respectively.



(a) **Data Path**



(b) **Controller Automaton**

Figure 4.2: **Example of an RTL design generated by a high-level synthesis system**

Current s_b	max	sum	val	grt	Next s_b
bs_0	-	-	-	-	bs_1
bs_1	-	-	-	-	bs_2
bs_2	-	-	-	-	bs_3
bs_3	-	-	-	-	bs_4
bs_4	-	-	-	-	bs_5
bs_5	-	-	-	ONE	bs_6
bs_5	-	-	-	ZERO	bs_2
bs_6	-	-	-	-	bs_2

Table 4.2: **The state transition function σ**

4.5 Design Model

An RTL implementation consists of a controller and a data-path. The controller is a finite state machine that interacts with the data path through *control signals* and *flags*. The controller extended finite state machine models the implementation. This automaton is represented with a five-tuple $(S_d, S0_d, R_d, \delta_d, \sigma_d)$. In this model S_d is the set of the states and $S0_d$ is the initial state of the automaton. There are two types of states: *assignment* and *conditional*. An assignment state is annotated with the control signals that must be asserted 'high' in that state and has exactly one outgoing transition. We assume that the remaining control signals must be asserted 'low' in that state. A conditional state has exactly two outgoing transitions; one of these transitions is annotated with the condition f and the other is annotated with the condition $\neg f$, where f is a status flag from the data path (the output signal of one of the data-path registers). [1] Conditional states have no control signal annotations implying that all control signals must be held 'low'. In particular, this means that no data path register will be loaded in conditional states. $R_d = \{r_{d_1}, r_{d_2}, \cdots, r_{d_{m'}}\}$ is the set of all design registers. σ_d, the next state or state transition function of the controller defines the next state in terms of the current state and condition flags. δ_d is the value function that symbolically defines the value of the behavior registers after each state transition.

Figure 4.2 shows a register level design generated by a simplistic high-level synthesis system. This design is obtained by a straightforward translation of the behavior constructs to basic structures implementing them without performing any scheduling or register optimization during the translation. The data path consists of a number of registers, an ALU, two buses and a number of wires. Each data path component may have one or more control signals. For registers, we assume that

[1]In this discussion we assume conditional states with exactly two outgoing transitions in both the behavior automaton and the RTL controller. This is easily generalizable to include conditional transitions based on multiple flags.

the 'load' control is active 'high.' This design is an implementation of the specification of Figure 4.1. In this example $S_d = \{ds_0, ds_1, ds_2, ds_3, ds_4, ds_5, ds_6, ds_7, ds_8\}$, $S0_d = ds_0$, $R_d = \{MAX, SUM, VAL, GRT\}$, and the functions δ_d and σ_d are defined similar to δ_b and σ_b.

4.6 Formal Specification of the Behavior

We discussed that the behavior automaton is modeled as an extended finite state machine. Each state of this finite state machine is either a conditional state, or an assignment state. A conditional state of the behavior automaton has two outgoing transitions. No operations are performed at a conditional state and only a decision about the transfer of control to one of the two possible next states is made. Therefore, in a state transition from the conditional state s_{b_i} to s_{b_j} the all specification variables maintain their values.

In an assignment statement s_{b_k} that corresponds to a statement in the form of ($r_b = exp;$) the value of the expression *exp* is assigned to the variable r_b. Then, in a transition from an assignment statement s_{b_k} to s_{b_j} all variables except for r_b maintain their values. The value of r_b at state s_{b_j} is defined by *exp*, where *exp* is a function of the values of the variables at s_{b_k}.

It can be noted that the behavior of the specification may be uniquely defined as a relation between the values of the variables after each state transition and their values prior to the state transition. It is straight forward to extract a relation between the outputs and inputs of the design.

Such relations may be extracted from the behavior automaton and translated into the predicates of higher order logic. Consider the transition $\langle BS_5, BS_6\rangle$ in the example of Figure 4.1. Since BS_5 is a conditional state, in transition $\langle BS_5, BS_6\rangle$ the values of all the variables are maintained. This may be described by an axiom such as:

$$\begin{aligned} bs_5\_bs_6\_trans\_ax : transition(bs_5, bs_6) \;\Rightarrow\; & (V_b(val, bs_6) = V_b(val, bs_5) \;\wedge \\ & V_b(max, bs_6) = V_b(max, bs_5) \;\wedge \\ & V_b(sum, bs_6) = V_b(sum, bs_5) \;\wedge \\ & V_b(grt, bs_6) = V_b(grt, bs_5)) \end{aligned}$$

BS_3 on the other hand, that corresponds to the statement $sum \Leftarrow sum + val$, is an assignment state. Therefore, in a transition such as $\langle BS_3, BS_4\rangle$ the values of all variables except *sum* that at state BS_3 is the target of an assignment will remain the same. The value of *sum* is defined by the expression assigned to it. This information may be translated to the following axiom:

$$\begin{aligned} bs_3\_bs_4\_trans\_ax : transition(bs_3, bs_4) \;\Rightarrow\; & (V_b(val, bs_4) = V_b(val, bs_3) \;\wedge \\ & V_b(max, bs_4) = V_b(max, bs_3) \;\wedge \\ & V_b(sum, bs_4) = sum(V_b(sum, bs_3), V_b(val, bs_3)) \;\wedge \\ & V_b(grt, bs_4) = V_b(grt, bs_3)) \end{aligned}$$

4.7 Formal Specification of the Design

An RTL design consist of a data-path and a controller. The register transfer operations occur in the data-path and are controlled by the controller. The controller and data-path communicate through the control signals and flags. The data-path is an interconnection of basic components. The control signals are the inputs to the control inputs of data-path components. At each state of the controller a set of control signals are asserted high. As a result a part of data-path circuitry is activated a some register transfer operations take place.

To formally describe the behavior of the RTL design, (1) the relational behavior of each RTL component, (2) the type and interconnection of RTL components, (3) the flow of state transitions in the controller, and (4) the control signals that are asserted high at each state should be formulated in higher-order logic.

Register_ax : $\forall$ $(in :$ $signal,$
$out :$ $signal,$
$ld :$ $bool\_signal,$
$s_1 :$ $RTL\_state,$
$s_2 :$ $RTL\_state) :$

$$\begin{aligned} (register(in, ld, out) \;\wedge\; transition(s_1, s_2)) \;\Rightarrow \\ (V_d(ld, s_1) \;\Rightarrow\; V_d(out, s_2) = V_d(in, s_1) \;\wedge \\ \neg V_d(ld, s_1) \;\Rightarrow\; V_d(out, s_2) = V_d(out, s_1)) \end{aligned}$$

Figure 4.3: **Relational description of the behavior of register component**

```
a2 : adder
   PORT MAP (a2.in1, a2.in2, a2.out);
```

Figure 4.4: **HDL description of the port-map of the multiplier component**

4.7.1 RTL Component Specification

The behavior and port map of each component that may be used in synthesizing RTL designs is described in a hardware description language in the high-level synthesis resource library. As part of formal specification of the design the behavior of each component is translated into predicates of higher-order logic. As an example, the behavior of a simplistic register component has been formally described in Figure 4.3.

4.7.2 Type and Interconnection of RTL Components

High-level synthesis systems usually generate the RTL design as a data-path and a controller described in a hardware description language. As part of the formal specification of the design behavior, this description is translated into higher order logic. During this process, the type and interconnection of the components are formally specified. For example, the port map of an adder component and an instantiation of it in a pseudo-HDL language are given in Figure 4.4. The instantiation of the port map of adder component given in this figure, describes the interface of an adder component labeled a_2 in the data-path of the RTL design. The same information can be conveyed more formally by the following predicate:

$$A_2\_ax \quad : \quad adder(A_2.in_1, A_2.in_2, A_2.out)$$

4.7.3 RTL Controller

Two types of information can be extracted from the HDL description of the controller: the state transition sequence and the control signals that are asserted at each state. This information can be simply described by two functions *next_state* and *control_signal*. The function *next_state* represents the function σ_d in implementation model. It defines the next state transition in terms of the contents of critical registers at each state. *control_signal* maps each control line of the design to a value 'true' or 'false' at each state. This in turn defines the values of the control inputs of a component at each state. For example, the HDL description of the controller may define the values of the control signals at a state ds_3 as a bit-stream where each bit represents the value of one control line at state ds_3. This is translated into a conjunction of predicates such as the following:

$$Control\_Signal(cs_9, ds_3) = \text{“}T\text{”}$$

Now let's assume that cs_9 is connected to the load signal of a register R_5. This is described by the following axioms, in the formal description of the data path:

$$(1) \quad register(R_5.in, R_5.ld, R_5.out)$$
$$(2) \quad R_5\_ld\_ax \;:\; (\forall s_d : RTL\_state \;:\; V_d(R_5.ld, s_d) \;=\; Control\_Signal(cs_9, s_d))$$

Now let's assume that based on the description of *next_state* we know that the next state of ds_3 is ds_4 regardless of the values of the registers. This means that the transition $\langle ds_3, ds_4\rangle$ is unconditional. This can be described by the following axiom:

$$(3) \quad transition(ds_3, ds4) = \text{“}T\text{”}$$

Axioms (1), 2 and (3) together with the $Register_a x$ axiom given in Figure 4.3 infer the following:

$$V_d(R_5.out, ds_4) = V_d(R_5.in, ds_3)$$

During the process, similar inference rules are used in conjunction with the axioms such as the above to prove various properties of the design.

4.8 Conclusion

In this chapter we showed how the behavior specification and the RTL implementation of a design may be mathematically modeled. We saw that the control flow of the operations in behavior specification and the RTL design may be modeled by finite state machines. These models as well as the specification of the data-path of the RTL design and the behavior description of each data-path component may be translated into axioms of higher-order logic. The behavioral and structural descriptions of the design in higher-order logic can be used during the verification process. The inference rules of this logic can be used in conjunction with these descriptions to prove various properties of the design. In the following chapters we will show how these formal descriptions are used to construct the proof of correctness of the design.

Chapter 5

Deploying Methodical Design Construction Knowledge for Identifying Design Properties

In this chapter we discuss how the knowledge of the construction method (synthesis process) can be used to extract design properties that can be employed for verification purposes. With the knowledge of the synthesis process, more information about the control and structural properties of the design will be available to verification engineers than otherwise possible. These properties may be supplied to the verification algorithm during the verification process in the form of assumptions or premises. Such properties have a tangible impact on the outcome of the proof exercise.

A behavior specification may have numerous RTL implementations. A synthesized RTL design is a transformed version of the behavior specification. Therefore, the relational properties of each RTL implementation and the behavior specification are influenced by the specific transformations and particular algorithms employed in synthesizing that implementation. Also, the structural and control properties of each implementation are influenced by particular procedures employed in generating it.

It is often the case that the designs that are generated followind the same synthesis steps have common properties. Hence, the properties of an RTL synthesized design are determined by its construction method, and we can learn more about these properties by a study of synthesis processes rather than by analysis of the structure and operation flow of the design in isolation from synthesis. We believe that the restrictions of most post-synthesis verification methods results from the fact that not enough information about the design is available. To remedy this problem we have adopted a synthesis aware post-design verification approach.

In the following sections, we present a detailed study of the high-level synthesis process that produces the synthesized designs at the register transfer level of abstraction from behavioral specifications. We will identify the control and architectural properties of the synthesized RTL designs and the relational properties of such a design and its specification that may facilitate their verification process.

5.1 An Overview of High-Level Synthesis Process

High-level synthesis systems generate register transfer level designs from behavior descriptions. [1] In different stages of the synthesis process the behavior description goes through many transformations. Various methods for synthesizing RTL designs, and numerous synthesis algorithms and transformations have been developed. The distinguishing characteristic of a synthesis system is the type of the algorithms and transformations it employs.

The synthesis system generates a control-data flow graph (CDFG) from the behavior specification. An abstract model of the behavior at the architecture level defines a set of *operations* and their *dependencies*. In the synthesis jargon these operations are referred to as *tasks*. The operation dependencies are due to several reasons: (1) availability of data - An input to an operation may be the result of another. In this case the former operation depends on the latter. (2) serialization constraints - A task may have to follow another regardless of data dependencies. An example is loading the data on a bus and then raising a flag. (3) resource sharing - two tasks may need to share the same resources, in which case one has to finish before the other starts.

The operations and their dependencies may be represented by a control-data flow graph. A CDFG is a directed graph $G(V, E)$ whose vertex set V is in one to one correspondence with the set of tasks. The directed edge set E is in correspondence with the transfer of data from one operation to another or other dependencies such as task serialization.

During the high-level synthesis process the operations(tasks) are bound to arithmetic/logic operators, the specification variables to registers and interconnect units and the vertices of the CDFG to a control FSM which represents the control-unit. The classical High-level synthesis process involves five tasks: module selection and scheduling, function unit allocation, register allocation, interconnect allocation and controller generation.

A predominant concern in real high-level synthesis systems is resource sharing. The goal of high-level synthesis is to determine constraint-satisfying sharing of ALUs, registers and interconnections.

[1] Resources [21, 17, 40, 63] have been a valuable source of information in preparing the material of this chapter. In particular I have adopted the material for scheduling from [17] and used the discussions presented in [40] for preparing the section on bit-level interconnect allocation.

To facilitate resource-sharing, high-level synthesis tool performs scheduling which permits time sharing of resources whose life times do not overlap across the scheduled time-scale.

The scheduling process may be viewed as code motion across a time scale. Operations may be scheduled at any time-step as long as the data and control dependencies are not violated. That is, if an operator is data-dependent on a source operator, it can only be scheduled after the source operator is scheduled. Similarly, if an operator is control dependent on a control operator then it is not scheduled until after the control operator is scheduled. All scheduling algorithms in high-level synthesis assume that control operators introduce sequential control flow points into the CDFG being scheduled [21, 17, 61, 10, 66, 33]. For example, all operators inside a *case* statement are scheduled only after the deciding expression has been scheduled. All statements following the *case* statement are scheduled only after all the branches of the *case* statement are scheduled. All statements inside a *while* statement are scheduled only after the deciding expression has been scheduled and all statements following the *while* statement would be scheduled only after the body of the *while* statement has been scheduled. This ensures that the control flow branches in the behavior specification are preserved and no new control flow branches are introduced. Scheduling, thus, is the process of implicit code motion possibly involving introduction of additional temporary variables in order to explore the design-space to determine a constraint-satisfying time-area tradeoff point.

The three tasks of function unit allocation, register allocation and interconnect allocation constitute the *resource allocation* stage of synthesis. *Resource binding* refers to the final assignment of behavioral operators, variables and data/control dependencies to RTL ALUs, registers and interconnect units at this stage. By maintaining links with the elements of the behavior specification throughout the high-level synthesis process, the high-level synthesis tool can generate detailed binding information. Operator, register and interconnect allocation algorithms perform no code motion and do not alter the control flow.Operator, register and interconnect allocation algorithms, are typically based on clique partitioning [62] or graph coloring [5] following life-cycle analysis of the scheduled flow graph.

These tasks may be performed in different order in different synthesis tools. Also different synthesis tools may use different algorithms with various level of complexity to perform each task. More sophisticated synthesis algorithms perform more elaborate transformations. Consequently, the final RTL implementation has very little in common with its specification. In such a case identifying common points of comparison between the two designs requires great effort.

5.2 Impact of Scheduling on Verification Methodology

The inputs to the synthesis system are the CDFG, detailed description of the *resources* being used (arithmetic and functional units, steering logic, storage elements and control logic) and a set of constraints (area, latency, cycle-time).

Each operation in the CDFG has an execution delay. The *start time* of an operation is the time at which the operation starts its execution. *Scheduling*[2] is the task of determining the start times, subject to the precedence constraints specified by the CDFG. A scheduled control-data flow graph is a vertex-weighted CDFG, where each vertex is labeled by its start time. A schedule may have to satisfy timing and/or resource usage constraints. The goal of scheduling is to optimize the scheduled CDFG by taking advantage of possible concurrency of the operations.

Scheduling algorithms can be divided into two groups: unconstrained scheduling and constrained scheduling. In constrained scheduling the number of resources that can be concurrently used are limited. In unconstrained scheduling, as the name suggests, there are no constraints on the number of the resources that can be concurrently used.

Suppose $G(V, E)$ is a scheduled CDFG. $V = \{v_i : 1 \leq i \leq n\}$ is the vertex set which is in one-to-one relation with the set of operations, $E = \{\langle v_i, v_j \rangle : 1 \leq i \leq n, 1 \leq j \leq n\}$ represent the dependencies, and $T = \{t_i : 1 \leq i \leq n\}$ the start time for the operations, i.e. the cycles in which the operations start. Suppose $D = \{d_i : 1 \leq i \leq n\}$ is the set of operation *execution delays*, and suppose n_{res} is the number of resource types. Then the unconstrained scheduling problem can be formally defined as:

> Given V, E and D, find an integer labeling of the operations $L : V \mapsto Z^+$ such that $\forall i, j, (\langle v_i, v_j \rangle \in E) : t_i = L(v_i) \;\wedge\; (t_i \leq t_j + d_j) \;\wedge\; t_n$ is minimum [17].

In the constrained scheduling, the number of resources of any given type is bounded from above by a set of integers $\{a_k : 1 \leq k \leq n_{res}\}$. Suppose we denote by function $TT : V \mapsto \{1, 2, \cdots n_{res}\}$ the unique resource type that implements an operation. Then the constrained scheduling problem can be formally defined as:

> Given V, E, D and upper bounds $\{a_k : 1 \leq k \leq n_{res}\}$, find an integer labeling of the operations $L : V \mapsto Z^+$ such that $\forall i, j, (\langle v_i, v_j \rangle \in E) : t_i = L(v_i) \;\wedge\; (t_i \leq t_j + d_j) \;\wedge$ $|\{v_i : TT(v_i) = k \wedge t_i \leq l \leq t_i + d_i\}| \leq a_k$ for each operation type k and schedule step $l \;\wedge\; t_n$ is minimum [17].

[2]Part of the material in this section is directly adopted from [17].

CDFG is a hierarchical graph. It is composed of a hierarchy of four types of constructs, referred to as blocks: (1) sequential construct that is a subgraph of CDFG where each vertex has one preceding edge and/or one succeeding edge; (2) a conditional construct; (3) an iterative construct; and (4) a procedure construct. Each block of the CDFG is in run composed of a hierarchy of these four types of synthesis blocks. A block at the innermost level of the hierarchy is called a *basic block* and is one of the following four types of constructs: (1) a *sequential basic block*, that is the largest subgraph of the CDFG where each vertex has one preceding edge (except the first vertex of the block which has no preceding edge) and one succeeding edge (except the last vertex of the block which has no succeeding edge), (2) a *conditional basic block*, or a subgraph of CDFG that represents a conditional construct of the behavior model. Each branch of a conditional basic block is a sequential basic block, (3) an *iterative basic block* or a subgraph of CDFG that represents an iterative construct of the behavior model. The body of an iterative basic block is a sequential basic block, or (4) a *procedure basic block*, or a subgraph of CDFG representing a subprogram. The body of a procedure basic block is a sequential basic block. A CDFG is considered *non-hierarchical* if all of its blocks are basic blocks, otherwise it is considered *hierarchical.*

Numerous scheduling algorithms have been developed. The scheduling problem of the hierarchical CDFGs is more complex than the scheduling problem of non-hierarchical CDFGs. The scheduling of both non-hierarchical and hierarchical CDFGs involve the scheduling of the basic blocks. The following sections discuss the scheduling of the basic blocks.

Sequential Basic Blocks - A sequential basic block is a sequence of consecutive operations, and is usually represented by a directed acyclic subgraph, such as the one shown in Figure 5.1. The scheduling of a sequential basic block consists of assigning time steps to the operations, by placing the operations that are data and control independent from one another in the same time step (parallel execution of the operations), in order to achieve the minimum execution delay across the CDFG, while satisfying all possible constraints. In this process two no-operation vertices are added to the block as the initial and final vertices of the block. The initial state is marked with a 'zero' execution delay before the scheduling starts. After scheduling the block, parallel branches may be generated, which represent the operations that can be executed concurrently. An edge is placed from initial state of the block to the initial vertex of each concurrent branch and an edge is placed from the final vertex of each concurrent branch to the final state of the block. If $\langle v_i, v_j \rangle$ is an edge of a branch of the basic block, t_i is the time-step assigned to v_i and t_j is the time-step assigned to v_j, and the execution delay of the operation associated with v_i is d_i, then we can say $t_j = t_i + d_i$. The execution delay of the block is the time-step assigned to the final state of the block and is the maximum time-step assigned to a vertex of the block.

For example, if we assume that in Figure 5.1 the operation OP_2 depends on the result of the operation OP_1, the result of the operation OP_2 is an operand of the operation OP_3, the operations

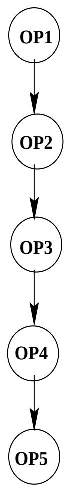


Figure 5.1: **A Sequential Basic Block**

OP_4 and OP_5 are independent from other operations, and the execution delay of the operations are t1,t2,t3,t4 and t5, respectively, then a possible schedule for this block is as shown in Figure 5.2.

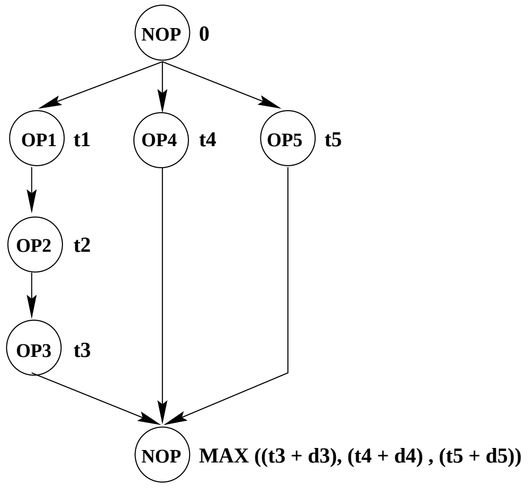


Figure 5.2: **Scheduled Sequential Basic Block**

Conditional Basic Block - A conditional basic block consists of an *initial state*, a number of branches and a *final* state. No operation - and therefore no execution delay - is assigned to the initial and final states of the block. The initial state is marked with a ‘zero’ execution delay before the scheduling starts. Each branch of a conditional basic block is a sequential basic block. Each branch is scheduled independently using a suitable scheduling algorithm, as explained above. Then, the maximum execution delay among the branches is assigned to final state. Figure 5.3 shows a scheduled conditional basic block.

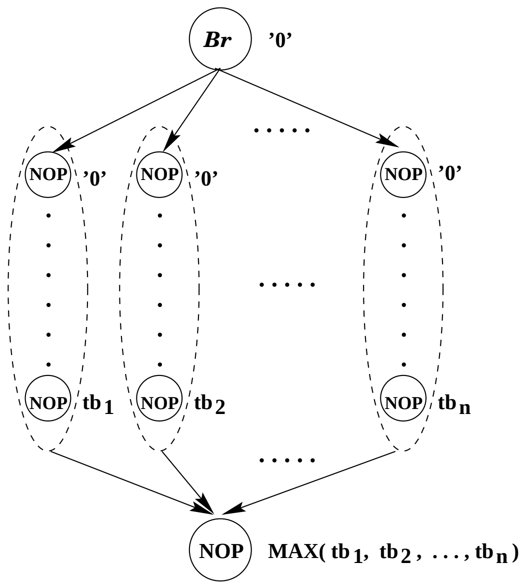


Figure 5.3: **Scheduled Conditional Basic Block**

Iterative Basic Block - The body of an iterative basic block is a sequential basic block. During the scheduling process, a schedule of the body of the iterative block is generated first. The execution delay of each iteration of an iterative basic block is equal to the execution delay of its body. Iterative blocks are divided to two types. For the fist type, the number of the iterations of the loop is known at the time of the scheduling. This type of the iterative block may be first unrolled into a sequential basic block, and then scheduled, to optimize its execution delay. If the scheduling algorithm does not unroll such an iterative block, its execution delay is defined as the execution delay of its and times the number of iterations. The second type of the iterative block is one in which the number of the iterations is not known in advance, and is determined during the execution of the loop. Also in this case, the execution delay of the block is a multiple of the execution delay of its body, or possibly '0' if the loop condition is never 'true'. Figure 5.4 shows a scheduled iterative basic block.

Procedure Basic Block - This basic block is a sequence of the operations that can be invoked multiple times in the CDFG. The body of a procedure, is a sequential basic block. The scheduling of a procedure basic block is the same as the scheduling of a sequential basic block. Figure 5.5 show a scheduled procedure basic block.

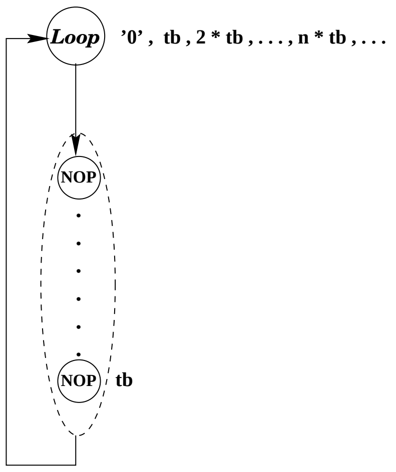


Figure 5.4: **Scheduled Iterative Basic Block**

5.2.1 Scheduling of Non-Hierarchical CDFGs

A non-hierarchical CDFG is a sequence of the basic blocks. The scheduling of a non-hierarchical CDFG consists of scheduling each basic block independently, and then calculating the execution delays of the vertices of the CDFG.

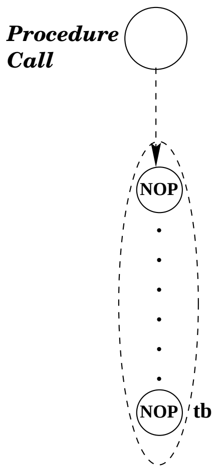


Figure 5.5: **Scheduled Procedure Basic Block**

Choosing a scheduling algorithm for non-hierarchical designs is a global decision and depends on the design constraints and their type. There are well known scheduling algorithms which handle different types of constraints. Once a scheduling algorithm is selected, it is used for scheduling all basic blocks if the CDFG.

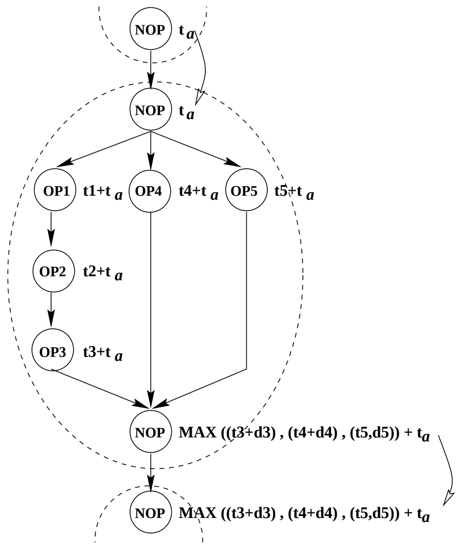
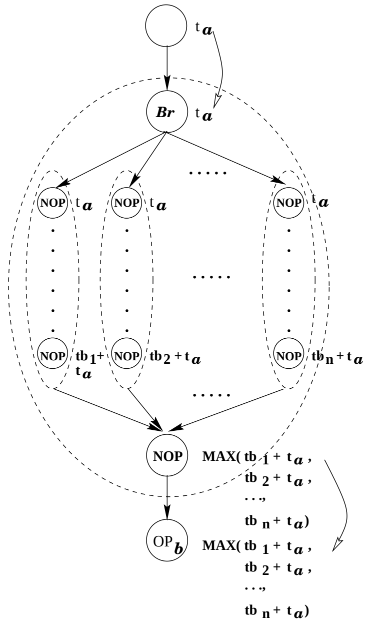

NOP
t a
NOP
t a
OP1
t1+t a
OP4
t4+t a
OP5
t5+t a
OP2
t2+t a
OP3
t3+t a
NOP
MAX ((t3+d3) , (t4+d4) , (t5,d5)) + t a
NOP
MAX ((t3+d3) , (t4+d4) , (t5,d5)) + t a

Figure 5.6: **Control-step assignment to a sequential basic block during the global scheduling of a non-hierarchical CDFG**

The time-step assignment of a sequential basic block is as follows: The time step of the vertex preceding the block is assigned to the first state of the block and the time-steps of the rest of the vertices are adjusted accordingly (by adding the time-step of the initial step to the time-step of each vertex). The time-step of the final vertex of the block is assigned to the vertex following it, which is the initial state of another basic block. Figure 5.6 shows the time-step adjustment of the basic block of Figure 5.2 in global scheduling of the CDFG.

During the scheduling of a non-hierarchical CDFG, the time-steps of conditional basic blocks are assigned as follows: First the conditional block is scheduled independent from the rest of the CDFG using a suitable scheduling algorithm, then its time-steps are adjusted to fit with the schedule of the rest of CDFG. To make this adjustment, the time-step of the branching vertex is assigned to the first vertex of the conditional block, and the time-steps of the rest of the vertices are adjusted accordingly. The time-step of the final vertex of the conditional block is assigned to its succeeding vertex during scheduling the next basic block in the sequence. Figure 5.3 shows the time-step adjustment of the basic block of Figure 5.7 after global scheduling of the CDFG.

During the scheduling a non-hierarchical CDFG, the time-steps of iterative basic blocks are assigned as follows: First the loop is scheduled independent from the rest of the CDFG using a suitable scheduling algorithm, then its time-steps are adjusted to fit with the rest of the CDFG. To make



t_a
Br
t_a
NOP
t_a
tb_1 + t_a
tb_2 + t_a
tb_n + t_a
MAX(tb_1 + t_a, tb_2 + t_a, ..., tb_n + t_a)
OP_b
MAX(tb_1 + t_a, tb_2 + t_a, ..., tb_n + t_a)

Figure 5.7: **Control-step assignment to a conditional basic block during the global scheduling of a non-hierarchical CDFG**

this adjustment, the time-step of the first vertex of the iterative block is assigned to the first vertex of the iterative block, and the time-steps of the rest of the vertices are adjusted accordingly. The time-step (sequence of time-steps) of the final vertex of the loop is assigned to its succeeding vertex during scheduling the next basic block in the sequence. Figure 5.4 shows the time-step adjustment of the basic block of Figure 5.8 after global scheduling of the CDFG.

During the scheduling of a non-hierarchical CDFG, the time-steps corresponding to a call to a procedure basic block are assigned as follows: first the procedure basic block is scheduled independent from the rest of the CDFG using a suitable scheduling algorithm, and then its time-steps are adjusted to fit with the rest of the CDFG. To make this adjustment, the time-step of the time-step of vertex corresponding to the procedure call is assigned to the first vertex of the procedure basic block, and the time-steps of the rest of the vertices are adjusted accordingly. The time step of the final vertex of the procedure basic block is assigned to its succeeding vertex during scheduling the next basic block in the sequence. Figure 5.5 shows the time-step adjustment of the basic block of Figure 5.9 after global scheduling of the CDFG.

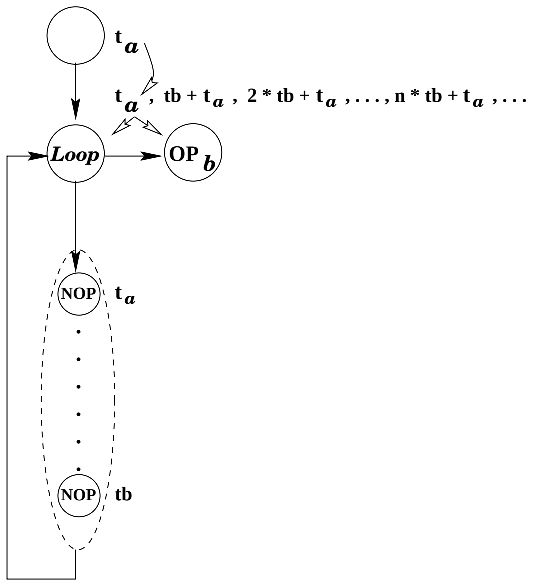

t_a
t_a , tb + t_a , 2 * tb + t_a , . . . , n * tb + t_a , . . .
Loop
OP_b
NOP
t_a
NOP
tb

Figure 5.8: **Control-step assignment to an iterative basic block during the global scheduling of a non-hierarchical CDFG**

5.2.2 Scheduling of Hierarchical CDFGs

The unconstrained scheduling of the hierarchical CDFGs is straight forward, but this is not the case with the constrained scheduling. Most scheduling algorithms do not allow resource sharing across different blocks in the hierarchy to avoid difficulties of hierarchical scheduling. In this case, each basic block is scheduled independently. Since the CDFG has a hierarchical structure, the scheduling task is performed hierarchically in a bottom up fashion. The innermost blocks of the hierarchy (the basic blocks) will be scheduled first and the time-steps of the vertices at each level will be calculated in terms of the time-steps of the vertices of its immediate lower level in a manner similar to what was explained for the non-hierarchical CDFGs. The structure of the generic blocks of the hierarchical CDFGs is similar to the structure of the basic blocks, except that they are hierarchical themselves. The same algorithms for assigning the time-steps from the top-level vertices to the initial states of the basic blocks and from final states of the basic blocks to the preceding vertices of the top-level can be applied in the hierarchical scheduling.

Different algorithms are used for scheduling of hierarchical and non-hierarchical CDFGs based on the type of temporal or spatial constraints posed on the designs. ASAP (As Soon As Possible) algorithm is usually used for unconstrained scheduling, ALAP (As Late As Possible) is used for latency-constrained scheduling, Relative Scheduling is used when the relative timing constraints are posed on the design, Hu's algorithm is used for resource constrained scheduling, list scheduling

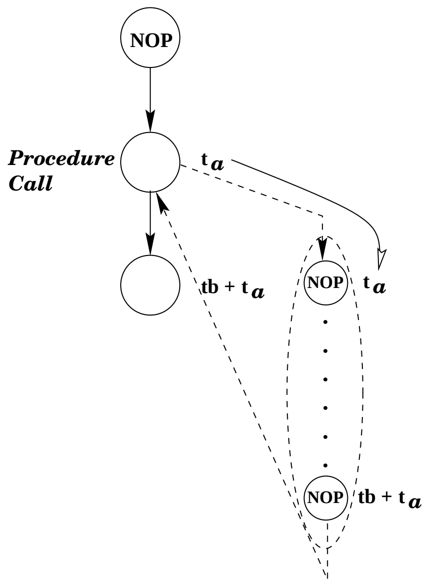


Figure 5.9: **Control-step assignment to a procedure basic block as part of scheduling of a non-hierarchical CDFG**

and heuristic force directed scheduling algorithms are used to solve resource-constrained as well as latency-constrained problems. These algorithms do not move the operations across the border points of the basic blocks.

5.2.3 Discussion

The scheduling transformations are performed as part of the synthesis process. Due to transformations of the scheduling phase, each coarse grain behavior operation is transformed into a set of finer grain register transfer operations (Figure 5.10.(a)). In addition, the order of the operations in the implementation (Figure 5.10.(b)), their relative level of parallelism (Figure 5.10.(c)) or both (Figure 5.10.(d)) are modified from those in specification. However, it should be noted that in most HLS systems, even though the scheduling process modifies the order and the relative degree of parallelism of operations, it preserves the basic specification constructs and their order. By this statement we mean that when a specification is being transformed, the order of statements and their relative degree of parallelism can be modified, but only within the border points of each basic construct (iterative or conditional). As the discussions of the following chapters will reveal, this restriction on allowable degree of code motion by high-level synthesis systems, is one of the key factors in developing a fully automated method for verifying the designs generated by these systems.

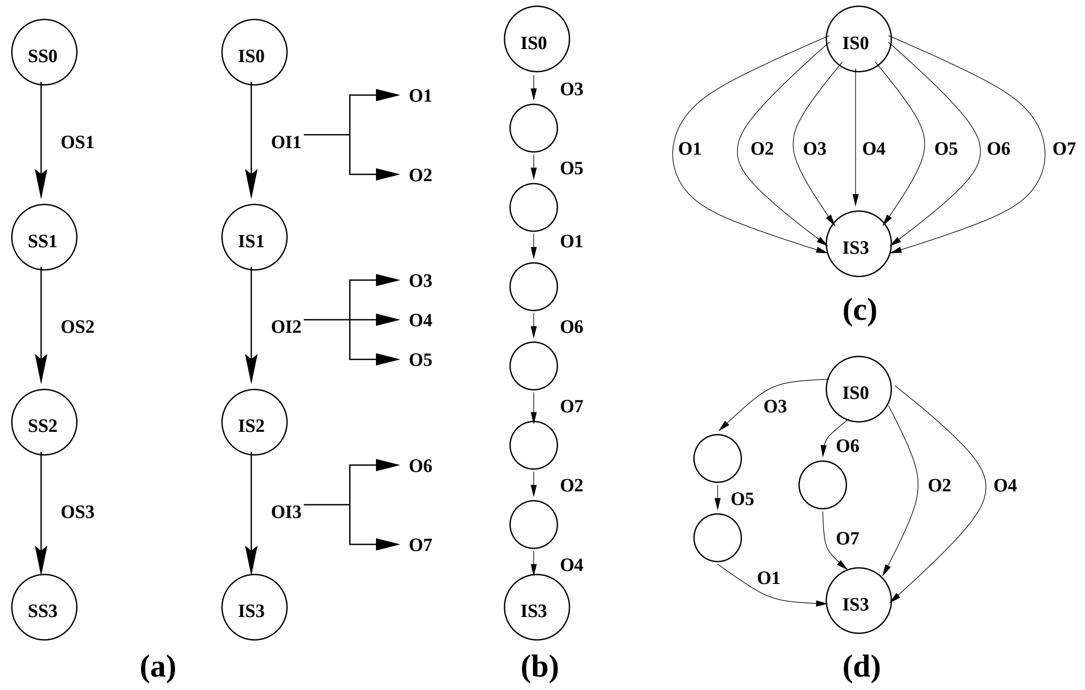


Figure 5.10: **The Effects of Scheduling**

5.3 Impact of Operator Allocation on Verification Methodology

We mentioned before that most high-level synthesis algorithms are oblivious to the mathematical properties of arithmetic and logic operators. This means that selection and sharing of the operators is done solely based on matching the uninterpreted function symbols and constants, e.g. an 'adder' functional unit of the component library is selected to perform the behavior operation '+', a 'multiplier' to perform '*', etc. This feature of high-level synthesis may be effectively used for verification purposes. Also, the domain of values for both specification variables, and RTL signals may be left uninterpreted. The specification variables may be the operands of some operations and/or may hold the results of some operations. The RTL signals are the input/output values of different data-path components. The input/output signals of data-path functional units correspond to the operands or results of behavior operations and therefore are assumed to have the same domain of values.

By assuming an uninterpreted domain of values for specification variables and RTL signals, and uninterpreted operation of behavior operators and RTL functional units, the comparison of the values may be done through symbolic analysis and making use of rewriting strategies. Under the above assumption, the bit widths of the variables or operators do not directly affect the verification time. Consequently, the verification exercise may be conducted more efficiently than otherwise possible and also the correctness of designs of reasonably large size may be verified. This observation has been one of the key elements in developing the verification methodology presented in this dissertation.

5.4 Impact of Register Allocation on Verification Methodology

The spatial properties of an RTL design are in as close connection with the register allocation problem as its temporal properties with the scheduling problem. The choice of register allocation and optimization schemes during the synthesis of a design directly influences its spatial properties.

In a simplistic synthesis process, with no register optimization task, each variable is bound to a physical component - a register - representing it, and there is a one to one relation between the specification variables and implementation design's architectural registers (Figure 5.11). In such a synthesis scheme a static binding function may be used to define the mapping between the specification variables and implementation design's architectural registers. This is not true when register optimization is performed as part of the synthesis process, and in synthesis systems that employ sophisticated register allocation algorithms.

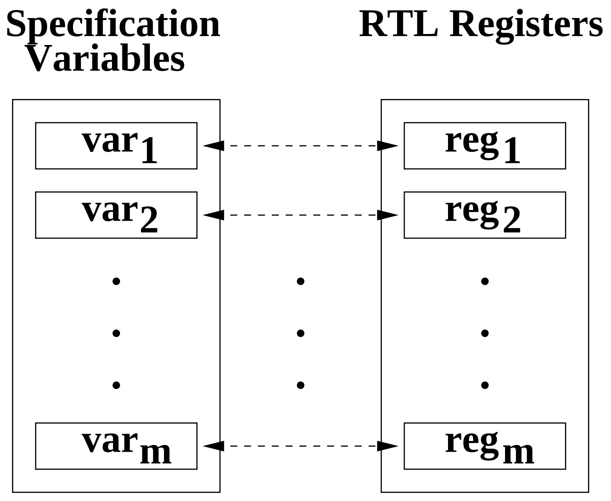


Figure 5.11: **Mapping relation between specification variables and design registers when no register optimization is performed**

The goal of register optimization is to share registers whose lifetimes do not overlap across the scheduled time-scale. This is done by life cycle analysis of the design registers. Each specification variable has a *lifetime* that is in the form of a union of intervals $[l_i, u_i]$. Each interval is from the variable's *birth* l_i, to its *death* u_i, where the former is the latest time at which its value is generated as an output of an operation and the latter is the latest time at which the variable is referenced as an input to another operation. In some register allocation schemes, if two variables have non-overlapping lifetimes, they are bound to the same register. In some other schemes, if two variables hold equivalent values, they are bound to the same register. In either case the mapping relation between the specification variables and design registers is no longer a bijection.

Two types of register allocation-optimization schemes are commonly found in high-level synthesis tools: *value based* and *carrier based.* When transformations based on these schemes are performed during the synthesis process, the mapping relation between the specification variables and RTL registers is no longer bijective. Since the definition of equivalence directly depends on the mapping between the variables and registers, a precise definition of this relation in each case is necessary. In what follows these common register allocation schemes are presented, and in each case the relation between the specification variables and design registers is precisely defined.

5.4.1 Carrier Based Register Allocation

The carrier based register allocation scheme yields a mapping from the variables to the registers. FACET [62], HAL [50], and CHARM [69] use carrier-based techniques for register optimization.Register optimization is only possible when two (or more) variables, have non-overlapping lifetimes, in which case they are bound to the same register. Life-cycle analysis of variables is performed to establish compatibility relation between pairs of variables. Then a *compatibility graph* $G_+(V, E)$, or a *conflict graph* $G_-(V, E)$ will be formed.

A compatibility graph $G_+(V, E)$, has a vertex set $V = \{v_i : 1 \leq i \leq n_{reg}\}$ which is in one to one correspondence with the registers, and an edge set $E = \{\langle v_i, v_j\rangle : 1 \leq i \leq n_{reg} \quad and \quad 1 \leq j \leq n_{reg}\}$ which denotes compatible registers (register pairs with non-overlapping lifetimes). The optimum register sharing algorithm minimizes the number of registers by folding compatible registers. In the compatibility graph, a group of mutually compatible registers correspond to a subset of mutually connected edges, i.e. to a clique. Therefore a maximal set of mutually compatible registers is represented by a maximal clique in the compatibility graph. Then the optimum register sharing problem is equivalent to partitioning the graph into a minimum number of cliques. This problem is solved by a clique partitioning algorithm.

A conflict graph $G_-(V, E)$, has a vertex set $V = \{v_i : 1 \leq i \leq n_{reg}\}$ which is in one to one correspondence with the registers, and an edge set $E = \{\langle v_i, v_j\rangle : 1 \leq i \leq n_{reg} \quad and \quad 1 \leq j \leq n_{reg}\}$ which denotes conflicting registers (register pairs with overlapping lifetimes). A set of mutually compatible registers corresponds to a subset of vertices that are not connected by edges, referred

```
variables a, b, neq, grt;

a := read_input();
b := read_input();
neq := (a ≠ b);

while neq do
    grt := (a > b);
    if grt then
        a := a - b;
    else
        b := b - a;
    endif;
    neq := (a ≠ b);
enddo;

write_output( a );
```

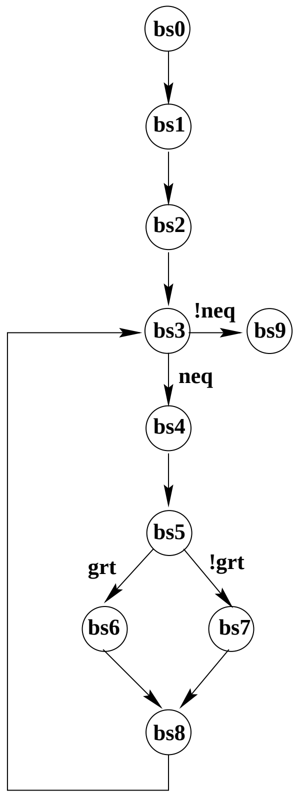


Figure 5.12: **Specification of the Greatest Common Divisor Design**

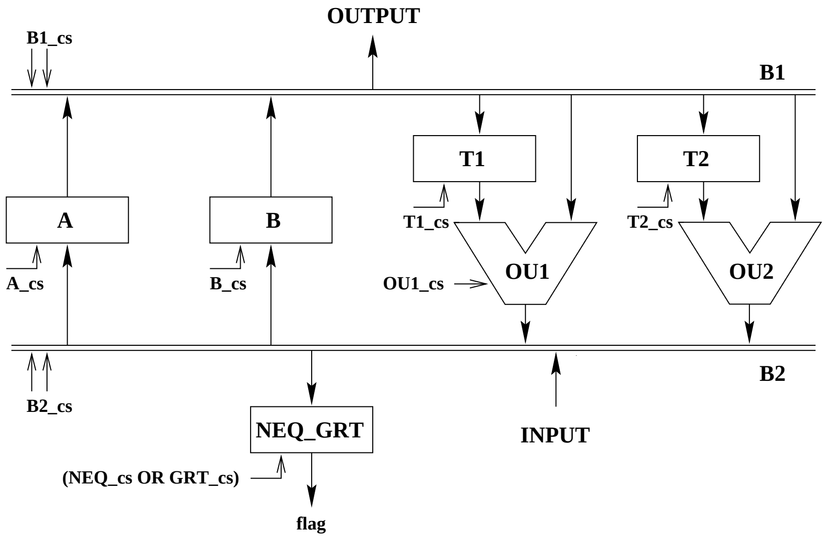


Figure 5.13: **Example of the data-path of an RTL design generated by a high-level synthesis system after scheduling and resource optimization**

to the independent set of $G_{-}(V, E)$. An optimum register sharing corresponds to a vertex coloring with a minimum number of colors. This problem can be solved by a graph coloring algorithm.

Regardless of the algorithm used for register optimization, two or more variables are bound to the same register as the result of register allocation (Figure 5.14). But, this is done by merging registers who do not have overlapping lifetimes, i.e. only one of these registers is "live" across any state and the other(s) is(are) dead. It is obvious that when register allocation is performed using this scheme, there is not a one to one correspondence between the specification variables and any subset of RTL registers. Now, the mapping between the variables and registers should be defined. To define such the register mapping relation, it should be noted that a specification variable may be alive at some states and dead at others. Therefore, the mapping relation between the variables and registers changes from one state to the other. Then, the mapping relation at each state is defined only between those variables that are alive at that state and the design registers. Then the mapping relation is dynamically defined at each state. The dynamic mapping relation will be discussed in following chapters.

Figure 5.12 shows the behavior specification of GCD, a circuit which finds the greatest common divisor of two numbers. The data-path of the RTL design generated from this specification, by a synthesis system that performs carrier based register optimization is shown in Figure 5.13. In this example it can be noted that the two variables *neq* and *grt* have non-overlapping lifetimes and may be subjected to register folding.

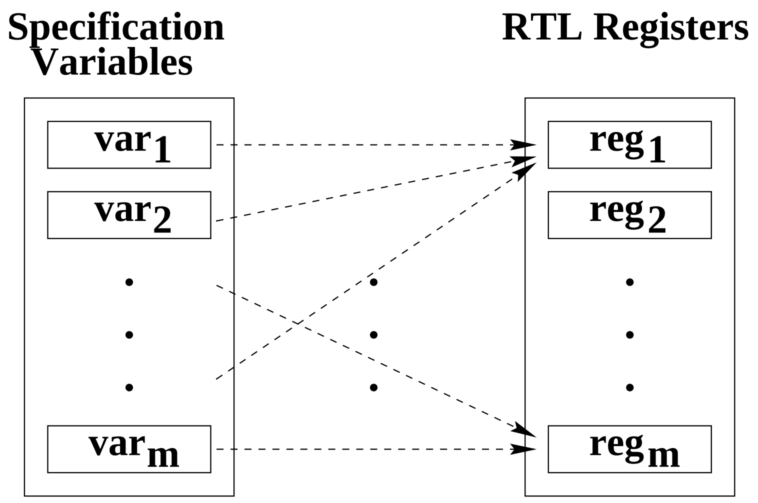
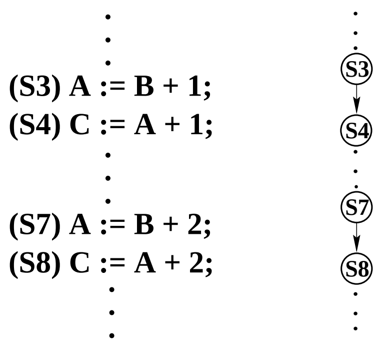


Figure 5.14: **Mapping relation between specification variables and design registers in a carrier-based register allocation scheme**

5.4.2 Value Based Register Allocation

The value based register allocation scheme yields a mapping from the values to registers. Different values can be assigned to the same specification variable in a behavioral description, therefore, a specification variable can be mapped to several design registers. Many high-level synthesis systems such as REAL [36], EMUCS [60] and EASY [58] are the systems which use value-based register optimization techniques.



```
        .
        .
        .
(S3) A := B + 1;
(S4) C := A + 1;
        .
        .
        .
(S7) A := B + 2;
(S8) C := A + 2;
        .
        .
        .
```

Figure 5.15: **Example of a partial specification**

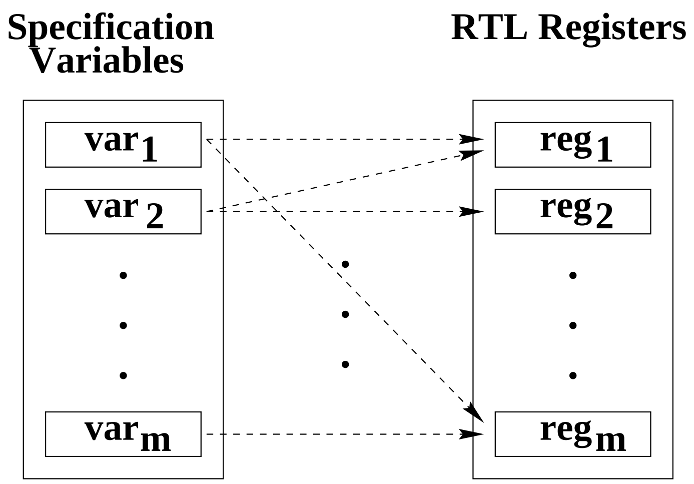


Figure 5.16: **Mapping relation between specification variables and design registers in a value-based register allocation scheme**

In value-based approach, register optimization is modeled as the problem of mapping data-values produced and used by operations in a data flow graph representation of the specification into registers. Register optimization is only possible when two (or more) operations use the same data values. Consider the partial specification of Figure 5.15. A value based register allocation scheme may assign register R_1, R_2, R_3 and R_4 to the values $B+1$, $A+1$, $B+2$ and $A+2$ respectively. Then, along the path $[\cdots, S_3, S_4, \cdots]$, the specification variable A is mapped to the register R_1 and along the path $[\cdots, S_7, S_8, \cdots]$ it is mapped to R_3.

Left Edge Algorithm for interval graphs can be used for performing register optimization. The register optimization problem is modeled as a channel routing problem; the life span of each value is modeled as a net interval. The number of tracks determined by the left-edge algorithm then corresponds to the number of registers needed.

It is clear that in this scheme also, there is not a one to one correspondence between the specification variables and any subset of the data-path registers. The same variable may correspond to different registers, and the same register may correspond to different variables (Figure 5.16). In this case also, a static mapping relation from the variables to registers is not appropriate. Like the case of carrier based register allocation, the mapping relation between the variables and registers at each state is defined dynamically.

Since register optimization in this scheme consists of merging the design registers that hold the same value, it is possible that more than one specification variable is mapped to the same design register at a state. This does not introduce a new problem in defining the mapping relation.

5.5 Impact of Interconnect Allocation on Verification Methodology

Two possible interconnect allocation schemes are found in high-level synthesis tools: point-to-point interconnect allocation and bus allocation. Data-path interconnections represent data-flow in behavior specification. Interconnect allocation (optimization) is a procedure similar to register allocation (optimization). To allow sharing of the interconnections during the interconnect allocation phase multiplexers may be introduced to the circuit. During the verification process, the contents of different registers in terms of uninterpreted values are symbolically evaluated. During this analysis, data-path components are traversed and all components, including the multiplexers and buses are accounted for. In case of an error in synthesis of the steering logic, that results in incorrect function of the design, the signal values are not correctly propagated and this error is detected during the observation equivalence checking.

In our verification method, all values, constants and operators are left uninterpreted, and all signals are assumed to have the same type. Also, the input and output values of all the combinational and sequential components of the data-path are assumed to belong to the same uninterpreted domain of values, too. The assumption of uninterpreted functions and values in our approach has the underlying advantage of reducing the complexities associated with bit-vector signal types. However, in the synthesis of RTL designs, there are situations when bit level effects should be dealt with.

In high-level synthesis, the ratio of the signals that are split to smaller bit-vectors or are merged into larger bit-vectors, compared to the total number of signals in the circuit, is often small. Dealing with all signals at the bit-level to account for these special cases drastically increases the complexity of the verification process, and is no an acceptable solution. To solve this problem, we introduce two virtual components 'SPLIT' and 'MERGE' to the set of synthesis resources. These components allow to model all possible types of interconnections in the data-path of the RTL design. These components allow for dealing with the problems at the bit level whenever necessary, while maintaining the signals uninterpreted at all other times.

5.5.1 Bit-Level Interconnect Allocation

When the global interconnection pattern between execution units is decided upon, this pattern is detailed to the bit-level. Bit-level effects, such as *type casts* and *signal alignment*, are considered at this stage [40].

- Most descriptions contain relations between signals of different types. Some signals are converted implicitly from one type to another by data path operators, e.g., a multiplier adds the word lengths of the operands to deduce the word length of the result. This is called *coercion.* For other signals, the designer explicitly changes the type, using a *cast* operation. Both kinds of conversions are implemented on the chip.

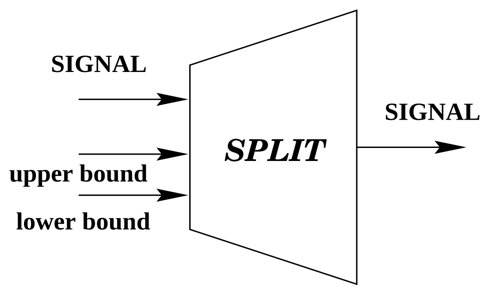
- If data-path operators are multiplexed, an operator with a certain word length may operate on signals of smaller word length. In this case, the operands are *aligned* either at the MSB side of the registers or at the LSB side. As a consequence, the result of the operations is also aligned MSB or LSB at the output of the execution unit. Some alignment requirements originate from the code expansion macros. For instance, for a Booth-multiplication on an ALU, both operands must be aligned at the LSB side. Other alignments can be selected freely.

Even if the word length of signals corresponds with the functional unit operating on them, the binary point position of the signals may vary. The bit-level interconnect strategies take these alignment aspects into account.

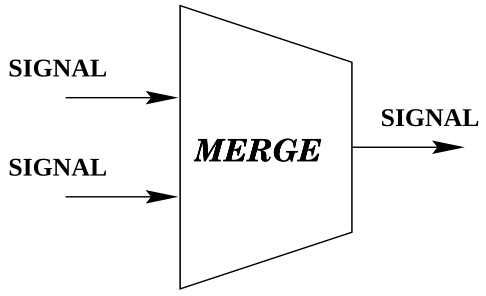
The two virtual components 'SPLIT' and 'MERGE' are introduced to the set of synthesis resources, to model all possible types of interconnections in the data-path of the RTL design. There are two types of interconnections that, due to the assumption of an uninterpreted domain of values for the signals, cannot be accounted for: (1) assigning a subrange of the bits of an output signal to the input of another component with smaller bit-size. (2) merging the bits of two (or more) output signals to produce an input signal for a component with larger bit-size. By using the 'SPLIT' and 'MERGE' virtual components, we do not introduce new components (extra hardware) to the circuit. On the contrary, we offer a solution for modeling a phenomenon that already exist in the circuit, but is not accounted for. Therefore, 'SPLIT' and 'MERGE' are actually a formal modeling of these two types of interconnections and aid us to define the bit-level properties of the interconnections axiomatically.

SPLIT Operator

SPLIT is considered a synthesis functional resource. This component takes as input (1) a signal value VAL which according to behavior description has $n+1$ bits : $VAL[n,0]$ (and is either an input signal or the output of another component), and (2) a subrange of its bits $[u,l]$ $(0 \leq l \leq u \leq n)$ and returns a signal value by converting the bit vector $VAL[u,l]$ into a signal. The operation of



(a) Split Component



(b) Merge Component

Figure 5.17: **Virtual Components : Accounting for Bit-Level Interconnections**

SPLIT is presented schematically in Figure 5.18.

MERGE Operator

MERGE is considered a functional resource. This component takes two signal values VAL_1 and VAL_2 as input which according to behavior description have $n+1$ and $m+1$ bits respectively and merge them into a signal VAL which if represented as bit-vector has $n+m+2$ bits. The signal VAL may not have a specific relation with the signals VAL_1 and VAL_2, but the bit-vector representation of it $val[n+m+1,0]$ can be defined in terms of the bit-vector representation of its input signals $val_1[n,0]$ and $val_2[m,0]$ as follows:

$$val[n+m+1,m+1] = val_1[n,0] \quad \text{and}$$
$$val[m,0] = val_2[m,0]$$

A detailed description of the MERGE component is presented graphically in Figure 5.19.

The behavior of the MERGE and SPLIT operations may be defined axiomatically. These axioms are used to define particular bit-level properties of the design during the verification process. An example is an axiom about merge operation, stating that given the signals VAL_1 and VAL_2 are merged into a signal VAL, if VAL_1 is equal to zero then the value of signal VAL is equal to the value of signal VAL_2. These axioms are useful when dealing with the bit-size effects discussed before.

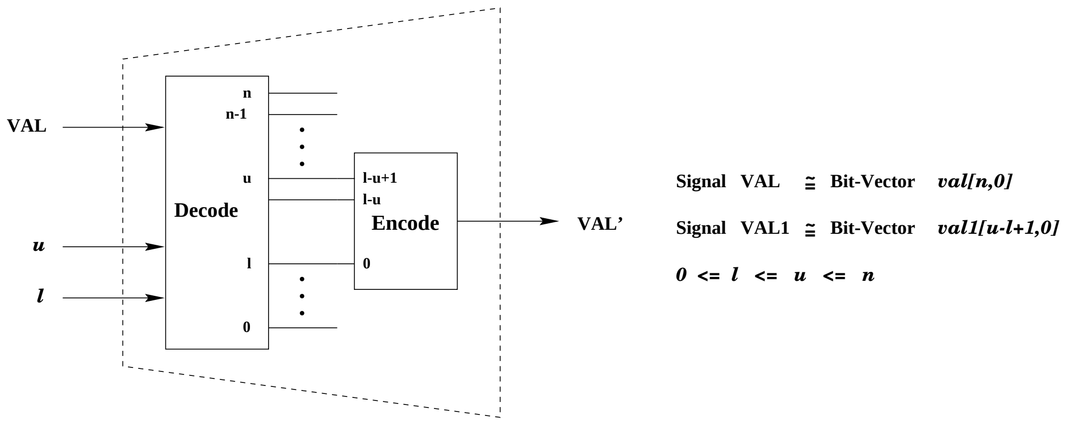

VAL
u
l
Decode
n
n-1
u
l
0
l-u+1
l-u
Encode
0
VAL’
Signal VAL ≅ Bit-Vector val[n,0]
Signal VAL1 ≅ Bit-Vector val1[u-l+1,0]
0 <= l <= u <= n

Figure 5.18: **Detailed SPLIT Component**

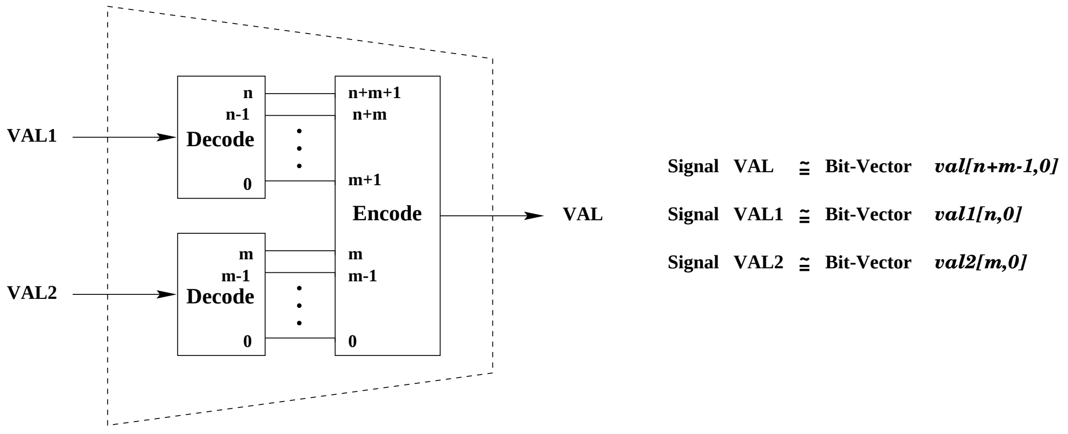

VAL1
VAL2
Decode
n
n-1
0
Decode
m
m-1
0
n+m+1
n+m
m+1
Encode
m
m-1
0
VAL
Signal VAL ≅ Bit-Vector val[n+m-1,0]
Signal VAL1 ≅ Bit-Vector val1[n,0]
Signal VAL2 ≅ Bit-Vector val2[m,0]

Figure 5.19: **Detailed SPLIT Component**

5.6 Impact of Controller Generation on Verification Methodology

At this stage of the synthesis process, the number of controller states, its inputs at each state (the flags from the data-path) and its outputs at each state (the control signals) are known. With this information the controller of the RTL design is synthesized. The incorrect design of the controller results in generation of incorrect control signals. The incorrect control signals lead to incorrect RTL data-transfers that result in incorrect values in registers. The incorrect register values are detected during the equivalence checking process. Therefore, the implementation of the controller need not be separately verified.

5.7 Conclusion

In this chapter we presented detailed discussions about various stages of high-level synthesis, and explained how the algorithms and transformations employed at each stage, affect the elements of verification. We believe that with the knowledge of synthesis process, synthesis aware verification algorithms may be developed, and educated reasoning leading to effective verification can replace the restricted, blind verification techniques.

In the course of this chapter we have identified a set of properties that are specific to synthesized designs. These properties may be summarized as follows:

(1) Each basic block of the behavior specification is scheduled independently. That is to say the RTL design may be partitioned into independently scheduled RTL modules, such that each module represents a basic block of the behavior specification. Therefore, there is a bijective mapping between the independently scheduled RTL modules and the basic blocks of the behavior specification.

(2) The input/output signals of the RTL components are considered to to have uninterpreted values, unless the bit-level properties of the design are of interest.

(3) When register optimization performed as part of synthesis, the mapping between the specification variables and design register should be defined dynamically. As a result of a dynamic mapping the relation between the variables and registers may be "many to many".

(4) The errors in the controller that result in malfunction of the RTL design can be detected by verifying the equivalence of the RTL design and its behavior description.

In the following chapters we will show how these properties may be exploited to develop an effective verification method.

Chapter 6

Formalization of the Verification Technique

Consider S to be a behavior specification, and I to be its synthesized RTL implementation. We showed that S may be partitioned into a set of sub-systems s_k where $1 \leq k \leq n$. Each sub-system of S is a partial behavior description whose operation flow is described by a directed acyclic sub-graph of the operation flow graph of S. The system I is also partitioned into n sub-systems i_1 to i_n, where each sub-system i_k of I is an RTL unit that inherits the data-path of I, and its controller is a directed acyclic sub-graph of the controller of I.

Consider that for the designs S and I the following two conditions are true: (1) the two systems have equivalent initial internal states, and (2) there is a bijective mapping between the sub-systems of S and sub-systems of I, such that each sub-system of I is an implementation of a sub-system of S. Now, let us assume that each sub-system of S (each synthesis basic block of S) is observation equivalent to a sub-system of I (an RTL unit of I). We claim that under the under these assumptions we can prove that I is equivalent to S, and the equivalence relation between S and I is no weaker than observation equivalence. In the following sections we offer a formal proof for this claim. The following next sections present some formal definitions and notations.

6.1 Critical Verification Elements

In Chapter 3 the equivalence of each pair of sub-systems s_k and i_k was defined based on their observation equivalence. The observable behavior of s_k (or i_k) was in turn defined in terms of two types of communications, its communication with other sub-systems of S (or I), and its communication with the environment of S (or I).

We explained that the behavior of each sub-system s_k is described by g_{b_k}, a directed acyclic sub-graph of the operation flow graph of S, except that g_{b_k} is modified so that at its initial state s_k reads the values of the specification variables from the inputs, and at its final state s_k writes the values of the specification variables. Even though these input/output operations are not observable at the system level, at the sub-system level they define the behavior of s_k.

The behavior of s_k is formulated in terms of the states it communicates with its environment and the values it communicates with its environment. Therefore, such states are referred to as *critical behavior states* and, the variables holding such values are referred to as *critical specification variables.*

Through an analogy, similar states of the controller of the RTL design and similar value holding elements of each RTL unit are defined as *critical design states* and *critical design registers.*

Also, since the correctness of the design is defined based on the particular proposed decomposition of the finite state machine models of the behavior specification and RTL implementation, each directed acyclic sub-graph of the behavior automaton is referred to as a *behavior critical path*, and each directed acyclic sub-graph of the controller automaton is referred to as a *design critical path.* A critical path is path from one critical state to another without visiting any critical states in between.

This suggests that the following states in behavior specification should be marked as *critical states*: (1) conditional states – states with more than one outgoing transition; (2) join states – states with more than one incoming transition; (3) input states – states that read from input ports; (4) output states – states that write to output ports; (5) the procedure call states – states that call a procedure; (6) the start state, and, (7) the final state. These states introduce control flow dependencies into the CDFG and the HLS system never moves code across these states. Further, since, specification code between a pair of these states is subject to scheduling and hence to possible code motion during high-level synthesis, no other states can be marked as being critical. This imposes a lower bound on the length of the critical paths: a critical path may not be shorter than a basic block, except if it is joining a conditional state to another critical state, in which case it may not be shorter than either of the conditional branches.

The following variables in the behavior specification should be marked as *critical variables* in the specification: (1) input variables, (2) output variables, (3) any variable that has a live value across a critical state [1]. The last category of critical variables are easily identified during life-time analysis that precedes register allocation. All critical variables are preserved by the HLS tool and manifest in the RTL design in the form of critical registers and all critical states in the behavior automaton are preserved by the HLS tool and manifest in the form of critical states in the controller automaton.

[1]Note that these include any variable used as the deciding variable in transitioning from a conditional state.

Therefore, conditional, join, input, output, procedure call, start and final states in the controller together form the critical states of the controller. We assume that all critical states are reachable from the start states respectively in the behavior automaton and the RTL controller, that is, any dead-code has been eliminated prior to high-level synthesis. We rely on the HLS tool to supply the critical binding functions, i.e. the functions mapping the critical specification variables to critical design registers, the critical behavior states to critical design states and behavior critical paths to design critical paths. Now, a formalization of the problem based on the above definitions may be presented.

6.2 Formalization

Let R be the set of all registers or all variables of the design. Let $CR \subseteq R$ be the set of *critical registers*. Let S be the set of states in the automaton and X be the set of transitions among these states. Let $S0$ be the unique start state of the automaton. Let $CS \subseteq S$ be the set of critical states. We assume that every critical state is reachable from the start state. Let CP be the set of critical paths among these critical states. For any critical path $p \in CP$, $F(p)$ and $L(p)$ denote the originating and terminating states of p.

Then in a behavioral specification, modeled as a behavior automaton, the following elements are defined:

R_b : the set of behavior registers or specification variables
$CR_b \subseteq R_b$: the set of *critical variables*
S_b : the set of states in the behavioral automaton
X_b : the set of transitions among behavior states
$S0_b$: the unique start state of the behavioral automaton
$CS_b \subseteq S_b$: the set of critical states
CP_b : the set of critical paths among critical states
$F_b(p)$: the originating state of the behavior critical path p
$L_b(p)$: the terminating state of the behavior critical path p

Similarly, in an RTL design the following elements are defined:

R_d : the set of registers in the register level data path
$CR_d \subseteq R_d$: the set of critical registers in the data path
S_d : the set of states in the controller
X_d : the set of transitions among controller states

$S0_d$: the unique start state of the controller
$CS_d \subseteq S_d$: the set of critical states in the controller
CP_d : the set of critical paths among controller states
$F_d(p)$: the originating state of the design critical path p
$L_d(p)$: the terminating state of the design critical path p

Following the previous discussion, we postulate that the high-level synthesis tool can produce, as a byproduct of the synthesis process, the following two mappings (called bindings in the synthesis terminology):

$$\begin{array}{ll} B_r : CR_b \rightarrow CR_d & \textit{critical register binding} \\ B_s : CS_b \rightarrow CS_d & \textit{critical state binding} \end{array}$$

The start state of the behavior is always mapped to the start state of the controller, that is:

$$B_s(S0_b) = S0_d$$

From B_s and B_r we can easily derive another mapping $B_p : CP_b \rightarrow CP_d$ that is the *critical path binding*. A critical path $p_b \in CP_b$ is mapped to critical path $p_d \in CP_d$ if and only if their originating and terminating states are mapped by B_s and the transition conditions, if any, on the outgoing transitions of their originating states match. More formally, if v ($\neg v$) is the condition variable annotation on the originating transition of the behavior critical path p_b, and f ($\neg f$) is the condition flag of the design critical path p_d then:

$$B_p(p_b) = p_d \Longleftrightarrow \begin{array}{ll} B_s(F_b(p_b)) = F_d(p_d) & \wedge \\ B_s(L_b(p_b)) = L_d(p_d) & \wedge \\ f = B_r(v) & \end{array}$$

The proposition $f = B_r(v)$ formally states that if p_b is annotated with the condition variable v, then p_d is annotated with the condition flag $B_r(v)$. This ensures that if p_b is traversed in the behavior, p_d will be traversed in the RTL controller.

An *execution path* in the behavior is a finite sequence of critical states such that the first state in the sequence is the start state and any two successive states in the sequence form a critical path, that is:

$$\begin{aligned}\forall e_b \in EP_b\ ,\ e_b = [s_1, s_2, \cdots, s_i, s_{i+1}, \cdots] :& \\ &(s_1 = S0_b\ \wedge \\ &(\forall i > 1, \exists p \in CP_b : s_i = F_b(p)\ \ \wedge\ \ s_{i+1} = L_b(p)))\end{aligned}$$

Where EP_b denotes the set of all possible behavioral execution paths. Execution path in the RTL controller is similarly defined and EP_d denotes the set of all possible RTL execution paths. We can construct an execution path binding, $B_e : EP_b \rightarrow EP_d$ using the critical state binding as follows:

$$e = \{s_1, s_2, \cdots, s_i, s_{i+1}, \cdots\}\ \Rightarrow\ B_e(e) = \{B_s(s_1), B_s(s_2), \cdots, B_s(s_i), B_s(s_{i+1}), \cdots\}$$

where e is an execution path in the behavior automaton, and $B_e(e)$ is its corresponding execution path in the controller of RTL design. The last state in an execution path e is called the *termination state* of e and denoted by $T_b(e)$ for the behavior automaton and by $T_d(e)$ for the RTL controller.

For the purposes of defining equivalence between the behavior and its RTL implementation, we postulate an uninterpreted domain of *values*, V. These values can be 'stored' in behavioral variables as well as RTL registers. We postulate two functions for assigning values to critical variables and critical registers:

$$\begin{aligned}V_b &: EP_b \times CR_b \rightarrow V \\ V_d &: EP_d \times CR_d \rightarrow V\end{aligned}$$

V_b determines the value of a critical variable r_b when the behavioral automaton traversed the execution path e_b and reached the state $T_b(e_b)$. V_d is similarly defined. In the next section, we show axiomatic definitions of V_b and V_d suitable for symbolic manipulation, automatically generated from behavioral specifications and RTL descriptions respectively.

We are now ready to define various equivalence relationships between the behavior and the RTL design. We say that the initial state $S0_b$ in behavior automaton is equivalent to the initial state $S0_d$ in the controller provided that when the two machines start operation, the value of each critical specification variable is the same as the content of its corresponding critical register. The **initial state equivalence** is denoted by $S0_b \equiv S0_d$:

$$\boxed{S0_b \equiv S0_d \Leftrightarrow \forall r \in CR_b, V_b([S0_b], r) = V_d([S0_d], B_r(r))}$$

We say that a critical state s in the behavior and $B_s(s)$, its corresponding RTL critical state are *equivalent* provided if the behavior automaton and RTL design start operation from equivalent initial states, and go through identical transition sequences that end at s and $B_s(s)$ respectively, then the value of each critical specification variable and the content of its corresponding design register at these states are equivalent. We denote **state equivalence** by $s \equiv B_s(s)$:

$$\boxed{\begin{array}{l} \forall s \in CS_b : s \equiv B_s(s) \Leftrightarrow \\ \qquad (S0_b \equiv S0_d \Rightarrow \\ \qquad\qquad (\forall e \in EP_b, s = T_b(e), \forall r \in CR_b : \\ \qquad\qquad\qquad\qquad\qquad\qquad V_b(e,r) = V_d(B_e(e), B_r(r)))) \end{array}}$$

State equivalence is defined only between initial states and between the terminating states of identical pairs of behavior-design execution paths, therefore, it is obvious that $s_b \equiv s_d$ only if $s_d = B_s(s_b)$.

We say that a behavioral execution path e is *equivalent* to its corresponding RTL execution path $B_e(e)$ provided any state of the behavior execution path is equivalent to its corresponding state in the design execution path. **Execution path equivalence** is defined only between a behavior execution path and the design execution path to which it is bound during the synthesis process, and is denoted by $e \equiv B_e(e)$:

$$\boxed{\forall e \in EP_b : e \equiv B_e(e) \Leftrightarrow (S0_b \equiv S0_d \Rightarrow \forall s \in e, s \equiv B_s(s))}$$

We say that the RTL design is equivalent to the behavior specification provided each possible behavior execution path is equivalent to its corresponding design execution path:

$$\boxed{M_b \equiv M_d \Leftrightarrow (\forall e \in EP_b : e \equiv B_e(e))}$$

M_b and M_d correspond to *Behavior Automaton* and *Design Automaton*, respectively.

We say a behavioral critical path p is equivalent to its corresponding RTL critical path $B_p(p)$ provided if starting from equivalent initial states, the two critical paths terminate in equivalent states. The **critical path equivalence** is defined only between a behavior critical path and the design critical path to which it is bound during the synthesis process and is denoted by $p \equiv B_p(p)$:

$$\boxed{\forall p \in CP_b : p \equiv B_p(p) \Leftrightarrow (F_b(p) \equiv F_d(B_p(p)) \Rightarrow L_b(p) \equiv L_d(B_p(p)))}$$

We claim that critical path equivalence implies execution path equivalence per the following theorem:

Equivalence Theorem: If every critical path in the behavior is equivalent to the RTL critical path to which it is bound during the synthesis process, then the RTL design is equivalent to the behavior specification. Formally:

$$\boxed{(\forall p \in CP_b : p \equiv B_p(p)) \Rightarrow M_b \equiv M_d}$$

The proof of this statement is straight forward and follows from the original assumptions that the initial states of the two machines are equivalent and that in both behavioral automaton and the RTL controller, the critical states are reachable from the respective start states. In this proof it is shown by induction on the length of the execution paths that critical path equivalence implies execution path equivalence, that in turn implies the equivalence of the designs:

$$\boxed{\forall p \in CP_b : p \equiv B_p(p) \Rightarrow \forall e \in EP_b : e \equiv B_e(e))}$$

The above theorem is one of the key components of this work and is the basis of our verification technique. We verify a synthesized RTL design by proving its equivalence to its behavior specification. The equivalence of the designs is established by proving the equivalence of each critical path of the controller of the RTL design to its corresponding critical path in behavior automaton.

The proof of the theorem follows from the fact that, in both behavior automaton and RTL controller, the critical states are reachable from the respective initial states. A detailed proof of the theorem, based on induction on the length of behavior execution paths, concludes this chapter. A mechanized version of this proof is given in Appendix A.

Induction Basis

Suppose that EP_{b1} is the set of all the behavior execution paths of length 1. We should prove that:

$$\forall p \in CP_b \ : \ p \equiv B_p(p) \Rightarrow \forall e \in EP_{b1} \ : \ e \equiv B_e(e)$$

Proof

1	$S0_b \equiv S0_d$	*premise*
2	$\forall p \in CP_b \ : \ p \equiv B_p(p)$	*premise*
3	$e_1 \in EP_{b1}$	*assumption*
4	$e_1 = \{SO_b\}$	3, *definition of execution path of length 1*
5	$e_1 \equiv B_e(e_1)$	1, *definition of equivalent execution paths*
6	$e_1 \in EP_{b1} \Rightarrow e_1 \equiv B_e(e_1)$	3-5 $\Rightarrow$*introduction*
7	$\forall e \in EP_{b1} \ : \ e \equiv B_e(e)$	6 $\forall$-*introduction*
8	$\forall p \in CP_b \ : \ p \equiv B_p(p) \Rightarrow$ $\forall e \in EP_{b1} \ : \ e \equiv B_e(e)$	2, 7 $\Rightarrow$*introduction*

QED

Induction Hypothesis

Suppose that EP_{bk} is the set of all behavior execution paths with the length less than or equal to k. Then the induction hypothesis is as follows:

$$\forall p \in CP_b \ : \ p \equiv B_p(p) \Rightarrow \forall e_k \in EP_{bk} \ : \ e_k \equiv B_e(e_k)$$

Inductive Step

Suppose that $EP_{b_{k+1}}$ is the set of all behavior execution paths with the length less than or equal to $k+1$. We should prove that:

$$\forall p \in CP_b \ : \ p \equiv B_p(p) \Rightarrow \forall e_{k+1} \in EP_{b_{k+1}} \ : \ e_{k+1} \equiv B_e(e_{k+1})$$

Proof

1	$S0_b \equiv S0_d$	*premise*
2	$\forall p \in CP_b \ : \ p \equiv B_p(p) \Rightarrow \forall e \in EP_{bk} \ : \ e \equiv B_e(e)$	*premise*
3	$\forall p \in CP_b \ : \ p \equiv B_p(p)$	*assumption*
4	$\forall e \in EP_{bk} \ : \ e \equiv B_e(e)$	2, 3 $\Rightarrow$-*introduction*
5	$(e_{j+1} = \{SB_0, bs_2, \cdots, bs_j, bs_{j+1}\}) \in EP_{b_{k+1}}$	*assumption*
6	$\exists e \in EP_{b_k} \ : \ e = \{SB_0, bs_2, \cdots, bs_j\}$	5, *definition of execution path*

7	$e_j = \{SB_0, bs_2, \cdots, bs_j\}$	6 $\exists$-*elimination*
8	$e_j \equiv B_e(e_j)$	4 $\forall$-*elimination*
9	$S0_b \equiv S0_d \Rightarrow \forall s_b \in e_j : s_b \equiv B_s(s_b)$	8, *definition of equivalent execution paths*
10	$\forall s_b \in e_j : s_b \equiv B_s(s_b)$	1, 9 $\Rightarrow$-*elimination*
11	$\exists p \in CP_b : p = Tail_b(e_{j+1})$	5, definition of critical path
12	$p_j = Tail_b(e_{j+1})$	11 $\exists$-*elimination*
13	$p_j \equiv B_p(p_j)$	3 $\forall$-*elimination*
14	$F_b(p_j) \equiv F_d(B_p(p_j)) \Rightarrow L_b(p_j) \equiv L_d(B_p(p_j))$	13, *definition of equivalent critical paths*
15	$B_s(F_b(p_j)) \equiv F_d(B_p(p_j)) \quad \wedge$ $B_s(L_b(p_j)) \equiv L_d(B_p(p_j))$	13, *definition of* B_p
16	$B_s(bs_j) \equiv F_d(B_p(p_j)) \quad \wedge$ $B_s(bs_{j+1}) \equiv L_d(B_p(p_j))$	12, 15 *substitution*
17	$F_b(p_j) \equiv B_s(bs_j) \Rightarrow L_b(p_j) \equiv B_s(bs_{j+1})$	16, 14 *substitution*
18	$bs_j \equiv B_s(bs_j) \Rightarrow bs_{j+1} \equiv B_s(bs_{j+1})$	12, 17 *substitution*
19	$bs_j \equiv B_s(bs_j)$	10 $\forall$-*elimination*
20	$bs_{j+1} \equiv B_s(bs_{j+1})$	18, 19 $\Rightarrow$-*elimination*
21	$\forall sb \in e_{j+1} : s_b \equiv B_s(sb)$	10, 19
22	$S0_b \equiv S0_d \Rightarrow \forall sb \in e_{j+1} : sb \equiv B_s(sb)$	1, 21 $\Rightarrow$-*introduction*
23	$e_{j+1} \equiv B_e(e_{j+1})$	21, *definition of equivalent execution paths*
24	$e_{j+1} \in EP_{b_{k+1}} \Rightarrow e_{j+1} = B_e(e_{j+1})$	5-22 $\Rightarrow$-*introduction*
25	$\forall e \in EP_{b_{k+1}} \ : \ e \equiv B_e(e)$	24 $\forall$-*introduction*
26	$\forall p \in CP_b \ : \ p \equiv B_p(p) \Rightarrow$ $\forall e \in EP_{b_{k+1}} \ : \ e \equiv B_e(e)$	3-25 $\Rightarrow$-*introduction*

QED

6.3 Dynamic Register Binding and Criticality Masking Technique

In previous discussions in this chapter, we assumed a static register binding between the critical specification variables and critical design registers, i.e. during the operation of RTL design, exactly one RTL register represents each behavior register. In such case, the register binding function is a bijection.

However, as it was explained in section 5.4, when register optimization is performed as part of the synthesis process, the critical register binding is no longer bijective. A register may represent a variable at some state and a different variable at another state. Let's consider two specification variables r_{b_1} and r_{b_2}, and critical path p_{b_k} (corresponding to sub-system s_k). Before register optimization is performed, two registers r_{d_1} and r_{d_2} are assigned to r_{b_1} and r_{b_2}, respectively. Let's also assume that r_{d_1} and r_{d_2} have non-overlapping lifetimes. Then, during the register optimization process, r_{d_1} and r_{d_2} are merged into a single register r_d. Now, even if the RTL design is correctly implemented, when the observation equivalence of p_{b_k} and its counterpart p_{d_k} is examined, exactly one of the following statements is true:

$$p_d \equiv B_p(p_b) \;\Leftrightarrow\; (F_b(p_b) \equiv F_d(B_p(p_b)) \;\Rightarrow\; V_b(r_{b_1}) = V_d(r_d))$$
$$p_d \equiv B_p(p_b) \;\Leftrightarrow\; (F_b(p_b) \equiv F_d(B_p(p_b)) \;\Rightarrow\; V_b(r_{b_2}) = V_d(r_d))$$

That means that the proof of the critical path lemma corresponding to p_{b_k} will fail. This failure of the proof is not as a result of design errors. It is due to the fact that in the proposed formalization, the register optimization properties of the design have not been accounted for. This problem can be solved by defining a *dynamic register binding* function, and through a technique referred to as *criticality masking*. A dynamic register binding function defines the register binding at each state:

$$B_r \;:\; CR_b \times CS_b \;\rightarrow\; CS_d$$

Also, since the set of critical registers vary from one state to next, we define the critical registers as a function rather than a set. So, if we denote the set of all the critical registers by CR'_b and (CR'_d in the case of design registers), then the critical registers at each state are defined by the function:

$$CR_b \;:\; CS_b \;\rightarrow\; \{r_b \mid r_b \in CR'_b\}$$

In a design such as described above, at the final state of the critical path p_b, at most one of the behavior registers r_{b_1} or r_{b_2} is alive. The behavior of the design is defined by this register, the contents of the other register does not influence the correct function of the design at this state. Therefore to solve the register binding problem, at the final state of each critical path, the criticality of any register that does not have a live value at this state is masked.

The critical registers at the initial state of the critical path p_{d_k} is defined as the intersection of sets of critical registers at the final states of the paths ending at that state:

$$CR_b(F_b(p_{b_k})) \;=\; \bigcap_{p_b \in \{p \mid p \in CP_b \,\wedge\, L_b(p) = F_b(p_{b_k})\}} CR_b(L_b(p_b))$$
$$CR_d(F_d(p_{d_k})) \;=\; \bigcap_{p_d \in \{p \mid p \in CP_d \,\wedge\, L_d(p) = F_d(p_{d_k})\}} CR_d(L_d(p_d))$$

Then for the correctness condition of the design to stay valid, we need to redefine the state equivalence as follows:

$$\boxed{\begin{array}{l} \forall s \in CS_b : s \equiv B_s(s) \Leftrightarrow \\ \qquad (S0_b \equiv S0_d \Rightarrow \\ \qquad\qquad (\forall e \in EP_b, s = T_b(e), \forall r \in CR_b(s) : \\ \qquad\qquad\qquad\qquad V_b(e,r) = V_d(B_e(e), B_r(r, B_s(s))))) \end{array}}$$

This technique is called criticality masking technique, since if at a certain state, the content of a register does not affect the correct function of the design, during the verification process, that register and its content at that state are ignored, or in other words its criticality at that state is masked.

6.4 Conclusion

In this chapter we presented the key elements of our verification approach in a formal setting. We revisited the notion of correctness formally, and presented a mathematical formulation of the problem decomposition discussed in Chapter 4. We presented a formal proof for correctness of the proposed decomposition and showed that a set of smaller and simpler design correctness lemmas can collectively capture the correctness condition of an RTL design. Based on this decomposition the proof of correctness of a design is reduced to the proof of design correctness lemmas. Finally, we presented extensions to our verification method to account for the verification of the designs when register optimization is performed as part of the synthesis process. In the following chapter we will explain how to automatically generate the proof of the design correctness lemmas.

Chapter 7

Proof Construction

In this chapter the construction of the proof of correctness for synthesized RTL designs is discussed. In Chapter 3 we defined correctness as an equivalence relation between the RTL design and its behavior specification. The correctness condition of the design can be defined based on this equivalence relation, that may be formulated as a theorem in logic (Equivalence Theorem). Then, to verify the correctness of the design a formal proof of this theorem should be generated.

As this theorem requires a very large proof - even for very small designs - constructing such a proof is usually impractical. To deal with the complexity of the proof generation, we introduced a decomposition for the RTL design and its specification. As a result of this decomposition, the behavior specification is partitioned into a set of simpler partial behavior descriptions, such that the operation flow of each partial behavior description is defined by a directed acyclic subgraph of the operation flow graph of the original behavior specification referred to as a behavior critical path. Also, as a result of this decomposition, the RTL design with the data-path D and controller C is partitioned into a set of simpler communicating RTL units, so that each RTL unit inherits the data-path D and its control flow is defined by a directed acyclic sub-graph of C called a design critical path.

We showed that there is a bijective relation between the partial behavior descriptions and the RTL units, such that each RTL unit is an implementation of a partial behavior description. Also, we showed that the proof of correctness of the RTL design is reducible to the proof of correctness of the RTL units. In this process, the equivalence of each RTL unit and the corresponding partial behavior description is verified. That is to say, the correctness condition of the design is reduced to the conjunction of a group of simpler correctness conditions - the critical path equivalence lemmas.

We saw that each correctness condition of the design corresponds to the equivalence of a pair of behavior and design critical paths, and may be formulated as a theorem in logic. In this discussion

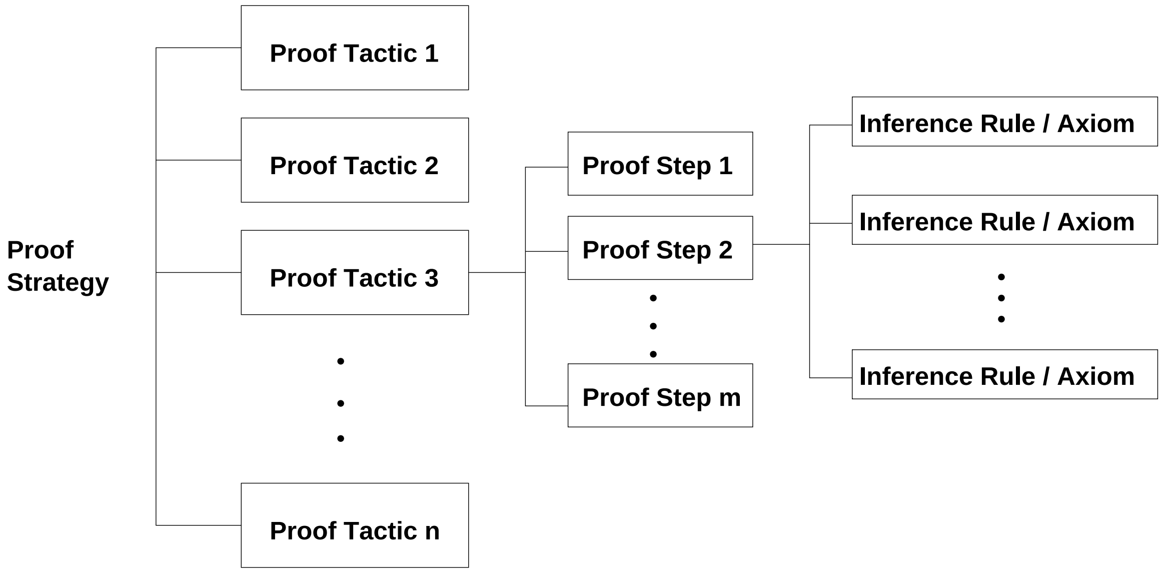


Figure 7.1: **Hierarchical proof generation**

such a theorem is referred to as a *lemma*. Now, the proof construction process consists of generating the formal proof of correctness for each correctness condition lemma.

The proof of correctness of each lemma is generated by simplifying it into subgoals, then proving each subgoal using the inference rules of logic, and the specific knowledge of the design. The correctness proofs may be fully hand-crafted; constructed interactively (such that the general steps are given by the verification engineer and the small steps are automatically generated); or they may be carried out automatically. In our approach the process of proof generation is completely automated.

Even though it will be a long while before the proof of arbitrary theorems of logic can be automatically generated, for specific classes of the problems there is the possibility of automating the proof effort, at least partially. In hardware verification domain as an example, if verification engineers adopt a specific style for constructing the proof, they find that they do similar steps over and over again [3]. By a careful study, those steps that are common to such proofs may be identified, and an assessment of the amount of reasoning amenable to automation can be made. Then, engines for automatic proof generation can be developed.

We have developed a completely automated engine for generating the proof of each correctness condition of an RTL design (i.e. the proof of each critical path equivalence lemma). These proofs consist of many repetitive steps. We have identified these steps and developed a general strategy for proving the critical path lemmas. Each general proof strategy consists of a sequence of more specific proof tactics. Each proof tactic may in turn consist of several proof steps (Figure 7.1). Each proof step is composed by application of inference rules to the design axioms. A design axiom captures part of our knowledge about the design or design properties.

It should be noted that there is always a tradeoff between the intricacy of proof constructs and the possibility of automatically generating the proofs. Between the two criteria we have given preference to the latter. Smart and sophisticated proofs may occasionally be generated automatically, however, this is not usually the case. The primary goals of our proof generation algorithm have been to generate 'a' proof for each critical path lemma and to construct this proof fully automatically.

To achieve this goal, we have aimed at very basic proof strategies and primitive proof tactics. This is due to the fact that we have emphasized the completeness of the approach rather than elegance of the proofs. Therefore, we can generate basic - and often very long - proofs for design correctness conditions in each and every case, rather than generating elegant proofs for some of the correctness conditions.

7.1 Proof Strategy

A strategy is a very high-level sketch of the proof without mentioning how the details are carried out. For example, a strategy for proving the a design correctness conditions, or a critical path equivalence lemma, may be as described by the following steps:

(1) extracting a relational formulation of the behavior of the specification,

(2) extracting a relational formulation of the RTL design,

(3) proving that both designs have equivalent behaviors.

Consider the behavior and design critical paths p_b and p_d representing the operation flow of a partial behavior description and an RTL unit, respectively. Let's assume that the communication of the of the partial behavior description with the environment is defined as follows:

- At the initial state of p_b all the critical variables are read from appropriate inputs.
- At the final state of p_b all the critical variables are written to appropriate outputs.
- If at a state s_b of p_b that the behavior specification reads/writes a value from/to a variable r_b to/from the environment, at the state s_b the partial behavior specification also reads/writes the same value from/to the variable r_b to/from the environment.

Similarly, the behavior of the RTL unit with controller p_d is defined as:

- At the initial state of p_d all the critical registers are read from appropriate inputs.
- At the final state of p_d all the critical registers are written to appropriate outputs.
- If at a state s_d of p_d that the RTL design reads/writes a value from/to a critical register r_d to/from the environment, then at the state s_d the RTL unit corresponding to p_d also reads/writes the same value from/to the critical register r_d to/from the environment.

As the first step in this proof strategy, the behavior of the partial behavior specification should be formulated mathematically. Part of these formulations may be similar to the following:

$$V_b(CR_b, L_b(p_b)) = f(V_b(CR_b, F_b(p_b)))$$

Similarly, at the second step of the proof strategy, the behavior of the RTL unit should be mathematically formulated:

$$V_d(CR_d, L_d(p_d)) = g(V_d(CR_d, F_d(p_d)))$$

Where the function V_b (or V_d) is overloaded in these definitions to denote a vector version of itself. Therefore, $V_b(CR_b, bs) \overset{B_r}{\equiv} V_d(CR_d, ds)$ should be interpreted as:

$$\forall r_b \in CR_b,\ \forall r_d \in CR_d \ : \ r_d = B_r(r_b) \ \Rightarrow \ V_b(r_b, b_s) \ \equiv \ V_d(r_d, d_s)$$

As the third step of the proof strategy, we need to show that the outputs of the two design are in the exact same relation with their inputs. Since we know that the inputs at the initial states of p_b and p_d are equivalent, then the outputs are proven to be equivalent. As part of this proof we may need to show:

$$\forall r_b \in CR_b \ : \ f(V_b(r_b, F_b(p_b))) = g(V_d(B_r(r_b), F_d(p_d)))$$

7.2 Proof Tactics

A proof strategy is an overall sketch of the proof, and does not offer any details. For example, the strategy sketched above, does not explain how each of the three steps of the proof should be carried out. Proof tactics simplify the general proof strategy into sub-goals.

$$
\begin{array}{lll}
\textbf{2D\_Delay\_Multiplier\_ax}: & \forall\ (in_1: & signal, \\
 & \quad in_2: & signal, \\
 & \quad out: & signal, \\
 & \quad s_1: & RTL\_state, \\
 & \quad s_2: & RTL\_state):
\end{array}
$$

$$
\begin{aligned}
&(2D\_delay\_muliplier(in_1, in_2, out)\ \wedge \\
&\qquad trans(s_1, s_2)\ \wedge \\
&\qquad\qquad Val_d(in_1, s_2) = Val_d(in_1, s_1)\ \wedge \\
&\qquad\qquad\qquad Val_d(in_2, s_2) = Val_d(in_2, s_1))\ \Rightarrow \\
&\qquad\qquad\qquad\qquad Val_d(out, s_2) = mul(Val_d(in_1, s_1), Val_d(in_2, s_1))
\end{aligned}
$$

Figure 7.2: **Relational description of the multiplier component**

For example for extracting a relational description of the behavior of specification (the first step in proof strategy), a proof tactic may be to traverse the path of data-transfers from the inputs to each output, and then extract the relational behavior of each component, then through term rewriting and simplifications the outputs may be defined in terms of the inputs. Therefore, in this case proof tactics may be to construct the relational behavior of the components from the description of this design, or to extract the interconnection information of two components, and then extract the following relation:

$$V_d(CR_d, L_d(p_d)) = g(V_d(CR_d, F_d(p_d)))$$

7.3 Proof Steps

Consider the description of the multiplier given in Figure 7.2. This multiplier has a delay of about 2 cycles, such that the output of the multiplier at the end of the second cycle, is the product of its two inputs at the first cycle. However, in order for the multiplier to function correctly, a few conditions should be met. Among these conditions, are the following two that state that the inputs of the multiplier in the two cycles of its operation should be stable:

$$(1)\quad V_d(in_1, s_2)\ =\ V_d(in_1, s_1)$$

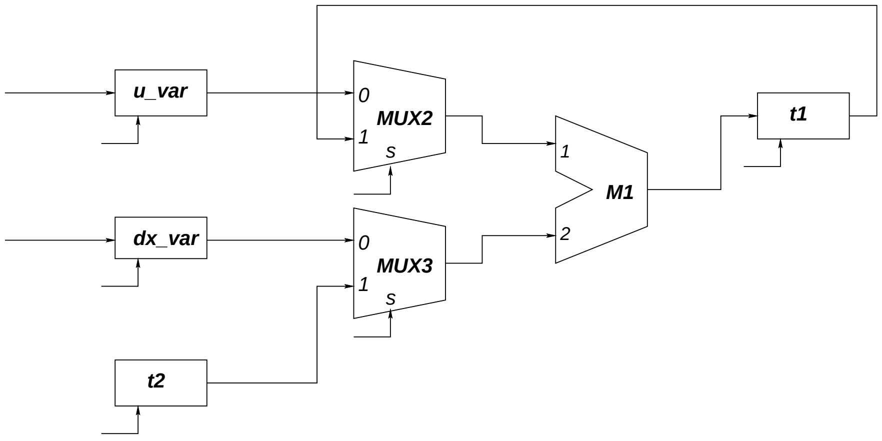


Figure 7.3: **Component interconnections as part of the data-path of an RTL design**

$$(2) \quad V_d(in_2, s_2) \;=\; V_d(in_2, s_1)$$

A proof step may be as simple as proving one condition such as above. Let's assume that the component M_1 in Figure 7.3 is a multiplier whose behavior is described by the $2D\_Delay\_Multiplier\_ax$ axiom. Then a proof step during the verification of the design may consist of establishing that the first condition for the correct operation of M_1 at the execution cycle $[ds_1, ds_2]$ is correct:

$$V_d(M_1.in_1, ds_2) \;=\; V_d(M_1.in_1, ds_1)$$

The proof of this condition is constructed from design axioms and the inference rules of higher order logic. This proof step is as follows:

(63) $V_d(M_1.in1, ds_1) = V_d(MUX_2.out, ds_1)$
(64) $V_d(MUX_2.out, ds_1) = V_d(MUX_2.in0, ds_1)$
(65) $V_d(MUX_2.in0, ds_1) = V_d(U\_REG.out, ds_1)$
(66) $V_d(M_1.in1, ds_1) = V_d(U\_REG.out, ds_1)$ (63)-(65)

(67) $V_d(M_1.in1, ds_2) = V_d(MUX_2.out, ds_2)$
(68) $V_d(MUX_2.out, ds_2) = V_d(MUX_2.in0, ds_2)$
(69) $V_d(MUX_2.in0, ds_2) = V_d(U\_REG.out, ds_2)$
(70) $V_d(U\_REG.out, ds_2) = V_d(U\_REG.out, ds_1)$

```
ENTITY 2d_multiplier_8bit IS
   PORT(input1 : IN bit_vector(7 downto 0);
        input2 : IN bit_vector(7 downto 0);
        prod : OUT bit_vector(15 downto 0)
       );
END 2d_multiplier_8bit;

m1 : 2d_multiplier_8bit
   PORT MAP (m1.in1(7 downto 0), m1.in2(7 downto 0), m1.out(15 downto 0));
```

Figure 7.4: **HDL description of the port-map of the multiplier component**

(71) $V_d(U\_REG.out, ds_1) = V_d(M_1.in1, ds_1)$ (66)
(72) $V_d(M_1.in1, ds_2) = V_d(M_1.in1, ds_1)$ (67)-(71)

7.4 Inference Rules and Axioms

Axioms are the translation of our knowledge about the design and its specification into logical formulas. For example, the port-map (interface) of a multiplier component of the RTL design, such as M_1 may be described as shown in Figure 7.4 in an HDL description of the data-path. During the process of modeling and specification of the design, this piece of information is translated to an axiom:

$$(1) \quad M_1\_ax \quad : \quad 2D\_Delay\_Multiplier(M_1.in_1, M_1.in_2, M_1.out)$$

The predicate $2D\_Delay\_Multiplier(M_1.in_1, M_1.in_2, M_1.out)$ asserts that $M_1.in_1$, $M_1.in_2$, and $M_1.out$ define the interface of a multiplier component of the type $2D\_Delay\_Multiplier$. Note that the bit sizes of the input and output signals are omitted as the values are uninterpreted in our technique. Now let's assume that the HDL decription of the controller may state that a transition from the ds_1 to ds_2 occurs. This information may also be translated to an axiom:

$$(2) \quad trans\_ds_1\_ds_2\_ax \quad : \quad transition(ds_1, ds_2)$$

Now, let's assume that the following conditions for the correct function of M_1 at the execution cycle $[ds_1, ds_2]$, have already been proved, in two separate proof steps:

$$(3) \quad V_d(M_1.in_1, ds_2) = V_d(M_1.in1, ds_1)$$
$$(4) \quad V_d(M_1.in_2, ds_2) = V_d(M_1.in2, ds_1)$$

During the verification of a design, the following sub-goal may be generated:

$$(5) \quad V_d(M_1.out, ds_2) = mul(V_d(M_1.in1, ds_1), V_d(M_1.in2, ds_1)$$

The proof of this sub-goal may be generated in a proof step, from the axioms specifying the design, previously proven goals, and inference rules of higher order logic. For example the $\forall$-elimination inference rule may be applied to $2D\_Delay\_Multiplier\_ax$ axiom, in_1 may be bound to $M_1.in_1$, in_2 to $M_1.in_2$, out to $M_1.out$, s_1 to ds_1 and s_2 to ds_2. As a result the following tautology is generated:

$$\begin{aligned}
&(2D\_delay\_muliplier(M_1.in_1, M_1.in_2, M_1.out) \ \wedge \\
&\qquad trans(ds_1, ds_2)) \ \Rightarrow \\
&\qquad\qquad ((V_d(M_1.in_1, ds_2) = V_d(M_1.in_1, ds_1) \ \wedge \\
&\qquad\qquad\qquad V_d(M_1.in_2, ds_2) = V_d(M_1.in_2, ds_1)) \ \Leftrightarrow \\
&\qquad\qquad\qquad\quad V_d(M_1.out, ds_2) = mul(V_d(M_1.in_1, ds_1), V_d(M_1.in_2, ds_1)))
\end{aligned}$$

From the axioms (1) and (2), and the previously proven sub-goals (3) and (4) we know that the antecedent of the above tautology is true. Then, by applying the $\Rightarrow$-elimination, the following tautology is generated:

$$V_d(M_1.out, ds_2) = mul(V_d(M_1.in1, ds_1), V_d(M_1.in2, ds_1)$$

This completes the proof of the sub-goal (5). The above example shows how during a proof step, the proof of a sub-goal is constructed.

7.5 Conclusion

In this chapter we discussed the construction of the proof of correctness for synthesized RTL designs. We showed that the correctness of an RTL design is established by verifying a set of critical-path equivalence lemmas. The proof of correctness of each lemma is generated by simplifying it into subgoals, then proving each subgoal using the inference rules of logic and design axioms. As the proofs consist of many repetitive steps, we have developed a completely automated engine for generating these proofs by constructing proof strategies consisting of repetitive proof steps. In the following chapter we introduce CCG (correctness condition generator), a formal verification tool based on the concepts presented in this dissertation. CCG has a proof construction engine that generates proof scripts as discussed in this section.

Chapter 8

Correctness Condition Generation

This chapter introduces the Correctness Condition Generator CCG, our formal verification tool that is built based on the principles discussed in previous chapters of this dissertation. CCG has powerful engines that perform model extraction and specification, correctness condition generation, and proof generation automatically. The resulting correctness proofs are further verified by an independent proof checker.

CCG is tightly integrated with the high-level synthesis system DSS [53], and serves to verify the correctness of the RTL designs generated by this system. We modified the high-level synthesis system DSS to generate the three bindings B_r, B_s and B_p. These bindings along with the behavior specification and the RTL design description are the inputs to the Correctness Condition Generator (CCG) shown in Figure 8.1. CCG processes the knowledge of RTL design and its behavior specification and generates formal descriptions of the specification and the design, correctness conditions of the design and their proof of correctness.

The proof is carried carried out in PVS theorem proving environment [48], and verified by PVS proof checker [56]. The choice of PVS came naturally to us, for two reasons: PVS specification language [47] is based on higher-order logic, and PVS proof checker is powerful. PVS proof checker examines the proof and detects the inconsistencies or violations of the inference rules, if they exist.

The failure of the proofs may be due to two reasons: logical errors in construction of the proof, or design errors. In case of failure, the source of the error can be identified. During our experiments we did not encounter errors of the former type. We believe that the use of primitive logic constructs and basic inference rules in construction of the proof has considerably contributed to this fact.

CCG is characterized by the ability to locate the errors in addition to detecting them. The immediate implication of the failure of the (correctly constructed) proof of a critical path equivalence lemma is that along that critical path the design has incorrect behavior. In such situation, the

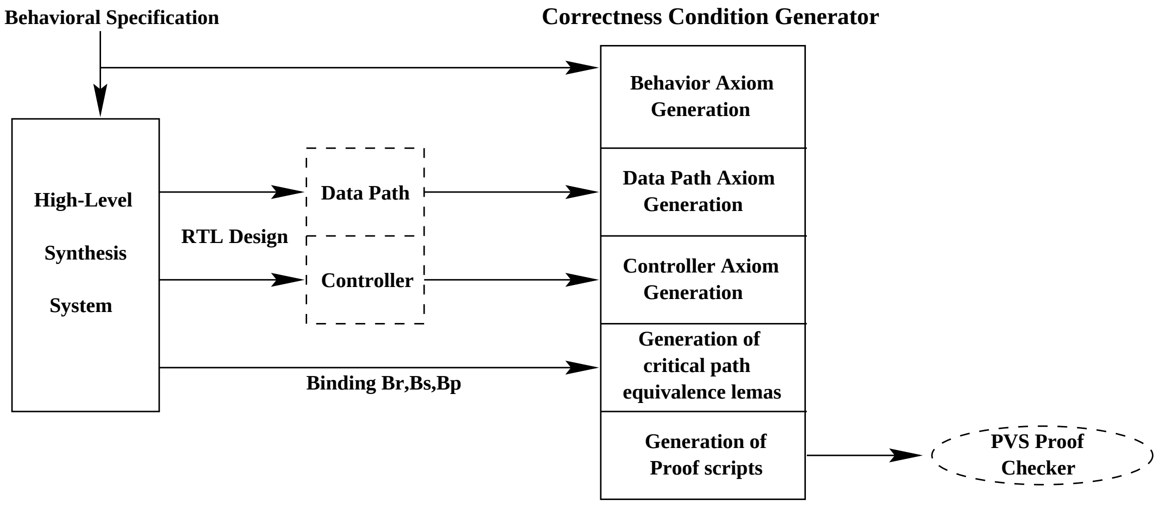

Behavioral Specification
Correctness Condition Generator
Behavior Axiom Generation
High-Level Synthesis System
RTL Design
Data Path
Controller
Data Path Axiom Generation
Controller Axiom Generation
Generation of critical path equivalence lemas
Binding Br,Bs,Bp
Generation of Proof scripts
PVS Proof Checker

Figure 8.1: **Stages of Correctness Condition Generation**

control traces along which the design has erroneous behavior, and in each case the route of the data transfers leading to error may be identified. Consequently, the circuitry involved in erroneous operations is pointed out. Even though CCG cannot determine the exact point of error, it localizes the neighborhood of the error in data-path. The localization of the error is invaluable in error recovery.

The goal of our proof effort is to show that the behavior specification and RTL implementation are equivalent. This is accomplished by decomposition of the proof of equivalence into proof of a set of lemmas as discussed in previous chapter. The proof of these lemmas together complete the proof of the equivalence of the two designs. Each lemma states that a critical path in the behavior is equivalent to its corresponding critical path in the structure. We assume that the operating environment of the designs ensures that $S0_b \equiv S0_d$. Typically, the environment ensures that all the data and control registers are reset at the start states. We develop the proof by using symbolic term rewriting in a higher-order logic theorem prover.

CCG has five five engines that each performs a part of the automated design verification effort. Each of the following tasks is performed by one of the engines:

1. behavior axiom generation;
2. data path axiom generation;
3. controller axiom generation;
4. generation of critical path equivalence lemmas; and,
5. proof generation.

```
equivalent_states : [beh_state,rtl_state -> bool]
beh_trans : TYPE+ = [# bs1 : beh_state, bs2 : beh_state , bc : bool #]
beh_critical_path : TYPE+ = list[beh_trans]
Val_b : [spec_var, beh_state -> value]
flag : [rtl_state -> bool]
beh_transition : PRED[beh_trans]
source_beh_trans(t : beh_trans) : beh_state = bs1(t)
target_beh_trans(t : beh_trans) : beh_state = bs2(t)
B_s : [beh_state -> rtl_state]
B_r : [spec_var -> comp_out]
B_p : [beh_critical_path -> rtl_critical_path]
First_b(cp : beh_critical_path) : beh_state = source_beh_trans(car(cp))
Last_b(cp  : beh_critical_path) : beh_state = target_beh_trans(car(reverse(cp)))
beh_transition_condition(cp : beh_critical_path) : bool = beh_transition(car(cp))
% similar declarations for RTL design go here.
```

Figure 8.2: **Some Generic Declarations**

Each of these tasks will be discussed in this chapter in details. We illustrate these tasks by showing selected fragments of the PVS code produced by CCG.

8.1 Generic Axioms

CCG has a library of generic declarations. These declarations are the prototypes of various functions, predicates, etc. that are used in the process of modeling and proof generation of all designs. The axioms of this library are used by the proof generator engine to construct proofs. In addition CCG has a library of synthesis resources axioms. These axioms formally describe the relational behavior (the behavior as a relation between inputs and outputs) of each component in the library of synthesis resources.

As the first step in axiom generation, CCG generates a set of design specific definitions. The latter set of axioms include information about specification variables, the RTL component declaration, critical path specification for behavior automaton and controller, and bindings B_r, B_s and B_p. Figures 8.2 and 8.3 show some of the generic declarations and some of the design-specific elements of the PVS model, respectively.

8.2 Behavior Axiom Generation

The first engine of CCG performs the tasks at this stage. As the first step it extracts a finite state machine model from the behavior specification, written in a simple subset of VHDL in our case. Then it examines the state machine model of the behavior specification, and converts it into a series of

```
spec_var : TYPE+ = { max, sum, val, grt }
comp_out : TYPE+ = { MAX_out, SUM_out, VAL_out, GRT_out, B1_out, B2_out,
                     OU_out, T1_out, T2_out, Z_out, RTL_input }
bt1  : beh_trans = (# bs1 := BS0 , bs2 := BS1 , bc := TRUE #)
bt2  : beh_trans = (# bs1 := BS1 , bs2 := BS2 , bc := TRUE #)
dt1  : rtl_trans = (# ds1 := DS0 , ds2 := DS1 , dc := TRUE #)
dt2  : rtl_trans = (# ds1 := DS1 , ds2 := DS2 , dc := TRUE #)
bcp3 : beh_critical_path = (: bt7 :)
bcp4 : beh_critical_path = (: bt6, bt8 :)
dcp3 : rtl_critical_path = (: dt8 :)
dcp4 : rtl_critical_path = (: dt7, dt9, dt10 :)
Bs_BS0_ax : AXIOM B_s(BS0) = DS0
Bs_BS2_ax : AXIOM B_s(BS2) = DS1
Bs_BS5_ax : AXIOM B_s(BS5) = DS6
Br_max_ax  : AXIOM B_r(max) = MAX_out
Br_sum_ax  : AXIOM B_r(sum) = SUM_out
Br_val_ax  : AXIOM B_r(val) = VAL_out
Br_grt_ax  : AXIOM B_r(grt) = GRT_out
Bp_bcp1_ax : AXIOM B_p(bcp1) = dcp1
Bp_bcp2_ax : AXIOM B_p(bcp2) = dcp2
Bp_bcp3_ax : AXIOM B_p(bcp3) = dcp3
Bp_bcp4_ax : AXIOM B_p(bcp4) = dcp4
state_equivalence : AXIOM (FORALL (bs : beh_state, ds : rtl_state):
   equivalent_states(bs,ds)
      IFF
   (ds = B_s(bs) AND
   Val_b(max,bs) = Val_d(MAX_out,ds) AND
   Val_b(sum,bs) = Val_d(SUM_out,ds) AND
   Val_b(val,bs) = Val_d(VAL_out,ds) AND
   Val_b(grt,bs) = Val_d(GRT_out,ds)))
```

Figure 8.3: **Some Basic Design-Specific Definitions and Axioms**

axioms that collectively specify the value transfers in the behavior. This second procedure is mainly a compilation task. For each state transition in the behavior design one axiom is generated. This axiom specifies the value of each specification variable after a transition, in terms of the values of that and other specification variables prior to that transition. Note that these axioms offer an implicit symbolic evaluation of specification variables after each transition.

Figure 8.4 shows the axioms for the critical path $[BS_5, BS_6, BS_2]$. This critical path consists of two transitions $t6 = \langle BS_5, BS_6\rangle$ and $t8 = \langle BS_6, BS_2\rangle$. For each transition, one axiom is generated. Val_b corresponds to state transition function δ_b.

The axiom $bs_5\_bs_6\_ax$ for example, as the name suggests corresponds to behavior state transition $bt_6 = \langle BS_5, BS_6\rangle$. This axiom defines the value of the specification variables at state BS_6, after transition $\langle BS_5, BS_6\rangle$, in terms of their values at state BS_5. This axiom is read: "In transition bt_6, from behavior state BS_5 to behavior state BS_6, the values of all specification variables remain unchanged". The axiom $bs_6\_bs_2\_ax$ is read: "In a transition bt_8 from behavior state BS_6 to behavior state BS_2, all specification variables preserve their values, except the specification variable *max*, that assumes the value of specification variable *val* prior to transition".

```
trans_bs5_bs6_ax : AXIOM beh_transition(bt6) = (Val_b(grt,BS5) = ONE)

bs5_bs6_ax : AXIOM  beh_transition(bt6)
        IMPLIES
            (Val_b(max,BS6) = Val_b(max,BS5) AND
               Val_b(sum,BS6) = Val_b(sum,BS5) AND
                 Val_b(val,BS6) = Val_b(val,BS5) AND
                   Val_b(grt,BS6) = Val_b(grt,BS5))

bs6_bs2_ax : AXIOM beh_transition(bt8)
        IMPLIES
            (Val_b(max,BS2) = Val_b(val,BS6) AND
               Val_b(sum,BS2) = Val_b(sum,BS6) AND
                 Val_b(val,BS2) = Val_b(val,BS6) AND
                   Val_b(grt,BS2) = Val_b(grt,BS6))
```

Figure 8.4: **Axioms for Some Behavior Transitions**

8.3 Data Path Axiom Generation

A pre-existing library of axioms defines the behavior of each type of RTL component. As it was mentioned before, each axiom of this library formally describes the relational behavior of an RTL component. The value at the output of a component at a particular state is defined in terms of the data and control inputs of the component at that state, or its output at a previous state in the case of sequential components. The input of a component can be the output of some other component or a primary input. Figure 8.5 shows the axiomatic behavior descriptions for the register, ALU and bus components. These axioms are not design specific, and are included as part of the specification of all designs as necessary.

The second engine of correctness condition generator CCG, models the data-path of the synthesized RTL design as a PVS theory [47]. For each component of the data-path, an axiom is generated, that specifies its type, and its interface with the rest of the components. The data-path axioms together with the axioms in the component library define the behavior and interface of each individual component. Also, the interconnection of the control inputs of the RTL components with the controller, and the interconnection of the flags from the data-path to the controller are specified in this theory. Axiomatic specifications of some of the data-path components of our design example are shown in Figure 8.6.

The axiom *grt_ax*, for example, specifies the type and interface of the register *GRT*. This axiom states that a register component belongs to the data path, whose input is connected to the output of component B_2 (the second bus), its output is *GRT_out* and its load signal is *GRT_cs*. The predicate *register* with the parameters *B2_out*, *GRT_out* and *GRT_cs* asserts that these three signals are the interface ports of a register as stated above.

The axiom *flag_ax* specifies the interconnection of a controller flag to the data-path. This axiom

```
register  : [comp_out, comp_out, signal[bool] -> bool]
alu       : [comp_out, comp_out, comp_out, signal[bool] -> bool]
bus       : [arr[comp_out], comp_out, arr[signal[bool]] -> bool]

reg_ax : AXIOM (FORALL (c_in       : comp_out,
                        c_out      : comp_out,
                        load       : signal[bool],
                        s1         : rtl_state,
                        s2         : rtl_state):
register(c_in,c_out,load) AND (s2 = next_state(s1)
      IMPLIES
         IF (load(s2)) THEN
            Val_d(c_out,s2) = Val_d(c_in,s2)
         ELSE
            Val_d(c_out,s2) = Val_d(c_out,s1)
         ENDIF))

alu_ax : AXIOM (FORALL (c_in1, c_in2, c_out   : comp_out,
                        op_mode               : signal[bool],
                        ds                    : rtl_state) :
alu(c_in1,c_in2,c_out,op_mode)
      IMPLIES
         IF (op_mode(ds)) THEN
            Val_d(c_out,ds) = add(Val_d(c_in1,ds),Val_d(c_in2,ds))
         ELSE
            Val_d(c_out,ds) = gt(Val_d(c_in1,ds),Val_d(c_in2,ds))
         ENDIF)

bus_ax  : AXIOM (FORALL (c_ins      : arr[comp_out],
                         c_out      : comp_out,
                         wrs        : arr[signal[bool]],
                         ds         : rtl_state,
                         index      : nat) :
bus(c_ins,c_out, wrs)
      IMPLIES
         (wrs(index)(ds) => (Val_d(c_out,ds) = Val_d(c_ins(index),ds))))
```

Figure 8.5: **Axioms for Some RTL Library Components**

states that the controller flag named *flag*, is in fact the output of the data-path component *GRT* (the register *GRT*).

The axiom *grt_cs_ax* specifies the interconnection of a control-signal and a data-path component's control input. This axiom states that the control input of the component *GRT* (the register load) is connected to the controller signal cs_0.

These 3 axioms specify the interface of the register component *GRT* to the rest of the RTL design. Also, the predicate *register* with three parameters B_2*_out*, *GRT_out* and *GRT_cs* together with the *reg_ax* axiom of the RTL component library, specify the behavior of the component *GRT* as a relation between its inputs (data input B_2*_out* and control input *GRT_cs*) and its output *GRT_out*.

```
GRT_cs, SUM_cs, MAX_cs, VAL_cs, T1_cs, T2_cs, OU_cs :  signal[bool]
B1_wrs      : arr[signal[bool]]
B1_ins      : arr[comp_out]
grt_cs_ax : AXIOM (FORALL (s : rtl_state) : GRT_cs(s) = Control_Signal(s,0))
b1_wrs_ax1 : AXIOM (FORALL (s : rtl_state) : B1_wrs(1)(s) = Control_Signal(s,7))
b1_ins_ax1 : AXIOM (B1_ins(1) = SUM_out)
flag_ax : AXIOM (FORALL (s : rtl_state) : flag(s) IFF (Val_d(GRT_out,s) = ONE))
grt_ax : AXIOM register(B2_out,GRT_out,GRT_cs)
z_ax : AXIOM constant_register(Z_out,ZERO)
ou_ax : AXIOM alu(T1_out,B1_out,OU_out,OU_cs)
b1_ax : AXIOM bus(B1_ins,B1_out,B1_wrs)
```

Figure 8.6: **Axioms for Some Data Path Components**

8.4 Controller Axiom Generation

At this step a constructive model of the RTL controller is generated. First, the third engine of CCG extracts a finite state machine model of the conroller of the RTL design. Then, it extracts the functions *next_state*, and *Control_Signal* from the finite state machine model of the controller, and translates them to PVS descriptions. In function *Control_Signal*, the values of control signals from the controller to data-path at each state of the controller are defined. In function *next_state*, the next state of each state is determined, in terms of the current state and the values of the flags from the data-path to the controller at that state. This function corresponds to the function σ_d in our implementation model. The function *next_state* and parts of the function *Control_Signal*, for our design example are shown in Figure 8.7.

```
next_state(s: rtl_state) : rtl_state =
   CASES s OF
      DS0 : DS1,
      DS1 : DS2,
      DS2 : DS3,
      DS3 : DS4,
      DS4 : DS5,
      DS5 : DS6,
      DS6 : IF (flag(DS6)) THEN DS7
            ELSE DS1
            ENDIF,
      DS7 : DS8,
      DS8 : DS1
   ENDCASES
Control_Signal(s: rtl_state, id: index) : bool =
   IF id=0 THEN
      CASES s OF
         DS5    : TRUE
         ELSE     FALSE
      ENDCASES
   ...
   ELSE
      FALSE
   ENDIF
```

Figure 8.7: **RTL Controller**

```
    eq_cp4_l1 : LEMMA
       (equivalent_states(First_b(bcp4),First_d(B_p(bcp4))) AND
       beh_transition_condition(bcp4) AND
       rtl_transition_condition(B_p(bcp4)))
          IMPLIES
       Val_b(max,Last_b(bcp4)) = Val_d(MAX_out,Last_d(B_p(bcp4)))
    eq_cp4 : LEMMA
       (equivalent_states(First_b(bcp4),First_d(B_p(bcp4))) AND
       beh_transition_condition(bcp4) AND
       rtl_transition_condition(B_p(bcp4)))
          IMPLIES
       equivalent_states(Last_b(bcp4),Last_d(B_p(bcp4)))
```

Figure 8.8: **Some Critical Path Equivalence Lemmas**

8.5 Generation of Critical Path Equivalence Lemmas

For each pair of the behavior-RTL critical paths which are bound by the function B_p, a set of lemmas are generated. A main lemma states that if the initial states of the pair of the critical paths are equivalent, their final states should be equivalent. An instance of such statement for the behavior critical path $bcp_4 = [BS_5, BS_6, BS_2]$ and its RTL counterpart $dcp_4 = B_p(bcp_4) = [DS_6, DS_7, DS_8, DS_1]$ is the lemma *eq_cp*4 in Figure 8.8. A set of sub-lemmas (one sub-lemma for each specification variable v) state that if the initial states of the pair of critical paths are equivalent, then the values stored in the specification variable v and its RTL counterpart $B_r(v)$ at the final states of the critical paths are the same. The proof of these sub-lemmas together infer the proof of the main lemma for the specific critical path. An instance of such a sub-lemma for the specification variable max, and critical paths $\langle bcp_4, dcp_4 \rangle$ is the lemma *eq_cp*4*_l*1 in Figure 8.8. Similar sub-lemmas for other critical variables sum, val and grt are generated. These four sub-lemmas are used for proving the main lemma, *eq_cp*4. The equivalence of the behavior specification and the RTL design is established by proof of these lemmas for all critical paths of the designs.

8.6 Generation of Proof Scripts

The generation of the proof scripts is the most elaborate task of CCG. At this step all the information about the RTL design and its behavioral specification is processed and the rules for proving the above mentioned lemmas are produced. The proof generation process was discussed in detail in Chapter 7.

During the proof generation process, the values of the specification variables and the contents of design registers are symbolically calculated. The symbolic values of specification variables after each state transition are implicitly defined by behavior axioms. Therefore, by inclusion as proof rules, the axioms corresponding to the behavior transitions belonging to the same critical path, the symbolic values of the specification variables at the final state of the critical path are implicitly defined in terms of their values at the initial state of the critical path. For example, the axiom *bs5_bs6_ax* defines the values of specification variables at state BS_6 and after the transition $\langle BS_5, BS_6\rangle$ in terms of their values at state BS_5, and the axiom *bs6_bs2_ax* defines the values of the specification variables at state BS_2 and after the transition $\langle BS_6, BS_2\rangle$ in terms of their values at state BS_5, therefore by including these two axioms as proof rules we have implicitly defined the values of specification variables at state BS_2, the final state of critical path $bcp_4 = [BS_5, BS_6, BS_2]$, in terms of their values at BS_5, the initial state of bcp_4. The two following rules which are part of the proof of the lemma *eq_cp4_l1* serve this purpose:

(LEMMA "bs6_bs2_ax")
(LEMMA "bs5_bs6_ax")

By including the proper behavior state transition axioms, the value of the specification variables at the final state of behavior critical paths are defined. Similarly, the values of the critical registers at the final state of design critical paths should be defined. Then, by comparing these sets of values, the equivalence of the critical paths of the designs can be verified.

The symbolic values of each critical register at the final state of a design critical path can be defined by a complete symbolic evaluation of the output of that register, i.e. calculating its value in terms of the value of the outputs of the critical design registers at the initial state of that critical path. CCG performs this symbolic analysis and as proof rules includes proper axioms that define the value of the output of components at desired state. For example, suppose we are interested in calculating the value of the register MAX_REG at DS_1, the final state of the critical path $dcp_4 = [DS_6, DS_7, DS_8, DS_1]$. Then CCG generates as a proof rule, an instance of the axiom *reg_ax* in the RTL component library, for register MAX_REG at the state DS_8. Therefore, the value of this register at state DS_1 is defined in terms of its input B_2_out at the current state DS_8, and its output MAX_out at previous state DS_1.

(LEMMA "reg_ax" ("c_in" "B2_out" "c_out" "MAX_out" "load" "MAX_cs" "s1" "DS8" "s2" "DS1"))
(ASSERT)

Since the load signal of MAX_REG at state DS_8 is asserted low, then the value of MAX_out at stateDS_1 is the same as its value at state DS_8. Therefore CCG generates proof rules that specify the values of the control signals at state DS_1 (to define MAX_{cs} the load input of MAX_REG at state DS_1) and an instantiation of axiom *reg_ax* of the RTL component library to define the input-output relation of MAX_REG at state DS_8.

(EXPAND "Control_Signal")
(ASSERT)
(LEMMA "reg_ax" ("c_in" "B2_out" "c_out" "MAX_out" "load" "MAX_cs" "s1" "DS8" "s2"

```
(AUTO-REWRITE-THEORY "des" :ALWAYS? T)
(FLATTEN)
(LEMMA "bs6_bs2_ax")
(LEMMA "bs5_bs6_ax")
(LEMMA " btc_bcp4_ax")
(PROP)
(LEMMA "Bp_bcp4_ax")
(REPLACE -1)
(EXPAND "beh_transition_condition")
(EXPAND "rtl_transition_condition")
(EXPAND "First_b")
(EXPAND "First_d")
(EXPAND "Last_b")
(EXPAND "Last_d")
(ASSERT)
(REPEAT (EXPAND "reverse"))
(REPEAT (EXPAND "append"))
(LEMMA "trans_ds6_ds7_ax")
(LEMMA "flag_ax" ("s" "DS6"))
(PROP)
(LEMMA "max_ax")
(LEMMA "reg_ax" ("c_in" "B2_out" "c_out" "MAX_out" "load" "MAX_cs" "s1" "DS8"
 "s2" "DS1"))
(ASSERT)
(EXPAND "Control_Signal")
(ASSERT)
. . . .
(ASSERT)
```

Figure 8.9: **Proof Scripts for a Correctness Theorem**

"DS1"))

This continues until the symbolic evaluation of MAX_out at DS_8 is complete. Proof scripts are generated by making use of the axioms and definitions generated in the previous steps when necessary. These proofs are then subjected to verification by the PVS proof checker. These proofs make extensive use of symbolic rewriting, involving instantiation of definitions, axioms and other proven lemmas. A portion of the proof script for the lemma *eq_cp4_l1* is shown in Figure 8.9.

Chapter 9

Structural Pipelining in Design Synthesis

Pipelining is used in circuit design to enhance the performance of systems. This technique can be exploited for the implementation of a design that performs a particular sequence of operations repeatedly. In such a design it is often possible to overlap the evaluation of operations in consecutive executions of the sequence. This can be achieved by partitioning the sequence of the operations into sub-sequences (called *stages*) such that (a) hardware circuits can be designed to execute the operations in each sub-sequence, (b) the outputs of the operations in each sub-sequence are the inputs to the operations in the succeeding sub-sequence, (c) other than the exchange of the inputs and outputs, there are no interrelationships between the operations in any two different sub-sequences, (d) usually, the execution times of sub-sequences of the operations in corresponding hardware circuits are approximately equal, and (e) the sequential evaluation of the sub-sequences and the evaluation of the original sequence generate equivalent results [34].

Even though pipelining does not affect the latency of the evaluation of the original sequence of operations λ, when this evaluation is performed repeatedly, it can increase the *throughput* δ_0, i.e. the rate at which the results of consecutive iterations are computed. If a design repeatedly executes a sequence of operations at the rate λ, a pipeline design with N_s stages can do the same at a rate up to $\lambda_p = \lambda / N_s$.

In synthesis of digital designs three different types of pipelining techniques may be used: structural pipelining, functional pipelining and loop winding. In the following two chapters of this dissertation, specific issues on synthesis of pipelined designs by exploiting these three techniques, and in each case the verification of the resulting pipelined designs is discussed. This chapter concentrates on structural pipelining, and the related topic of multi-cycle synthesis resources, and the following chapter focuses on loop winding and functional pipelining.

In the next section we first explain the general issues in design synthesis with multi-cycle resources, and then discuss the synthesis using pipelined resources (structural pipelining) that constitute a special class of multi-cycle resources.

9.1 Multi-Cycle Resources

The propagation delays of the most combinational components in synthesis resources library are much smaller than the clock cycles of synthesized designs. A set of values may be applied to the inputs of a component and propagated to the outputs after a short delay during the same clock cycle. For example, the behavior of a combinational multiplier that is used in a circuit with a bigger clock cycle compared to its propagation delay may be described as:

$$\begin{aligned} \mathit{multiplier\_axiom}: \ \forall(\mathit{in}_1 &: \mathit{signal}, \\ \mathit{in}_2 &: \mathit{signal}, \\ \mathit{out} &: \mathit{signal}, \\ \mathit{ds} &: \mathit{state}) : \end{aligned}$$

$$\mathit{multiplier}(\mathit{in}_1, \mathit{in}_2, \mathit{out}) \Rightarrow V_d(\mathit{out}, \mathit{ds}) = \mathit{mul}(V_d(\mathit{in}_1, \mathit{ds}), V_d(\mathit{in}_2, \mathit{ds}))$$

where *multiplier* is a predicate that is true iff in_1, in_2, and *out* are the interface of a combinational multiplier such as described above. *mul* is the uninterpreted multiplication function, and V_d is a function that defines the values at the input/output ports of components at each state. From the above description it is obvious that the output of this multiplier component replies to the input stimuli instantaneously or at least in less than one clock cycle.

However, in some designs the propagation delay of some combinational components that perform a certain operation may be longer than one clock cycle. Also, in some designs the user constraints may enforce the use of application specific resources that are not purely combinational or have long propagation delays for executing a particular operation. In either case, the delay associated with execution of operation exceeds one clock cycle. Such resources are referred to as *multi-cycle delay resources*.

The class of pipelined resources that will be discussed in detail in the following section is sequential in nature and belongs to this group of synthesis resources. Pipelined resources are distinguished from the other multi-cycle resources by the property that at the steady state of their operation

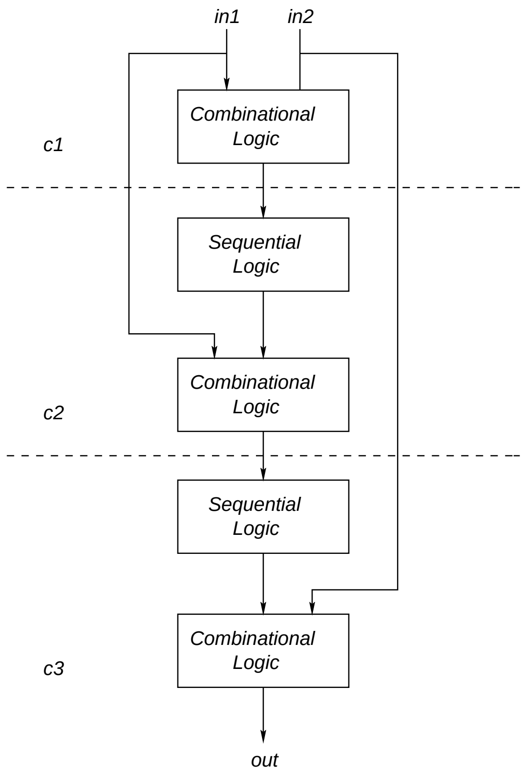


Figure 9.1: **The stages of a multi-cycle component**

they can consume values and produce results at every clock cycle. The non-pipelined multi-cycle resources on the other hand, usually require that the inputs stay stable throughout their operation cycle. A multi-cycle resource may perform an operation in several stages, where the duration of each stage is a multiple of the clock cycle. At each stage some intermediate results are generated and stored in sequential storage components.

An example of a non-pipelined multi-cycle resource with two inputs and one output is given in Figure 9.1. In this figure, an operation O is performed in three stages. The execution cycle of this component spans over three consecutive clock cycles, where each cycle corresponds to one stage of the operation. Since in this example, the result of the operations at each stage (except the first) depends on the result of the operation at the previous stage, as well as the input values, it is obvious that the inputs should be held stable throughout the operation cycle of the component. The stability of the inputs is a precondition for the correct function of the design.

In high-level design synthesis, multi-cycle resources including pipelined resources need special treatment. For example, when pipelined resources are used, the scheduling problem is more complex and requires specific solutions, that are different from the case where resources are not pipelined. However, we will show that in our verification method, the verification of designs with multi-cycle resources is similar to verification of designs without multi-cycle resources. The multi-cycle characteristic of a component is reflected in formal specification of its behavior. During the verification process, special properties of these designs are accounted for through these formal specifications. As an example, the multi-cycle resources may require some precondition to be met, in order to function correctly. During the verification process, it should be additionally verified that these preconditions hold. This notion can be further clarified through an example. The behavior of a multi-cycle multiplier component with two cycle delay may be described as:

$$\begin{array}{lrl} 2D\_Delay\_Multiplier\_ax: \ \forall & (in_1: & signal, \\ & in_2: & signal, \\ & out: & signal, \\ & ds_1: & RTL\_state, \\ & ds_2: & RTL\_state): \end{array}$$

$$\begin{array}{l} (2D\_delay\_muliplier(in_1, in_2, out) \ \wedge \\ \qquad transition(ds_1, ds_2)) \ \Rightarrow \\ \qquad\qquad ((V_d(in_1, ds_2) = V_d(in_1, ds_1) \ \wedge \\ \qquad\qquad\qquad V_d(in_2, ds_2) = V_d(in_2, ds_1)) \ \Leftrightarrow \\ \qquad\qquad\qquad\qquad V_d(out, ds_2) = mul(V_d(in_1, ds_1), V_d(in_2, ds_1))) \end{array}$$

In contrast to the simple combinational multiplier discussed in the previous section, the output of the multi-cycle multiplier responds to the input stimuli after one clock cycle. In addition, the precondition for correct function of this component is that its inputs be kept stable throughout the two consecutive cycles of its operation. If this precondition is not met, the relation between the output and inputs of the multiplier is no longer valid. This means that during the verification process, before we can define the output at a state ds_2 in terms of the inputs at the previous state ds_1, it should be verified that the two preconditions for correct operation of the multiplier are valid, i.e.:

$$\begin{array}{l} V_d(in_1, ds_2) = V_d(in_1, ds_1) \ \wedge \\ \qquad V_d(in_2, ds_2) = V_d(in_2, ds_1) \end{array}$$

In the following discussions we compare the synthesis and verification issues of pipelined and non-pipelined multi-cycle resources through examples. Given the same behavior specification we com-

```
(bs_1)    t1    := u_var * dx_var;
(bs_2)    t2    := 3 * x_var;
(bs_3)    t3    := 3 * y_var;
(bs_4)    t4    := t1 * t2;
(bs_5)    t5    := dx_var * t3;
(bs_6)    t6    := u_var - t4;
(bs_7)    u_var := t6 - t5;
(bs_8)    ...
```

Figure 9.2: **Partial specification adopted from the Differential Equation benchmark**

pare the implementation designs that are synthesized under different resource constraints and with pipelined or non-pipelined multi-cycle resources. In these examples, we consider the partial behavioral specification given in Figure 9.2 that is taken from a benchmark named Differential Equation. A formal description of this partial behavior specification is given in Figure 9.3. Note that the variables t_1, t_2, $\cdots$, t_6 in the complete Differential Equation Specification are not critical. However, for the purpose of self-containment of our examples, in this specification segment of the Differential Equation we consider them critical.

We have mentioned in previous discussions that one of the inputs to the synthesis tool is a set of user constraints. Resource constraints are one class of such constraints. Synthesis can be performed with or without resource constraints. In synthesis without resource constraints, the goal is that assuming unlimited resources available, generate a schedule that satisfies particular timing constraints such as an upper bound on latency. In synthesis with resource constraints, there is a limit on the number of each type of resource available or the total area of the synthesized design.

$$\begin{aligned}
\mathbf{bs_1\_bs_2\_ax} : transition(bs_1, bs_2) \Rightarrow\ & V_b(u\_var, bs_2) = V_b(u\_var, bs_1) \wedge \\
& V_b(x\_var, bs_2) = V_b(x\_var, bs_1) \wedge \\
& V_b(y\_var, bs_2) = V_b(y\_var, bs_1) \wedge \\
& V_b(dx\_var, bs_2) = V_b(dx\_var, bs_1) \wedge \\
& V_b(t_1, bs_2) = mul(V_b(u\_var, bs_1)\ ,\ V_b(dx\_var, bs_1)) \wedge
\end{aligned}$$

$$\begin{aligned}
V_b(t_2, bs_2) &= V_b(t_2, bs_1) \wedge \\
V_b(t_3, bs_2) &= V_b(t_3, bs_1) \wedge \\
V_b(t_4, bs_2) &= V_b(t_4, bs_1) \wedge \\
V_b(t_5, bs_2) &= V_b(t_5, bs_1) \wedge \\
V_b(t_6, bs_2) &= V_b(t_6, bs_1)
\end{aligned}$$

$$\mathbf{bs_2\_bs_3\_ax} : transition(bs_1, bs_3) \Rightarrow \begin{aligned}[t]
V_b(u\_var, bs_3) &= V_b(u\_var, bs_2) \wedge \\
V_b(x\_var, bs_3) &= V_b(x\_var, bs_2) \wedge \\
V_b(y\_var, bs_3) &= V_b(y\_var, bs_2) \wedge \\
V_b(dx\_var, bs_3) &= V_b(dx\_var, bs_2) \wedge \\
V_b(t_1, bs_3) &= V_b(t_1, bs_2) \wedge \\
V_b(t_2, bs_3) &= mul(const(3)\ ,\ V_b(x\_var, bs_2)) \wedge \\
V_b(t_3, bs_3) &= V_b(t_3, bs_2) \wedge \\
V_b(t_4, bs_3) &= V_b(t_4, bs_2) \wedge \\
V_b(t_5, bs_3) &= V_b(t_5, bs_2) \wedge \\
V_b(t_6, bs_3) &= V_b(t_6, bs_2)
\end{aligned}$$

$$\mathbf{bs_3\_bs_4\_ax} : transition(bs_3, bs_4) \Rightarrow \begin{aligned}[t]
V_b(u\_var, bs_4) &= V_b(u\_var, bs_3) \wedge \\
V_b(x\_var, bs_4) &= V_b(x\_var, bs_3) \wedge \\
V_b(y\_var, bs_4) &= V_b(y\_var, bs_3) \wedge \\
V_b(dx\_var, bs_4) &= V_b(dx\_var, bs_3) \wedge \\
V_b(t_1, bs_4) &= V_b(t_1, bs_3) \wedge \\
V_b(t_2, bs_4) &= V_b(t_2, bs_3) \wedge \\
V_b(t_3, bs_4) &= mul(const(3)\ ,\ V_b(y\_var, bs_3)) \wedge \\
V_b(t_4, bs_4) &= V_b(t_4, bs_3) \wedge \\
V_b(t_5, bs_4) &= V_b(t_5, bs_3) \wedge \\
V_b(t_6, bs_4) &= V_b(t_6, bs_3)
\end{aligned}$$

$$\mathbf{bs_4\_bs_5\_ax} : transition(bs_4, bs_5) \Rightarrow \begin{aligned}[t]
V_b(u\_var, bs_5) &= V_b(u\_var, bs_4) \wedge \\
V_b(x\_var, bs_5) &= V_b(x\_var, bs_4) \wedge \\
V_b(y\_var, bs_5) &= V_b(y\_var, bs_4) \wedge \\
V_b(dx\_var, bs_5) &= V_b(dx\_var, bs_4) \wedge \\
V_b(t_1, bs_5) &= V_b(t_1, bs_4) \wedge \\
V_b(t_2, bs_5) &= V_b(t_2, bs_4) \wedge \\
V_b(t_3, bs_5) &= V_b(t_3, bs_4) \wedge \\
V_b(t_4, bs_5) &= mul(V_b(t_1, bs_4)\ ,\ V_b(t_2, bs_4)) \wedge \\
V_b(t_5, bs_5) &= V_b(t_5, bs_4) \wedge
\end{aligned}$$

$$V_b(t_6, bs_5) \ = \ V_b(t_6, bs_4)$$

$$\begin{aligned}
\mathbf{bs_5\_bs_6\_ax} : transition(bs_5, bs_6) \ \Rightarrow \ & V_b(u\_var, bs_6) \ = \ V_b(u\_var, bs_5) \wedge \\
& V_b(x\_var, bs_6) \ = \ V_b(x\_var, bs_5) \wedge \\
& V_b(y\_var, bs_6) \ = \ V_b(y\_var, bs_5) \wedge \\
& V_b(dx\_var, bs_6) \ = \ V_b(dx\_var, bs_5) \wedge \\
& V_b(t_1, bs_6) \ = \ V_b(t_1, bs_5) \wedge \\
& V_b(t_2, bs_6) \ = \ V_b(t_2, bs_5) \wedge \\
& V_b(t_3, bs_6) \ = \ V_b(t_3, bs_5) \wedge \\
& V_b(t_4, bs_6) \ = \ V_b(t_4, bs_5) \wedge \\
& V_b(t_5, bs_6) \ = \ mul(V_b(dx\_var, bs_5)\ , \ V_b(t_3, bs_5)) \wedge \\
& V_b(t_6, bs_6) \ = \ V_b(t_6, bs_5)
\end{aligned}$$

$$\begin{aligned}
\mathbf{bs_6\_bs_7\_ax} : transition(bs_6, bs_7) \ \Rightarrow \ & V_b(u\_var, bs_7) \ = \ V_b(u\_var, bs_6) \wedge \\
& V_b(x\_var, bs_7) \ = \ V_b(x\_var, bs_6) \wedge \\
& V_b(y\_var, bs_7) \ = \ V_b(y\_var, bs_6) \wedge \\
& V_b(dx\_var, bs_7) \ = \ V_b(dx\_var, bs_6) \wedge \\
& V_b(t_1, bs_7) \ = \ V_b(t_1, bs_6) \wedge \\
& V_b(t_2, bs_7) \ = \ V_b(t_2, bs_6) \wedge \\
& V_b(t_3, bs_7) \ = \ V_b(t_3, bs_6) \wedge \\
& V_b(t_4, bs_7) \ = \ V_b(t_4, bs_6) \wedge \\
& V_b(t_5, bs_7) \ = \ V_b(t_5, bs_6) \wedge \\
& V_b(t_6, bs_7) \ = \ sub(V_b(u\_var, bs_6)\ , \ V_b(t_4, bs_6))
\end{aligned}$$

$$\begin{aligned}
\mathbf{bs_7\_bs_8\_ax} : transition(bs_7, bs_8) \ \Rightarrow \ & V_b(u\_var, bs_8) \ = \ sub(V_b(t_6, bs_7)\ , \ V_b(t_5, bs_7)) \wedge \\
& V_b(x\_var, bs_8) \ = \ V_b(x\_var, bs_7) \wedge \\
& V_b(y\_var, bs_8) \ = \ V_b(y\_var, bs_7) \wedge \\
& V_b(dx\_var, bs_8) \ = \ V_b(dx\_var, bs_7) \wedge \\
& V_b(t_1, bs_8) \ = \ V_b(t_1, bs_7) \wedge \\
& V_b(t_2, bs_8) \ = \ V_b(t_2, bs_7) \wedge \\
& V_b(t_3, bs_8) \ = \ V_b(t_3, bs_7) \wedge \\
& V_b(t_4, bs_8) \ = \ V_b(t_4, bs_7) \wedge \\
& V_b(t_5, bs_8) \ = \ V_b(t_5, bs_7) \wedge \\
& V_b(t_6, bs_8) \ = \ V_b(t_6, bs_7)
\end{aligned}$$

Figure 9.3: **Formal description of the partial specification of Figure 9.2**

9.1.1 Synthesis with Unconstrained Non-pipelined Multi-Cycle Resources

Figure 9.4 shows a possible schedule for synthesis of the partial behavior specification of Figure 9.2 under no resource constraints. Under this scheme, a latency of 6 cycles is achieved using 3 multi-cycle multipliers (each with a delay of 2 clock cycles) and 1 subtractor. None of the resources in this implementation are pipelined. The data-path and controller of the RTL design, synthesized from the schedule of Figure 9.4, are given in Figures 9.5 and 9.6, respectively.

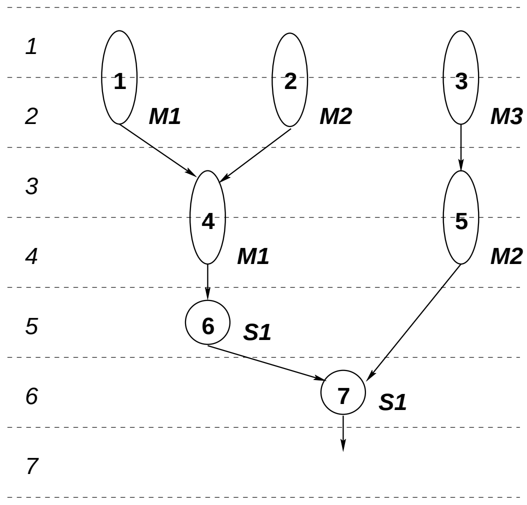


Figure 9.4: **A possible unconstrained schedule for the partial specification of Figure 9.2**

At each state of the controller, a set of control signals that activate some register transfer operations in the data path are asserted high. In Figure 9.6 the control inputs of some data-path components are listed close to each state. These are all the control inputs that are activated by control signals at that particular state.

In the data-path of the RTL design three different types of components are used: 2×1 multiplexers, subtractors and 2 cycle delay multipliers. The axiomatic descriptions of these components are given in Figure 9.7.

Following our verification algorithm, correctness of this example synthesized design can be validated if we can show that given the set of variables, the set of registers, and B_r, the mapping from the variables to registers, if at the initial states of the behavior and controller finite state machines, and under the register binding B_r, the set of critical variables and registers are equivalent, and transitions from the initial states to final states occur, then at the final states of the two finite state machines, and under the same register binding, the set of critical variables and registers are

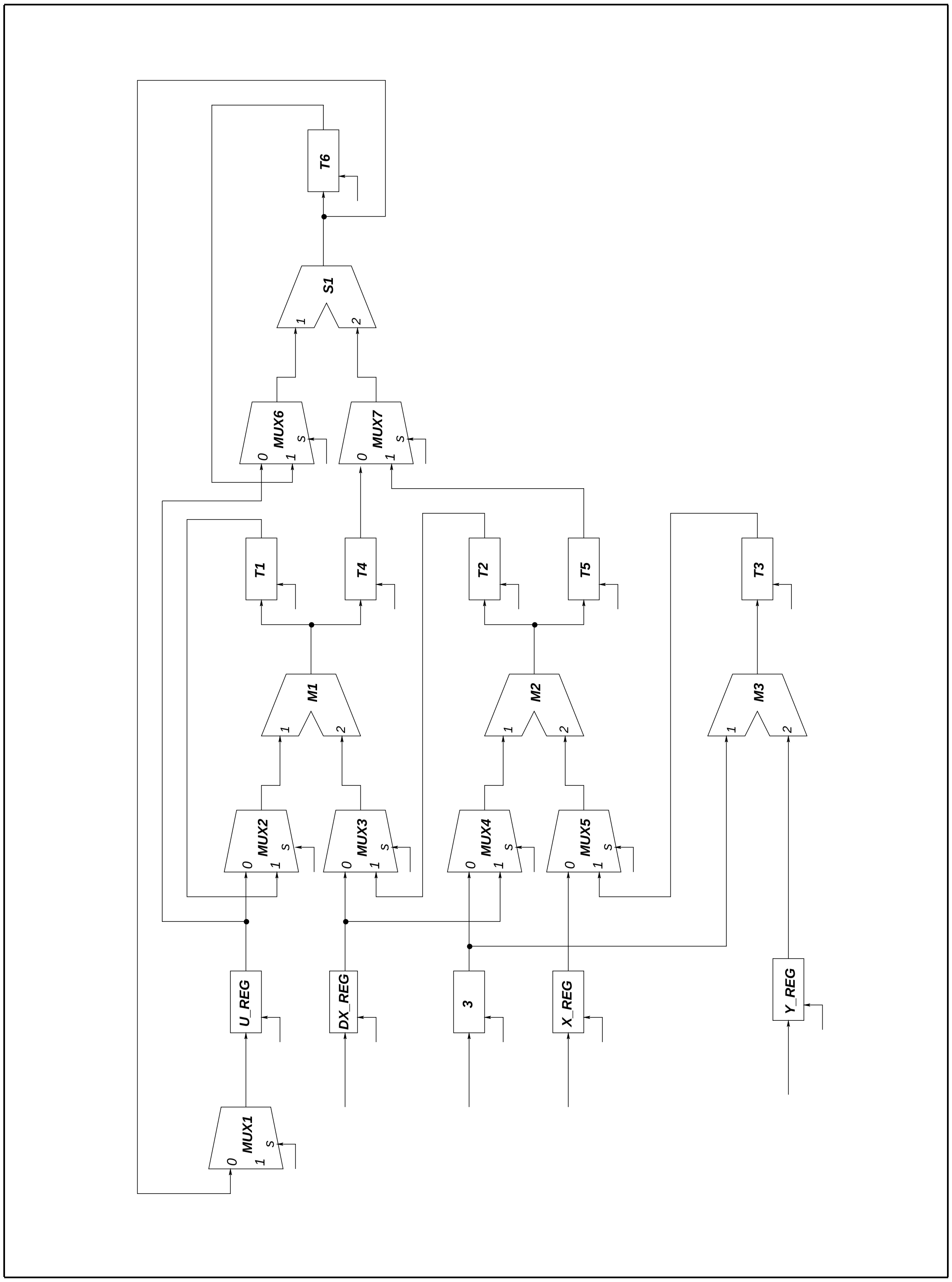


Figure 9.5: **Data path of the design synthesized from the schedule of Figure 9.4**

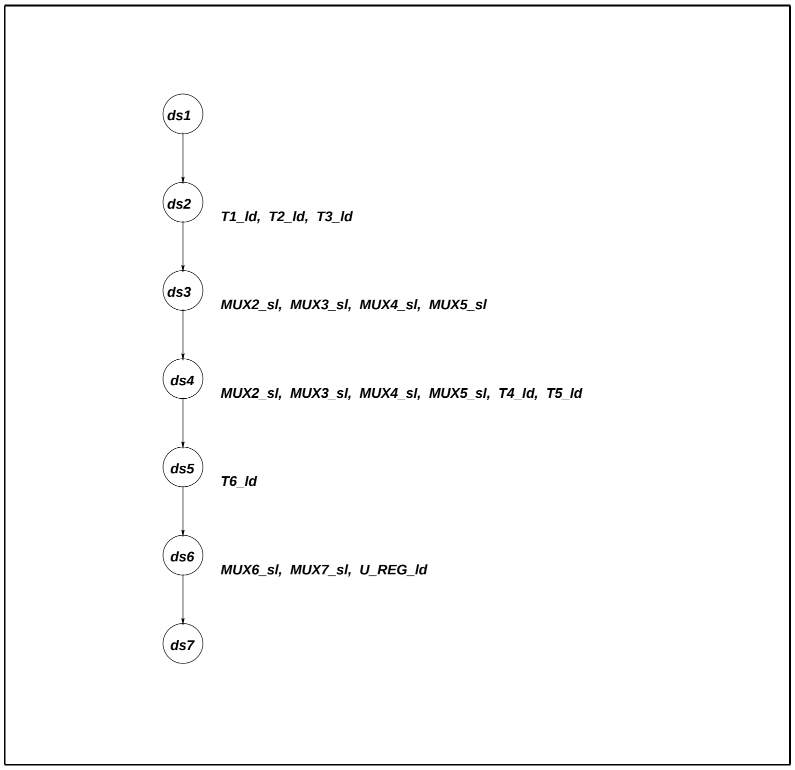

ds1
ds2
T1_ld, T2_ld, T3_ld
ds3
MUX2_sl, MUX3_sl, MUX4_sl, MUX5_sl
ds4
MUX2_sl, MUX3_sl, MUX4_sl, MUX5_sl, T4_ld, T5_ld
ds5
T6_ld
ds6
MUX6_sl, MUX7_sl, U_REG_ld
ds7

Figure 9.6: **Controller of the design synthesized from the schedule of Figure 9.4**

$$\textbf{2D\_Delay\_Multiplier\_ax}: \ \forall \ (\begin{array}{ll} in_1: & signal, \\ in_2: & signal, \\ out: & signal, \\ ds_1: & RTL\_state, \\ ds_2: & RTL\_state): \end{array}$$

$$\begin{array}{l} (2D\_delay\_muliplier(in_1, in_2, out) \ \wedge \\ \qquad trans(ds_1, ds_2)) \ \Rightarrow \\ \qquad\qquad (V_d(in_1, ds_2) = V_d(in_1, ds_1) \ \wedge \\ \qquad\qquad\qquad V_d(in_2, ds_2) = V_d(in_2, ds_1)) \ \Rightarrow \\ \qquad\qquad\qquad\qquad V_d(out, ds_2) = mul(V_d(in_1, ds_1), V_d(in_2, ds_1)) \end{array}$$

$$\textbf{2\_to\_1\_Multiplexer\_ax}: \ \forall \ (\begin{array}{ll} in_0: & signal, \\ in_1: & signal, \\ out: & signal, \\ sl: & bool\_signal, \\ ds: & RTL\_state): \end{array}$$

$$\begin{array}{l} multiplexer\_2\_1(in_0, in_1, sl, out) \ \Rightarrow \ (\neg V_d(sl, ds) \ \Rightarrow \ V_d(out, ds) = V_d(in_0, ds) \ \wedge \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad V_d(sl, ds) \ \Rightarrow \ V_d(out, ds) = V_d(in_1, ds)) \end{array}$$

$$\textbf{Register\_ax}: \ \forall \ (\begin{array}{ll} in: & signal, \\ out: & signal, \\ ld: & bool\_signal, \\ ds_1: & RTL\_state, \\ ds_2: & RTL\_state): \end{array}$$

$$\begin{array}{l} (register(in, ld, out) \ \wedge \ transition(ds_1, ds_2)) \ \Rightarrow \\ \qquad\qquad\qquad\qquad (V_d(ld, ds_1) \ \Rightarrow \ V_d(out, ds_2) = V_d(in, ds_1) \ \wedge \\ \qquad\qquad\qquad\qquad \neg V_d(ld, ds_1) \ \Rightarrow \ V_d(out, ds_2) = V_d(out, ds_1)) \end{array}$$

$$\begin{array}{llll}
\textbf{Subtractor\_ax}: & \forall\ (in_1: & signal, \\
 & \quad in_2: & signal, \\
 & \quad out: & signal, \\
 & \quad ds: & RTL\_state): \\
\end{array}$$

$$(subtractor(in_1, in_2, out) \Rightarrow V_d(out, ds) = sub(V_d(in_1, ds), V_d(in_2, ds))$$

Figure 9.7: **Description of behavior of the components in the data-path of Figure 9.5**

equivalent (It is understood without further explanation that in this discussion a mapping by B_s between the initial and final states and a mapping by B_p between the single critical paths of the partial behavioral specification and the controller is assumed):

Theorem 9.1

$$\begin{array}{l}
ds_1 = B_s(bs_1)\ \wedge\ ds_7 = B_s(bs_8)\ \wedge \\
\quad inter\_block\_transition(bs_1, bs_8)\ \wedge \\
\quad\quad inter\_block\_transition(ds_1, ds_8)\ \wedge \\
\quad\quad\quad (V_b(CR_b, bs_1)\ \overset{B_r}{\equiv}\ V_d(CR_d, ds_1)))\ \Rightarrow\ (V_b(CR_b, bs_8)\ \overset{B_r}{\equiv}\ V_d(CR_d, ds_7))
\end{array}$$

Before we continue with our discussion, we find it necessary to provide an explanation for the notation used in Theorem 9.1.

1. The function V_b (or V_d) is overloaded in the above theorem to denote a vector version of itself. Therefore, $V_b(CR_b, bs)\ \overset{B_r}{\equiv}\ V_d(CR_d, ds)$ should be interpreted as:

$$\forall r_b \in CR_b,\ \forall r_d \in CR_d\ :\ r_d = B_r(r_b)\ \Rightarrow\ V_b(r_b, b_s)\ \equiv\ V_d(r_d, d_s)$$

2. $inter\_block\_transition(s_i, s_f)$ denotes the condition flag of the initial transition of the critical path starting at the state s_i and ending at state s_f (a critical path corresponds to the sequence of transitions through a basic block, hence the naming inter_block_transition). As a critical path consists of a sequence of state transitions, if its condition flag is true, then the condition flag of all the state transitions composing the critical path will be true:

$$\textbf{ibt\_b\_ax}: \quad inter\_block\_transition(bs_1, bs_8) \Rightarrow \begin{array}[t]{l}(transition(bs_1, bs_2) \wedge \\ transition(bs_2, bs_3) \wedge \\ transition(bs_3, bs_4) \wedge \\ transition(bs_4, bs_5) \wedge \\ transition(bs_5, bs_6) \wedge \\ transition(bs_6, bs_7) \wedge \\ transition(bs_7, bs_8))\end{array}$$

$$\textbf{ibt\_d\_ax}: \quad inter\_block\_transition(ds_1, ds_8) \Rightarrow \begin{array}[t]{l}(transition(ds_1, ds_2) \wedge \\ transition(ds_2, ds_3) \wedge \\ transition(ds_3, ds_4) \wedge \\ transition(ds_4, ds_5) \wedge \\ transition(ds_5, ds_6) \wedge \\ transition(ds_6, ds_7))\end{array}$$

For this example the following is true:

$$CR_b = \{u\_var, x\_var, y\_var, dx\_var, t_1, t_2, t_3, t_4, t_5, t_6\} \tag{9.1}$$

$$CR_b = \{U\_REG, X\_REG, Y\_REG, DX\_REG, T_1, T_2, T_3, T_4, T_5, T_6\} \tag{9.2}$$

$$\begin{aligned} B_r = \{&u\_var \mapsto U\_REG,\ x\_var \mapsto X\_REG,\ y\_var \mapsto Y\_REG,\ dx\_var \mapsto DX\_REG, \\ &t_1 \mapsto T1,\ t_2 \mapsto T2,\ t_3 \mapsto T3,\ t_4 \mapsto T4,\ t_5 \mapsto T5,\ t_6 \mapsto T6\} \end{aligned} \tag{9.3}$$

Given the register mapping B_r Equation 9.3, the expressions $(V_b(CR_b, bs_1) \overset{B_r}{\equiv} V_d(CR_d, ds_1))$ and $(V_b(CR_b, bs_8) \overset{B_r}{\equiv} V_d(CR_d, ds_7))$ in Theorem 9.1 may be expanded:

$$(V_b(CR_b, bs_1) \overset{B_r}{\equiv} V_d(CR_d, ds_1)) \Leftrightarrow \begin{array}[t]{l}(V_b(u\_var, bs_1) = V_d(U\_REG, ds_1) \wedge \\ V_b(x\_var, bs_1) = V_d(X\_REG, ds_1) \wedge \\ V_b(y\_var, bs_1) = V_d(Y\_REG, ds_1) \wedge \\ V_b(dx\_var, bs_1) = V_d(DX\_REG, ds_1) \wedge \\ V_b(t_1, bs_1) = V_d(T_1, ds_1) \wedge \\ V_b(t_2, bs_1) = V_d(T_2, ds_1) \wedge \\ V_b(t_3, bs_1) = V_d(T_3, ds_1) \wedge \\ V_b(t_4, bs_1) = V_d(T_4, ds_1) \wedge \\ V_b(t_5, bs_1) = V_d(T_5, ds_1) \wedge \\ V_b(t_6, bs_1) = V_d(T_6, ds_1))\end{array}$$

$$\begin{aligned}(V_b(CR_b, bs_8) \overset{B_r}{\equiv} V_d(CR_d, ds_7)) \quad \Leftrightarrow \quad & (V_b(u\_var, bs_8) = V_d(U\_REG, ds_7) \ \wedge \\ & V_b(x\_var, bs_8) = V_d(X\_REG, ds_7) \ \wedge \\ & V_b(y\_var, bs_8) = V_d(Y\_REG, ds_7) \ \wedge \\ & V_b(dx\_var, bs_8) = V_d(DX\_REG, ds_7) \ \wedge \\ & V_b(t_1, bs_8) = V_d(T_1, ds_7) \ \wedge \\ & V_b(t_2, bs_8) = V_d(T_2, ds_7) \ \wedge \\ & V_b(t_3, bs_8) = V_d(T_3, ds_7) \ \wedge \\ & V_b(t_4, bs_8) = V_d(T_4, ds_7) \ \wedge \\ & V_b(t_5, bs_8) = V_d(T_5, ds_7) \ \wedge \\ & V_b(t_6, bs_8) = V_d(T_6, ds_7))\end{aligned}$$

Therefore, Theorem 9.1 may be considered as a conjunction of 4 lemmas, that may be independently verified. Below we show the proof of one of these lemmas and in the next section we will make a comparison between the proof of this lemma and a similar lemma, when pipelined resources are used in synthesis of this design.

Assumption 9.1

$$\begin{aligned}&((\langle bs_1, bs_8\rangle \overset{B_s}{\mapsto} \langle ds_1, ds_8\rangle) \ \wedge \\ &\quad inter\_block\_transition(bs_1, bs_8) \ \wedge \\ &\qquad inter\_block\_transition(ds_1, ds_8) \ \wedge \\ &\qquad\quad (V_b(CR_b, bs_1) \overset{B_r}{\equiv} V_d(CR_d, ds_1)))\end{aligned}$$

Lemma 9.1

$$(V_b(u\_var, bs_8) = V_d(U\_REG, ds_7)$$

Proof 9.1

(1) $transition(bs_1, bs_2)$
(2) $transition(bs_2, bs_3)$
(3) $transition(bs_3, bs_4)$
(4) $transition(bs_4, bs_5)$
(5) $transition(bs_5, bs_6)$
(6) $transition(bs_6, bs_7)$
(7) $transition(bs_7, bs_8)$

(8) $V_b(u\_var, bs_6) = V_b(u\_var, bs_5)$
(9) $V_b(u\_var, bs_5) = V_b(u\_var, bs_4)$
(10) $V_b(u\_var, bs_4) = V_b(u\_var, bs_3)$
(11) $V_b(u\_var, bs_3) = V_b(u\_var, bs_2)$
(12) $V_b(u\_var, bs_2) = V_b(u\_var, bs_1)$
(13) $V_b(u\_var, bs_6) = V_b(u\_var, bs_1)$ *(8)-(12)*

(14) $V_b(t_1, bs_4) = V_b(t_1, bs_3)$
(15) $V_b(t_1, bs_3) = V_b(t_1, bs_2)$
(16) $V_b(t_1, bs_2) = mul(V_b(u\_var, bs_1), V_b(dx_var, bs_1))$
(17) $V_b(t_1, bs_4) = mul(V_b(u\_var, bs_1), V_b(dx_var, bs_1))$ *(14)-(15)*

(18) $V_b(x\_var, bs_2) = V_b(x\_var, bs_1)$

(19) $V_b(t_2, bs_4) = V_b(t_2, bs_3)$
(20) $V_b(t_2, bs_3) = mul(const(3), V_b(x\_var, bs_2))$
(21) $V_b(t_2, bs_3) = mul(const(3), V_b(x\_var, bs_1))$ (18)
(22) $V_b(t_2, bs_4) = mul(const(3), V_b(x\_var, bs_1))$ (19)-(21)

(23) $V_b(t_4, bs_6) = V_b(t_4, bs_5)$
(24) $V_b(t_4, bs_5) = mul(V_b(t_1, bs_4), V_b(t_2, bs_4)$
(25) $V_b(t_4, bs_5) = mul(mul(V_b(u\_var, bs_1), V_b(dx_var, bs_1)),$
$mul(const(3), V_b(x\_var, bs_1)))$ (17),(22)

(26) $V_b(t_4, bs_6) = mul(mul(V_b(u\_var, bs_1), V_b(dx_var, bs_1)),$
$\qquad mul(const(3), V_b(x\_var, bs_1)))$ *(23)-(25)*

(27) $V_b(t_6, bs_7) = sub(V_b(u\_var, bs_6), V_b(t_4, bs_6))$
(28) $V_b(t_6, bs_7) = sub(V_b(u\_var, bs_1),$
$\qquad mul(mul(V_b(u\_var, bs_1), V_b(dx_var, bs_1)),$
$\qquad mul(const(3), V_b(x\_var, bs_1))))$ *(13),(26)*

(29) $V_b(dx\_var, bs_5) = V_b(dx\_var, bs_4)$
(30) $V_b(dx\_var, bs_4) = V_b(dx\_var, bs_3)$
(31) $V_b(dx\_var, bs_3) = V_b(dx\_var, bs_2)$
(32) $V_b(dx\_var, bs_2) = V_b(dx\_var, bs_1)$
(33) $V_b(dx\_var, bs_5) = V_b(dx\_var, bs_1)$ *(29)-(32)*

(34) $V_b(y\_var, bs_3) = V_b(y\_var, bs_2)$
(35) $V_b(y\_var, bs_2) = V_b(y\_var, bs_1)$
(36) $V_b(y\_var, bs_3) = V_b(y\_var, bs_1)$ *(34)-(35)*

(37) $V_b(t_3, bs_5) = V_b(t_3, bs_4)$
(38) $V_b(t_3, bs_4) = mul(const(3), V_b(y\_var, bs_3))$
(39) $V_b(t_3, bs_5) = mul(const(3), V_b(y\_var, bs_3))$ *(37)-(38)*
(40) $V_b(t_3, bs_5) = mul(const(3), V_b(y\_var, bs_1))$ *(36)*

(41) $V_b(t_5, bs_7) = V_b(t_5, bs_6)$
(42) $V_b(t_5, bs_6) = mul(V_b(dx\_var, bs_5), V_b(t_3, bs_5))$
(43) $V_b(t_5, bs_7) = mul(V_b(dx\_var, bs_5), V_b(t_3, bs_5))$ *(41)-(42)*
(44) $V_b(t_5, bs_7) = mul(V_b(dx\_var, bs_1),$
$\qquad mul(const(3), V_b(y\_var, bs_1)))$ *(33),(40)*

(45) $V_b(u\_var, bs_8) = sub(V_b(t_6, bs_7), V_b(t_5, bs_7))$
(46) $V_b(u\_var, bs_8) = sub(sub(V_b(u\_var, bs_1),$
$\qquad mul(mul(V_b(u\_var, bs_1),$
$\qquad V_b(dx\_var, bs_1)),$
$\qquad mul(const(3),$
$\qquad V_b(x\_var, bs_1)))),$
$\qquad mul(V_b(dx\_var, bs_1),$
$\qquad mul(const(3),$
$\qquad V_b(y\_var, bs_1))))$ *(28),(44)*

(47) $transition(ds_1, ds_2)$
(48) $transition(ds_2, ds_3)$
(49) $transition(ds_3, ds_4)$
(50) $transition(ds_4, ds_5)$
(51) $transition(ds_5, ds_6)$
(52) $transition(ds_6, ds_7)$

(53) $V_d(U\_REG.out, ds_5) = V_d(U\_REG.out, ds_4)$
(54) $V_d(U\_REG.out, ds_4) = V_d(U\_REG.out, ds_3)$
(55) $V_d(U\_REG.out, ds_3) = V_d(U\_REG.out, ds_2)$
(56) $V_d(U\_REG.out, ds_2) = V_d(U\_REG.out, ds_1)$
(57) $V_d(U\_REG.out, ds_5) = V_d(U\_REG.out, ds_1)$ *(53)-(56)*

(58) $V_d(S_1.in1, ds_5) = V_d(MUX_6.out, ds_5)$
(59) $V_d(MUX_6.out, ds_5) = V_d(MUX_6.in0, ds_5)$
(60) $V_d(MUX_6.in0, ds_5) = V_d(U\_REG.out, ds_5)$
(61) $V_d(U\_REG.out, ds_5) = V_d(U\_REG.out, ds_1)$ *(57)*
(62) $V_d(S_1.in1, ds_5) = V_d(U\_REG.out, ds_1)$ *(58)-(61)*

(63) $V_d(M_1.in1, ds_1) = V_d(MUX_2.out, ds_1)$
(64) $V_d(MUX_2.out, ds_1) = V_d(MUX_2.in0, ds_1)$
(65) $V_d(MUX_2.in0, ds_1) = V_d(U\_REG.out, ds_1)$
(66) $V_d(M_1.in1, ds_1) = V_d(U\_REG.out, ds_1)$ (63)-(65)

(67) * $V_d(M_1.in1, ds_2) = V_d(MUX_2.out, ds_2)$
(68) * $V_d(MUX_2.out, ds_2) = V_d(MUX_2.in0, ds_2)$
(69) * $V_d(MUX_2.in0, ds_2) = V_d(U\_REG.out, ds_2)$
(70) * $V_d(U\_REG.out, ds_2) = V_d(U\_REG.out, ds_1)$
(71) * $V_d(U\_REG.out, ds_1) = V_d(M_1.in1, ds_1)$ (66)
(72) * $V_d(M_1.in1, ds_2) = V_d(M_1.in1, ds_1)$ (67)-(71)

(73) $V_d(M_1.in2, ds_1) = V_d(MUX_3.out, ds_1)$
(74) $V_d(MUX_3.out, ds_1) = V_d(MUX_3.in0, ds_1)$
(75) $V_d(MUX_3.in0, ds_1) = V_d(DX\_REG.out, ds_1)$
(76) $V_d(M_1.in2, ds_1) = V_d(DX\_REG.out, ds_1)$ (73)-(75)

(77) * $V_d(M_1.in2, ds_2) = V_d(MUX_3.out, ds_2)$
(78) * $V_d(MUX_3.out, ds_2) = V_d(MUX_3.in0, ds_2)$
(79) * $V_d(MUX_3.in0, ds_2) = V_d(DX\_REG.out, ds_2)$
(80) * $V_d(DX\_REG.out, ds_2) = V_d(DX\_REG.out, ds_1)$
(81) * $V_d(DX\_REG.out, ds_1) = V_d(M_1.in2, ds_1)$ (76)
(82) * $V_d(M_1.in2, ds_2) = V_d(M_1.in2, ds_1)$ (77)-(81)

(83) $V_d(T_1.out, ds_3) = V_d(T_1.in, ds_2)$
(84) $V_d(T_1.in, ds_2) = V_d(M_1.out, ds_2)$
(85) $V_d(M_1.out, ds_2) = mul(V_d(M_1.in1, ds_1),\ V_d(M_1.in2, ds_1))$ (72),(82)
(86) $V_d(M_1.out, ds_2) = mul(V_d(U\_REG.out, ds_1),\ V_d(DX\_REG.out, ds_1))$ (66),(76)
(87) $V_d(T_1.out, ds_3) = mul(V_d(U\_REG.out, ds_1),\ V_d(DX\_REG.out, ds_1))$ (83)-(86)

(88) $V_d(M_1.in1, ds_3) = V_d(MUX_2.out, ds_3)$
(89) $V_d(MUX_2.out, ds_3) = V_d(MUX_2.in1, ds_3)$
(90) $V_d(MUX_2.in1, ds_3) = V_d(T_1.out, ds_3)$
(91) $V_d(M_1.in1, ds_3) = V_d(T_1.out, ds_3)$ (88)-(90)
(92) $V_d(T_1.out, ds_3) = mul(V_d(U\_REG.out, ds_1), V_d(DX\_REG.out, ds_1))$ (87)
(93) $V_d(M_1.in1, ds_3) = mul(V_d(U\_REG.out, ds_1), V_d(DX\_REG.out, ds_1))$ (91)-(92)

(94) * $V_d(M_1.in1, ds_4) = V_d(MUX_2.out, ds_4)$
(95) * $V_d(MUX_2.out, ds_4) = V_d(MUX_2.in1, ds_4)$
(96) * $V_d(MUX_2.in1, ds_4) = V_d(T_1.out, ds_4)$
(97) * $V_d(T_1.out, ds_4) = V_d(T_1.out, ds_3)$
(98) * $V_d(T_1.out, ds_3) = V_d(M_1.in1, ds_3)$ (91)
(99) * $V_d(M_1.in1, ds_4) = V_d(M_1.in1, ds_3)$ (94)-(98)

(100) $V_d(M_2.in1, ds_1) = V_d(MUX_4.out, ds_1)$
(101) $V_d(MUX_4.out, ds_1) = V_d(MUX_4.in0, ds_1)$
(102) $V_d(MUX_4.in0, ds_1) = const(3)$
(103) $V_d(M_2.in1, ds_1) = const(3)$ (100)-(102)

(104) * $V_d(M_2.in1, ds_2) = V_d(MUX_4.out, ds_2)$
(105) * $V_d(MUX_4.out, ds_2) = V_d(MUX_4.in0, ds_2)$
(106) * $V_d(MUX_4.in0, ds_2) = const(3)$
(107) * $V_d(M_2.in1, ds_2) = const(3)$ (104)-(106)
(108) * $V_d(M_2.in1, ds_1) = const(3)$ (103)
(109) * $V_d(M_2.in1, ds_2) = V_d(M_2.in1, ds_1)$ (107),(108)

(110)		$V_d(M_2.in2, ds_1) = V_d(MUX_5.out, ds_1)$	
(111)		$V_d(MUX_5.out, ds_1) = V_d(MUX_5.in0, ds_1)$	
(112)		$V_d(MUX_5.in0, ds_1) = V_d(X\_REG.out, ds_1)$	
(113)		$V_d(M_2.in2, ds_1) = V_d(X\_REG.out, ds_1)$	(110)-(112)
(114)	*	$V_d(M_2.in2, ds_2) = V_d(MUX_5.out, ds_2)$	
(115)	*	$V_d(MUX_5.out, ds_2) = V_d(MUX_5.in0, ds_2)$	
(116)	*	$V_d(MUX_5.in0, ds_2) = V_d(X\_REG.out, ds_2)$	
(117)	*	$V_d(X\_REG.out, ds_2) = V_d(X\_REG.out, ds_1)$	
(118)	*	$V_d(M_2.in2, ds_2) = V_d(X\_REG.out, ds_1)$	(114)-(117)
(119)	*	$V_d(M_2.in2, ds_1) = V_d(X\_REG.out, ds_1)$	(113)
(120)	*	$V_d(M_2.in2, ds_2) = V_d(M_2.in2, ds_1)$	(118),(119)
(121)		$V_d(T_2.out, ds_3) = V_d(T_2.in, ds_2)$	
(122)		$V_d(T_2.in, ds_2) = V_d(M_2.out, ds_2)$	
(123)		$V_d(M_2.out, ds_2) = mul(V_d(M_2.in1, ds_1), V_d(M_2.in2, ds_1)$	(109),(120)
(124)		$V_d(M_2.out, ds_2) = mul(const(3), V_d(X\_REG.out, ds_1))$	(103),(113)
(125)		$V_d(T_2.out, ds_3) = mul(const(3), V_d(X\_REG.out, ds_1))$	(121)-(124)
(126)		$V_d(M_1.in2, ds_3) = V_d(MUX_3.out, ds_3)$	
(127)		$V_d(MUX_3.out, ds_3) = V_d(MUX_3.in1, ds_3)$	
(128)		$V_d(MUX_3.in1, ds_3) = V_d(T_2.out, ds_3)$	
(129)		$V_d(M_1.in2, ds_3) = V_d(T_2.out, ds_3)$	(126)-(128)
(130)		$V_d(T_2.out, ds_3) = mul(const(3), V_d(X\_REG.out, ds_1))$	(125)
(131)		$V_d(M_1.in2, ds_3) = mul(const(3), V_d(X\_REG.out, ds_1))$	(129)-(130)
(132)	*	$V_d(M_1.in2, ds_4) = V_d(MUX_3.out, ds_4)$	
(133)	*	$V_d(MUX_3.out, ds_4) = V_d(MUX_3.in1, ds_4)$	
(134)	*	$V_d(MUX_3.in1, ds_4) = V_d(T_2.out, ds_4)$	
(135)	*	$V_d(T_2.out, ds_4) = V_d(T_2.out, ds_3)$	
(136)	*	$V_d(M_1.in2, ds_4) = V_d(T_2.out, ds_3)$	(132)-(135)
(137)	*	$V_d(M_1.in2, ds_3) = V_d(T_2.out, ds_3)$	(129)
(138)	*	$V_d(M_1.in2, ds_4) = V_d(M_1.in2, ds_3)$	(136),(137)
(139)		$V_d(T_4.out, ds_5) = V_d(T_4.in, ds_4)$	
(140)		$V_d(T_4.in, ds_4) = V_d(M_1.out, ds_4)$	
(141)		$V_d(M_1.out, ds_4) = mul(V_d(M_1.in1, ds_3), V_d(M_1.in2, ds_3))$	(99),(138)

(142) $V_d(M_1.out, ds_4) = mul(mul(V_d(U\_REG.out, ds_1), V_d(DX\_REG.out, ds_1)),$
$mul(const(3), V_d(X\_REG.out, ds_1))),$ (93),(131)

(143) $V_d(T_4.out, ds_5) = mul(mul(V_d(U\_REG.out, ds_1), V_d(DX\_REG.out, ds_1)),$
$mul(const(3), V_d(X\_REG.out, ds_1)))$ (139)-(142)

(144) $V_d(S_1.in2, ds_5) = V_d(MUX_7.out, ds_5)$
(145) $V_d(MUX_7.out, ds_5) = V_d(MUX_7.in0, ds_5)$
(146) $V_d(MUX_7.in0, ds_5) = V_d(T_4.out, ds_5)$
(147) $V_d(T_4.out, ds_5) = mul(mul(V_d(U\_REG.out, ds_1), V_d(DX\_REG.out, ds_1)),$
$mul(const(3), V_d(X\_REG.out, ds_1)))$ (143)
(148) $V_d(S_1.in2, ds_5) = mul(mul(V_d(U\_REG.out, ds_1), V_d(DX\_REG.out, ds_1)),$
$mul(const(3), V_d(X\_REG.out, ds_1)))$ (144)-(147)

(149) $V_d(T_6.out, ds_6) = V_d(T_6.in, ds_5)$
(150) $V_d(T_6.in, ds_5) = V_d(S_1.out, ds_5)$
(151) $V_d(S_1.out, ds_5) = sub(V_d(S_1.in1, ds_5), V_d(S_1.in2, ds_5))$
(152) $V_d(S_1.out, ds_5) = sub(V_d(U\_REG.out, ds_1),$
$mul(mul(V_d(U\_REG.out, ds_1),$
$V_d(DX\_REG.out, ds_1)),$
$mul(const(3),$
$V_d(X\_REG.out, ds_1))))$ (62),(148)

(153) $V_d(T_6.out, ds_6) = sub(V_d(U\_REG.out, ds_1),$
$mul(mul(V_d(U\_REG.out, ds_1),$
$V_d(DX\_REG.out, ds_1)),$
$mul(const(3),$
$V_d(X\_REG.out, ds_1))))$ (149)-(152)

(154) $V_d(S_1.in1, ds_6) = V_d(MUX_6.out, ds_6)$
(155) $V_d(MUX_6.out, ds_6) = V_d(MUX_6.in1, ds_6)$
(156) $V_d(MUX_6.in1, ds_6) = V_d(T_6.out, ds_6)$
(157) $V_d(T_6.out, ds_6) = sub(V_d(U\_REG.out, ds_1),$
$mul(mul(V_d(U\_REG.out, ds_1),$
$V_d(DX\_REG.out, ds_1)),$
$mul(const(3),$
$V_d(X\_REG.out, ds_1))))$ (153)

(158) $V_d(S_1.in1, ds_6) = sub(V_d(U\_REG.out, ds_1),$
$mul(mul(V_d(U\_REG.out, ds_1),$
$V_d(DX\_REG.out, ds_1)),$
$mul(const(3),$
$V_d(X\_REG.out, ds_1))))$ (154)-(157)

(159) $V_d(DX\_REG.out, ds_3) = V_d(DX\_REG.out, ds_2)$
(160) $V_d(DX\_REG.out, ds_2) = V_d(DX\_REG.out, ds_1)$
(161) $V_d(DX\_REG.out, ds_3) = V_d(DX\_REG.out, ds_1)$ (159)-(160)

(162) $V_d(M_2.in1, ds_3) = V_d(MUX_4.out, ds_3)$
(163) $V_d(MUX_4.out, ds_3) = V_d(MUX_4.in1, ds_3)$
(164) $V_d(MUX_4.in1, ds_3) = V_d(DX\_REG.out, ds_3)$
(165) $V_d(DX\_REG.out, ds_3) = V_d(DX\_REG.out, ds_1)$ (161)
(166) $V_d(M_2.in1, ds_3) = V_d(DX\_REG.out, ds_1)$ (162)-(165)

(167) * $V_d(M_2.in1, ds_4) = V_d(MUX_4.out, ds_4)$
(168) * $V_d(MUX_4.out, ds_4) = V_d(MUX_4.in1, ds_4)$
(169) * $V_d(MUX_4.in1, ds_4) = V_d(DX\_REG.out, ds_4)$
(170) * $V_d(DX\_REG.out, ds_4) = V_d(DX\_REG.out, ds_1)$ (161)
(171) * $V_d(M_2.in1, ds_4) = V_d(DX\_REG.out, ds_1)$ (167)-(170)
(172) * $V_d(M_2.in1, ds_3) = V_d(DX\_REG.out, ds_1)$ (166)
(173) * $V_d(M_2.in1, ds_4) = V_d(M_2.in1, ds_3)$ (171),(172)

(174) $V_d(M_3.in1, ds_1) = const(3)$

(175) $V_d(M_3.in1, ds_2) = const(3)$
(176) $V_d(M_3.in1, ds_2) = V_d(M_3.in1, ds_1)$ (174),(175)

(177) $V_d(M_3.in2, ds_1) = V_d(Y\_REG.out, ds_1)$

(178) * $V_d(M_3.in2, ds_2) = V_d(Y\_REG.out, ds_2)$
(179) * $V_d(Y\_REG.out, ds_2) = V_d(Y\_REG.out, ds_1)$
(180) * $V_d(M_3.in2, ds_2) = V_d(Y\_REG.out, ds_1)$ (178)-(179)
(181) * $V_d(M_3.in2, ds_1) = V_d(Y\_REG.out, ds_1)$ (177)
(182) * $V_d(M_3.in2, ds_2) = V_d(M_3.in2, ds_1)$ (180),(181)

(183)	$V_d(T_3.out, ds_3) = V_d(T_3.in, ds_2)$	
(184)	$V_d(T_3.in, ds_2) = V_d(M_3.out, ds_2)$	
(185)	$V_d(M_3.out, ds_2) = mul(V_d(M_3.in1, ds_1), V_d(M_3.in2, ds_1))$	(176),(182)
(186)	$V_d(M_3.out, ds_2) = mul(const(3), V_d(Y\_REG.out, ds_1))$	(174),(177)
(187)	$V_d(T_3.out, ds_3) = mul(const(3), V_d(Y\_REG.out, ds_1))$	(183)-(186)
(188)	$V_d(M_2.in2, ds_3) = V_d(MUX_5.out, ds_3)$	
(189)	$V_d(MUX_5.out, ds_3) = V_d(MUX_5.in1, ds_3)$	
(190)	$V_d(MUX_5.in1, ds_3) = V_d(T_3.out, ds_3)$	
(191)	$V_d(T_3.out, ds_3) = mul(const(3), V_d(Y\_REG.out, ds_1))$	(187)
(192)	$V_d(M_2.in2, ds_3) = mul(const(3), V_d(Y\_REG.out, ds_1))$	(188)-(191)
(193)	* $V_d(M_2.in2, ds_4) = V_d(MUX_5.out, ds_4)$	
(194)	* $V_d(MUX_5.out, ds_4) = V_d(MUX_5.in1, ds_4)$	
(195)	* $V_d(MUX_5.in1, ds_4) = V_d(T_3.out, ds_4)$	
(196)	* $V_d(T_3.out, ds_4) = V_d(T_3.out, ds_3)$	
(197)	* $V_d(T_3.out, ds_3) = mul(const(3), V_d(Y\_REG.out, ds_1))$	
(198)	* $V_d(M_2.in2, ds_4) = mul(const(3), V_d(Y\_REG.out, ds_1))$	(193)-(197)
(199)	* $V_d(M_2.in2, ds_3) = mul(const(3), V_d(Y\_REG.out, ds_1))$	(192)
(200)	* $V_d(M_2.in2, ds_4) = V_d(M_2.in2, ds_3)$	(198),(199)
(201)	$V_d(T_5.out, ds_6) = V_d(T_5.out, ds_5)$	
(202)	$V_d(T_5.out, ds_5) = V_d(T_5.in, ds_4)$	
(203)	$V_d(T_5.in, ds_4) = V_d(M_2.out, ds_4)$	
(204)	$V_d(M_2.out, ds_4) = mul(V_d(M_2.in1, ds_3), V_d(M_2.in2, ds_3))$	(173),(200)
(205)	$V_d(M_2.out, ds_4) = mul(V_d(DX\_REG.out, ds_1), mul(const(3), V_d(Y\_REG.out, ds_1)))$	(166),(192)
(206)	$V_d(T_5.out, ds_6) = mul(V_d(DX\_REG.out, ds_1), mul(const(3), V_d(Y\_REG.out, ds_1)))$	(201)-(205)
(207)	$V_d(S_1.in2, ds_6) = V_d(MUX_7.out, ds_6)$	
(208)	$V_d(MUX_7.out, ds_6) = V_d(MUX_7.in1, ds_6)$	
(209)	$V_d(MUX_7.in1, ds_6) = V_d(T_5.out, ds_6)$	
(210)	$V_d(T_5.out, ds_6) = mul(V_d(DX\_REG.out, ds_1), mul(const(3), V_d(Y\_REG.out, ds_1)))$	(206)

(211) $V_d(S_1.in2, ds_6) = mul(V_d(DX\_REG.out, ds_1),$
$\quad mul(const(3),$
$\quad\quad V_d(Y\_REG.out, ds_1)))$ (207)-(210)

(212) $V_d(U\_REG.out, ds_7) = V_d(U\_REG.in, ds_6)$
(213) $V_d(U\_REG.in, ds_6) = V_d(MUX_1.out, ds_6)$
(214) $V_d(MUX_1.out, ds_6) = V_d(MUX_1.in0, ds_6)$
(215) $V_d(MUX_1.in0, ds_6) = V_d(S_1.out, ds_6)$
(216) $V_d(S_1.out, ds_6) = sub(V_d(S_1.in1, ds_6), V_d(S_1.in2, ds_6)))$
(217) $V_d(S_1.out, ds_6) = sub(sub(V_d(U\_REG.out, ds_1),$
$\quad mul(mul(V_d(U\_REG.out, ds_1),$
$\quad\quad V_d(DX\_REG.out, ds_1)),$
$\quad\quad mul(const(3),$
$\quad\quad\quad V_d(X\_REG.out, ds_1)))),$
$\quad mul(V_d(DX\_REG.out, ds_1)$
$\quad\quad mul(const(3),$ (158),(211)
$\quad\quad\quad V_d(Y\_REG.out, ds_1))))$ (122),(157)
(218) $V_d(U\_REG.in, ds_7) = sub(sub(V_d(U\_REG.out, ds_1),$
$\quad mul(mul(V_d(U\_REG.out, ds_1),$
$\quad\quad V_d(DX\_REG.out, ds_1)),$
$\quad\quad mul(const(3),$
$\quad\quad\quad V_d(X\_REG.out, ds_1)))),$
$\quad mul(V_d(DX\_REG.out, ds_1)$
$\quad\quad mul(const(3),$
$\quad\quad\quad V_d(Y\_REG.out, ds_1))))$ (212)-(217)

(219) $V_b(u\_var, bs_7) = V_d(U\_REG.out, ds_8)$ (46),(218)
(220) $Q.E.D.$

Discussion

We saw that for proving the Lemma 9.1 the value of the variable u_var at state bs_8 and the content of register U_REG at state ds_8 are symbolically calculated in terms of the values of the critical specification variables at state bs_1 and the contents of the critical design registers at state ds_1, respectively. During these symbolic calculations the RTL operations are back traced in temporal as well as spatial domain. This means that starting from the component U_REG and the state ds_8 the output of each component is recursively calculated in terms of its input(s)/output at that or a previous state. As part of this process, the output of the multi-cycle multiplier component M_1

of the Figure 9.5 at state ds_4 is calculated in terms of its inputs at the previous state, ds_3. By instantiating the axiom $2D\_Delay\_Multiplier\_ax$ in Figure 9.7, the behavior of the multi-cycle M_1 at states ds_3 and ds_4 is described by:

$$\begin{aligned}(V_d(M_1.in_1, ds_4) = V_d(M_1.in_1, ds_3)\ \wedge \\ V_d(M_1.in_2, ds_4) = V_d(M_1.in_2, ds_3))\ \Leftrightarrow \\ V_d(M_1.out, ds_4) = mul(V_d(M_1.in_1, ds_3), V_d(M_1.in_2, ds_3))\end{aligned}$$

We note that the relation:

$$V_d(M_1.out, ds_4) = mul(V_d(M_1.in_1, ds_3), V_d(M_1.in_2, ds_3))$$

holds between the value at the output of M_1 at the state ds_4 and the values of its inputs at state ds_3 (line 141 of the proof script), provided that the preconditions for the correct function of M_1 at states d_3 and ds_4 are true, i.e.:

$$\begin{aligned}(V_d(M_1.in_1, ds_4) = V_d(M_1.in_1, ds_3)\ \wedge \\ V_d(M_1.in_2, ds_4) = V_d(M_1.in_2, ds_3))\end{aligned}$$

This is translated into additional proof steps required for verifying these preconditions (lines $94-99$ and $132 - 138$ of the proof scripts). The additional proof steps required for verifying the preconditions are marked by $*$ in the proof script. It should be noted that if the purely combinational multiplier described by $multiplier\_axiom$ is used in synthesizing the design, the input-output relation of this component at state ds_4 would be simply defined by:

$$V_d(M_1.out, ds_4) = mul(V_d(M_1.in1, ds_4), V_d(M_1.in_2, ds_4))$$

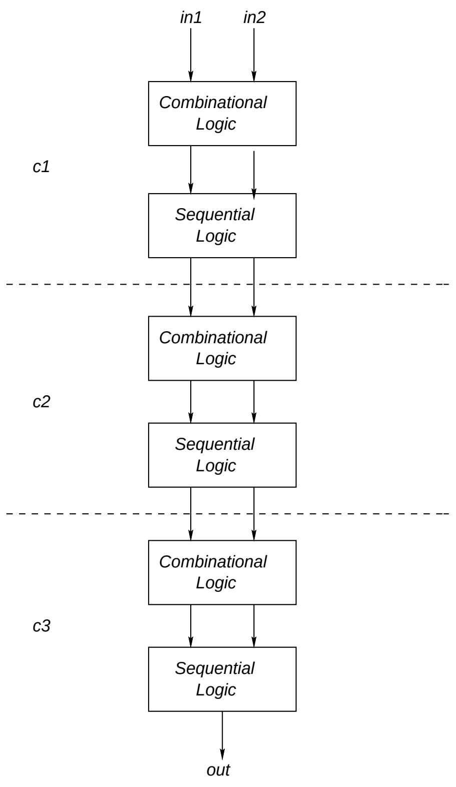


Figure 9.8: **The stages of a pipelined component**

9.2 Pipelined Resources

If a circuit is synthesized from a non-pipelined CDFG, but using *pipelined resources*, structural pipelining is said to be present in the circuit. A pipelined resource can consume data faster than it can process it. More formally, in a pipelined resource with the execution delay λ, throughput τ, and the data introduction interval $\delta_0 = 1/\tau$, $\lambda = k\delta_0$ where $k \in Z$ and $k > 1$. A non-pipelined resource is the special case where $k = 1$.

Pipelined resources like other multi-cycle resources perform an operation in a few stages. The inputs to the first stage are the main inputs to the component. The inputs to each following stage are the outputs from the previous stage. Once an stage completes its operation it passes the results to the following stage. Other than this exchange of inputs and outputs, there is no interrelationship between the function of any two stages (Figure 9.8). As a result different stages of a pipelined resource can operate on different sets of data. To be more precise, assuming that δ_0

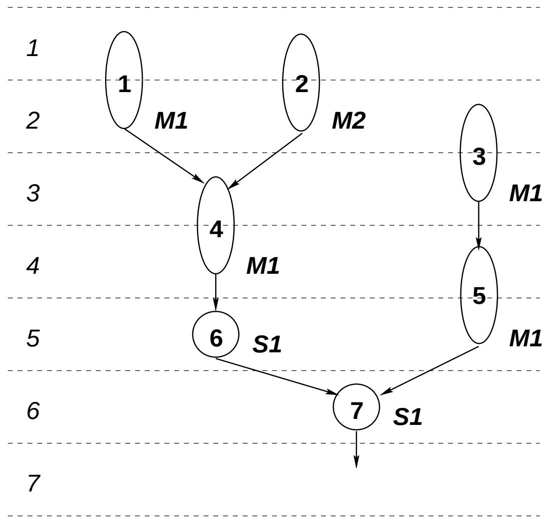


Figure 9.9: **A possible unconstrained schedule with pipelined resources for the partial specification of Figure 9.2**

is the maximum delay through a stage of a pipelined resource, it can consume a new set of data at every δ_0 cycle. In this case, at no point in time two different stages of the pipeline operate on the same set of data. This is a unique characteristic of the pipelined resources compared to the other multi-cycle resources.

A typical example of a pipelined resource is a parallel multiplier with an execution delay $\lambda =$ 2 cycles [34, 17]. In the first cycle (or stage) a Wallace-tree reduction of the partial products into two summands is done. In the second cycle (or stage) the final addition and rounding are performed. By overlapping the computations at the two cycles (or stages), data can be produced and consumed at each cycle, i.e. $\delta_0 = 1$ cycle and $\tau = 1/\delta_0 = 1$ per cycle.

Figure 9.9 shows a second possible schedule for the partial behavior specification of Figure 9.2. Like the previous schedule, this schedule has no resource constraints, but unlike that, it uses pipelined resources: the two multipliers M_1 and M_2. Due to the use of pipelined resources, this design can be implemented with one less multiplier than the previous one. Under this scheme also, a latency of 6 cycles is achieved.

Consider the Figure 9.9. Even though the latency of each pipelined multiplier in this design is 2 cycles, it can consume and produce data at the rate of one per cycle. In this scheme, four multiplication operations are scheduled and bound to a single multiplier, M_1. These multiplications, that each has a delay of 2 cycles, are initiated in 4 consecutive states. The result of the first multiplication operation is produced after 2 cycles (the latency of the multiplier) and one result is produced at each of the next 3 cycles. This is different from the multipliers used in the previous design, in that, no new data can be input to the multipliers before it has completely processed a previous set of data, or else incorrect results will be generated.

The controller of the design generates the necessary control signals at each state and activates the appropriate register transfer operations in the data-path. The control input signals listed close to each state in Figure 9.11 are the only signals that are asserted high at that state.

In the data-path of Figure 9.10 several different types of components have been used: 4×1 and 2×1 multiplexers, subtractors, 2 cycle delay pipelined multipliers and registers. The axiomatic description of these components are given in Figure 9.12.

Since structural pipelining does not generate new operational dependencies between the operations within a basic block of synthesis and operations out of that block, our verification algorithm may be safely used for verification of the designs with pipelined resources. As before, a part of the proof is constructed by following the register transfer operations of the design. In this process, the formal specifications of Figure 9.2 are instantiated to describe the input-output behavior of various components at different states.

Now, we will present the proof of the Lemma 9.1, for this second implementation of the behavior description of Figure 9.2, where the design is synthesized from pipelined resources. The assumptions in this case are as the previous case. Since both implementations are synthesized from identical behavioral specifications, the first steps of the proofs of correctness of the Lemma 9.1, where the behavior operations are analyzed, are the same as the previous proof (steps *(1)-(46)*). As always a mapping by B_s between the initial and final states and a mapping by B_p between the single critical paths of the partial behavioral specification and the controller is assumed.

The two inter-block transitions $\langle bs_1, bs_8 \rangle$ and $\langle ds_1, ds_7 \rangle$ along the critical paths cp_b and cp_d consist of a sequence of single state transitions:

$$\begin{array}{llcl} \textbf{ibt\_b\_ax}: & inter\_block\_transition(bs_1, bs_8) & \Rightarrow & (transition(bs_1, bs_2) \wedge \\ & & & \;transition(bs_2, bs_3) \wedge \\ & & & \;transition(bs_3, bs_4) \wedge \\ & & & \;transition(bs_4, bs_5) \wedge \\ & & & \;transition(bs_5, bs_6) \wedge \\ & & & \;transition(bs_6, bs_7) \wedge \\ & & & \;transition(bs_7, bs_8)) \end{array}$$

$$\begin{array}{llcl} \textbf{ibt\_d\_ax}: & inter\_block\_transition(ds_1, ds_7) & \Rightarrow & (transition(ds_1, ds_2) \wedge \end{array}$$

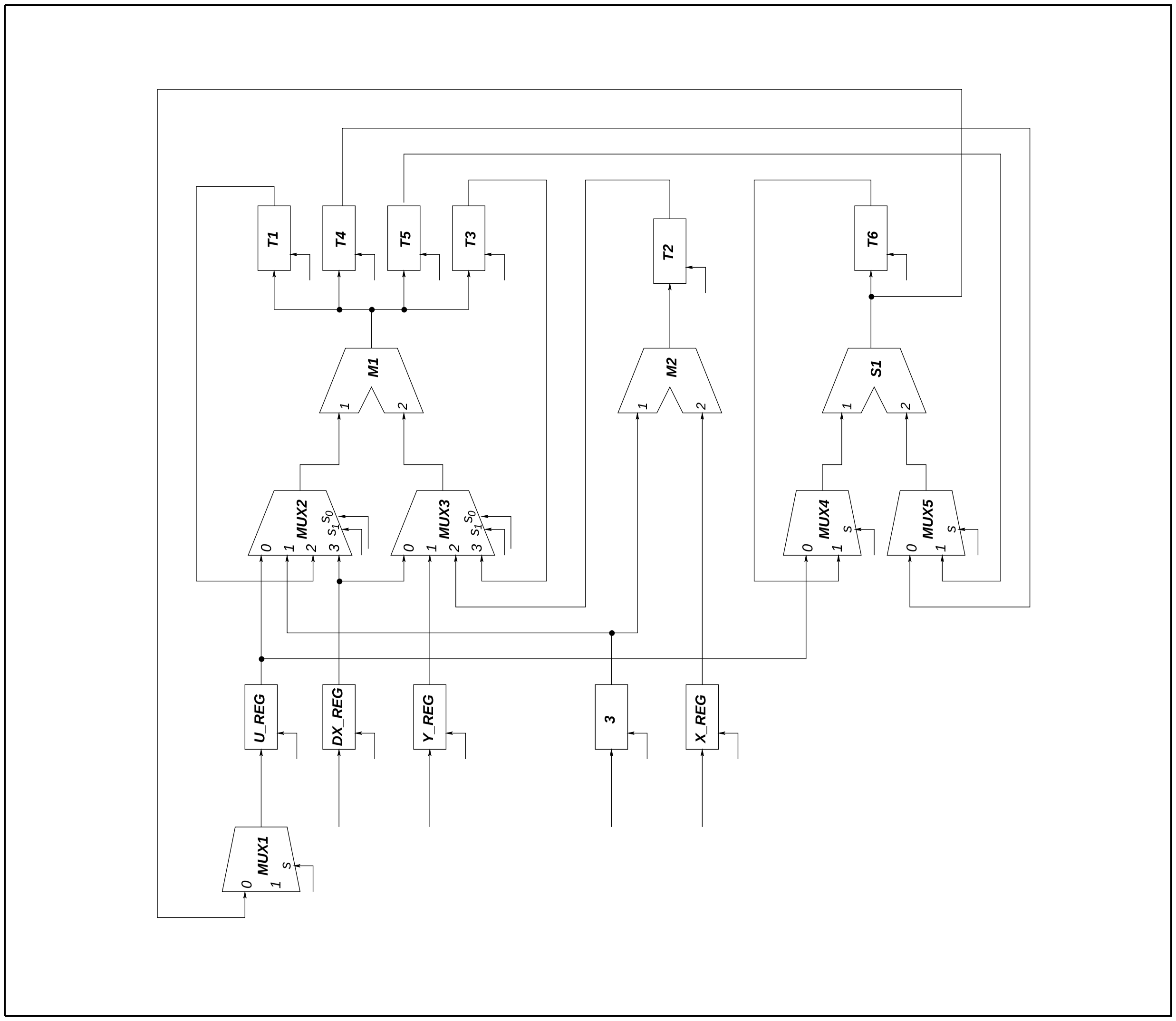


Figure 9.10: **Data path of the design synthesized from the schedule of Figure 9.9**

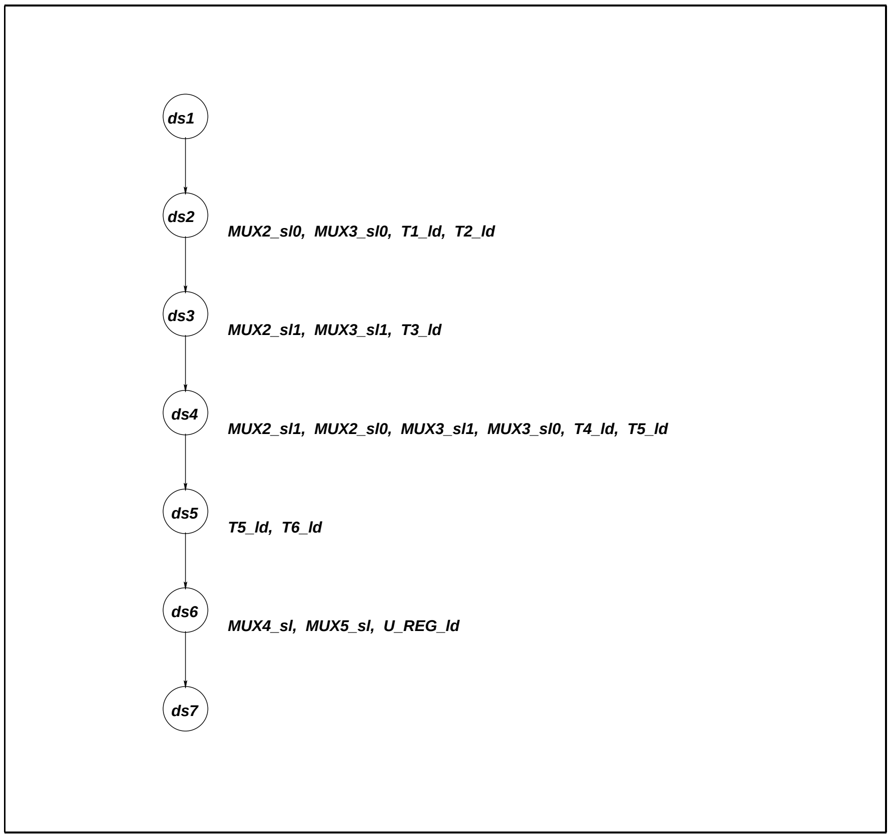

ds1
ds2
MUX2_sl0, MUX3_sl0, T1_ld, T2_ld
ds3
MUX2_sl1, MUX3_sl1, T3_ld
ds4
MUX2_sl1, MUX2_sl0, MUX3_sl1, MUX3_sl0, T4_ld, T5_ld
ds5
T5_ld, T6_ld
ds6
MUX4_sl, MUX5_sl, U_REG_ld
ds7

Figure 9.11: **Controller of the design synthesized from the schedule of Figure 9.9**

$$\textbf{2D\_Pipelined\_Multiplier\_ax}: \forall \begin{array}{ll} (in_1 : & signal, \\ in_2 : & signal, \\ out : & signal, \\ rd : & bool\_signal, \\ ds_1 : & RTL\_state, \\ ds_2 : & RTL\_state, \\ ds_3 : & RTL\_state) : \end{array}$$

$$\begin{array}{l} (2D\_Pipelined\_multiplier(in_1, in_2, rd, out) \;\wedge \\ \qquad trans(ds_1, ds_2) \;\wedge\; trans(ds_2, ds_3)) \;\Rightarrow \\ \qquad\qquad V_d(out, ds_3) = mul(V_d(in_1, ds_1), V_d(in_2, ds_1))) \end{array}$$

$$\textbf{4\_to\_1\_Multiplexer\_ax}: \forall \begin{array}{ll} (in_0 : & signal, \\ in_1 : & signal, \\ in_2 : & signal, \\ in_3 : & signal, \\ out : & signal, \\ sl_0 : & bool\_signal, \\ sl_1 : & bool\_signal, \\ ds : & RTL\_state) : \end{array}$$

$$\begin{array}{l} multiplexer\_4\_1(in_0, in_1, in_2, in_3, sl_0, sl_1, out) \;\Rightarrow \\ \qquad ((\neg V_d(sl_1, ds) \;\wedge\; \neg V_d(sl_0, ds)) \;\Rightarrow\; V_d(out, ds) = V_d(in_0, ds) \;\wedge \\ \qquad (\neg V_d(sl_1, ds) \;\wedge\; V_d(sl_0, ds)) \;\Rightarrow\; V_d(out, ds) = V_d(in_1, ds) \;\wedge \\ \qquad (V_d(sl_1, ds) \;\wedge\; \neg V_d(sl_0, ds)) \;\Rightarrow\; V_d(out, ds) = V_d(in_2, ds) \;\wedge \\ \qquad (V_d(sl_1, ds) \;\wedge\; V_d(sl_0, ds) \;\Rightarrow\; V_d(out, ds) = V_d(in_3, ds)) \end{array}$$

$$\textbf{Register\_ax}: \forall \begin{array}{ll} (in : & signal, \\ out : & signal, \\ ld : & bool\_signal, \\ ds_1 : & RTL\_state, \\ ds_2 : & RTL\_state) : \end{array}$$

$$(register(in, ld, out) \;\wedge\; transition(ds_1, ds_2)) \;\Rightarrow$$
$$(V_d(ld, ds_1) \;\Rightarrow\; V_d(out, ds_2) = V_d(in, ds_1) \;\wedge$$
$$\neg V_d(ld, ds_1) \;\Rightarrow\; V_d(out, ds_2) = V_d(out, ds_1))$$

$$\begin{array}{lll} \textbf{Subtractor\_ax}: \;\forall & (in_1: & signal, \\ & in_2: & signal, \\ & out: & signal, \\ & ds: & RTL\_state): \end{array}$$

$$(subtractor(in_1, in_2, out) \;\Rightarrow\; V_d(out, ds) = sub(V_d(in_1, ds), V_d(in_2, ds))$$

Figure 9.12: **Description of behavior of the components in the data-path of Figure 9.10**

$$\begin{aligned}
&transition(ds_2, ds_3) \ \wedge \\
&transition(ds_3, ds_4) \ \wedge \\
&transition(ds_4, ds_5) \ \wedge \\
&transition(ds_5, ds_6) \ \wedge \\
&transition(ds_6, ds_7))
\end{aligned}$$

For this example also, the following is true:

$$CR_b = \{u\_var, x\_var, y\_var, dx\_var, t_1, t_2, t_3, t_4, t_5, t_6\} \tag{9.4}$$

$$CR_d = \{U\_REG, X\_REG, Y\_REG, DX\_REG, T_1, T_2, T_3, T_4, T_5, T_6\} \tag{9.5}$$

$$\begin{aligned}B_r = \{&u\_var \mapsto U\_REG,\ x\_var \mapsto X\_REG,\ y\_var \mapsto Y\_REG,\ dx\_var \mapsto DX\_REG,\\ &t_1 \mapsto T1,\ t_2 \mapsto T2,\ t_3 \mapsto T3,\ t_4 \mapsto T4,\ t_5 \mapsto T5,\ t_6 \mapsto T6\}\end{aligned} \tag{9.6}$$

Assumption 9.2

$$\begin{aligned}
&(\langle bs_1, bs_8\rangle \overset{B_s}{\mapsto} \langle ds_1, ds_7\rangle) \ \wedge \\
&\quad inter\_block\_transition(bs_1, bs_8) \ \wedge \\
&\qquad inter\_block\_transition(ds_1, ds_7) \ \wedge \\
&\qquad\quad (V_b(CR_b, bs_1) \overset{B_r}{\equiv} V_d(CR_d, ds_1)))
\end{aligned}$$

Lemma 9.2

$$(V_b(u\_var, bs_8) = V_d(U\_REG, ds_7)$$

Proof 9.2

(1) $transition(bs_1, bs_2)$
(2) $transition(bs_2, bs_3)$
(3) $transition(bs_3, bs_4)$
(4) $transition(bs_4, bs_5)$
(5) $transition(bs_5, bs_6)$
(6) $transition(bs_6, bs_7)$
(7) $transition(bs_7, bs_8)$

(8) $V_b(u\_var, bs_6) = V_b(u\_var, bs_5)$
(9) $V_b(u\_var, bs_5) = V_b(u\_var, bs_4)$
(10) $V_b(u\_var, bs_4) = V_b(u\_var, bs_3)$
(11) $V_b(u\_var, bs_3) = V_b(u\_var, bs_2)$
(12) $V_b(u\_var, bs_2) = V_b(u\_var, bs_1)$
(13) $V_b(u\_var, bs_6) = V_b(u\_var, bs_1)$ *(8)-(12)*

(14) $V_b(t_1, bs_4) = V_b(t_1, bs_3)$
(15) $V_b(t_1, bs_3) = V_b(t_1, bs_2)$
(16) $V_b(t_1, bs_2) = mul(V_b(u\_var, bs_1), V_b(dx_var, bs_1))$
(17) $V_b(t_1, bs_4) = mul(V_b(u\_var, bs_1), V_b(dx_var, bs_1))$ *(14)-(15)*

(18) $V_b(x\_var, bs_2) = V_b(x\_var, bs_1)$

(19) $V_b(t_2, bs_4) = V_b(t_2, bs_3)$
(20) $V_b(t_2, bs_3) = mul(const(3), V_b(x\_var, bs_2))$
(21) $V_b(t_2, bs_3) = mul(const(3), V_b(x\_var, bs_1))$ (18)
(22) $V_b(t_2, bs_4) = mul(const(3), V_b(x\_var, bs_1))$ (19)-(21)

(23) $V_b(t_4, bs_6) = V_b(t_4, bs_5)$
(24) $V_b(t_4, bs_5) = mul(V_b(t_1, bs_4), V_b(t_2, bs_4)$
(25) $V_b(t_4, bs_5) = mul(mul(V_b(u\_var, bs_1), V_b(dx_var, bs_1)), mul(const(3), V_b(x\_var, bs_1)))$ (17),(22)
(26) $V_b(t_4, bs_6) = mul(mul(V_b(u\_var, bs_1), V_b(dx_var, bs_1)), mul(const(3), V_b(x\_var, bs_1)))$ (23)-(25)

(27) $V_b(t_6, bs_7) = sub(V_b(u\_var, bs_6), V_b(t_4, bs_6))$
(28) $V_b(t_6, bs_7) = sub(V_b(u\_var, bs_1), mul(mul(V_b(u\_var, bs_1), V_b(dx_var, bs_1)), mul(const(3), V_b(x\_var, bs_1))))$ (13),(26)

(29) $V_b(dx\_var, bs_5) = V_b(dx\_var, bs_4)$
(30) $V_b(dx\_var, bs_4) = V_b(dx\_var, bs_3)$
(31) $V_b(dx\_var, bs_3) = V_b(dx\_var, bs_2)$
(32) $V_b(dx\_var, bs_2) = V_b(dx\_var, bs_1)$
(33) $V_b(dx\_var, bs_5) = V_b(dx\_var, bs_1)$ (29)-(32)

(34) $V_b(y\_var, bs_3) = V_b(y\_var, bs_2)$

(35) $V_b(y\_var, bs_2) = V_b(y\_var, bs_1)$

(36) $V_b(y\_var, bs_3) = V_b(y\_var, bs_1)$ *(34)-(35)*

(37) $V_b(t_3, bs_5) = V_b(t_3, bs_4)$

(38) $V_b(t_3, bs_4) = mul(const(3), V_b(y\_var, bs_3))$

(39) $V_b(t_3, bs_5) = mul(const(3), V_b(y\_var, bs_3))$ *(37)-(38)*

(40) $V_b(t_3, bs_5) = mul(const(3), V_b(y\_var, bs_1))$ *(36)*

(41) $V_b(t_5, bs_7) = V_b(t_5, bs_6)$

(42) $V_b(t_5, bs_6) = mul(V_b(dx\_var, bs_5), V_b(t_3, bs_5))$

(43) $V_b(t_5, bs_7) = mul(V_b(dx\_var, bs_5), V_b(t_3, bs_5))$ *(41)-(42)*

(44) $V_b(t_5, bs_7) = mul(V_b(dx\_var, bs_1), mul(const(3), V_b(y\_var, bs_1)))$ *(33),(40)*

(45) $V_b(u\_var, bs_8) = sub(V_b(t_6, bs_7), V_b(t_5, bs_7))$

(46) $V_b(u\_var, bs_8) = sub(sub(V_b(u\_var, bs_1), mul(mul(V_b(u\_var, bs_1), V_b(dx\_var, bs_1)), mul(const(3), V_b(x\_var, bs_1)))), mul(V_b(dx\_var, bs_1), mul(const(3), V_b(y\_var, bs_1))))$ *(28),(44)*

(47) $transition(ds_1, ds_2)$

(48) $transition(ds_2, ds_3)$

(49) $transition(ds_3, ds_4)$

(50) $transition(ds_4, ds_5)$

(51) $transition(ds_5, ds_6)$

(52) $transition(ds_6, ds_7)$

(53) $V_d(U\_REG.out, ds_5) = V_d(U\_REG.out, ds_4)$

(54) $V_d(U\_REG.out, ds_4) = V_d(U\_REG.out, ds_3)$

(55) $V_d(U\_REG.out, ds_3) = V_d(U\_REG.out, ds_2)$

(56) $V_d(U\_REG.out, ds_2) = V_d(U\_REG.out, ds_1)$

(57) $V_d(U\_REG.out, ds_5) = V_d(U\_REG.out, ds_1)$ *(53)-(56)*

(58) $V_d(S_1.in1, ds_5) = V_d(MUX_4.out, ds_5)$
(59) $V_d(MUX_4.out, ds_5) = V_d(MUX_4.in0, ds_5)$
(60) $V_d(MUX_4.in0, ds_5) = V_d(U\_REG.out, ds_5)$
(61) $V_d(U\_REG.out, ds_5) = V_d(U\_REG.out, ds_1)$ *(57)*
(62) $V_d(S_1.in1, ds_5) = V_d(U\_REG.out, ds_1)$ *(58)-(61)*

(63) $V_d(M_1.in1, ds_1) = V_d(MUX_2.out, ds_1)$
(64) $V_d(MUX_2.out, ds_1) = V_d(MUX_2.in0, ds_1)$
(65) $V_d(MUX_2.in0, ds_1) = V_d(U\_REG.out, ds_1)$
(66) $V_d(M_1.in1, ds_1) = V_d(U\_REG.out, ds_1)$ (63)-(65)

(67) $V_d(M_1.in2, ds_1) = V_d(MUX_3.out, ds_1)$
(68) $V_d(MUX_3.out, ds_1) = V_d(MUX_3.in0, ds_1)$
(69) $V_d(MUX_3.in0, ds_1) = V_d(DX\_REG.out, ds_1)$
(70) $V_d(M_1.in2, ds_1) = V_d(DX\_REG.out, ds_1)$ (67)-(69)

(71) $V_d(T_1.out, ds_3) = V_d(T_1.in, ds_2)$
(72) $V_d(T_1.in, ds_2) = V_d(M_1.out, ds_2)$
(73) $V_d(M_1.out, ds_2) = mul(V_d(M_1.in1, ds_1),\ V_d(M_1.in2, ds_1))$
(74) $V_d(M_1.out, ds_2) = mul(V_d(U\_REG.out, ds_1),\ V_d(DX\_REG.out, ds_1)$ (66),(70)
(75) $V_d(T_1.out, ds_3) = mul(V_d(U\_REG.out, ds_1),\ V_d(DX\_REG.out, ds_1)$ (71)-(74)

(76) $V_d(M_1.in1, ds_3) = V_d(MUX_2.out, ds_3)$
(77) $V_d(MUX_2.out, ds_3) = V_d(MUX_2.in2, ds_3)$
(78) $V_d(MUX_2.in2, ds_3) = V_d(T_1.out, ds_3)$
(79) $V_d(T_1.out, ds_3) = mul(V_d(U\_REG.out, ds_1), V_d(DX\_REG.out, ds_1))$ (75)
(80) $V_d(M_1.in1, ds_3) = mul(V_d(U\_REG.out, ds_1), V_d(DX\_REG.out, ds_1))$ (76)-(79)

(81) $V_d(M_2.in1, ds_1) = const(3)$

(82) $V_d(M_2.in2, ds_1) = V_d(X\_REG.out, ds_1)$

(83) $V_d(T_2.out, ds_3) = V_d(T_2.in, ds_2)$
(84) $V_d(T_2.in, ds_2) = V_d(M_2.out, ds_2)$
(85) $V_d(M_2.out, ds_2) = mul(V_d(M_2.in1, ds_1), V_d(M_2.in2, ds_1)$

(86) $V_d(M_2.out, ds_2) = mul(const(3), V_d(X\_REG.out, ds_1))$ (81),(82)

(87) $V_d(T_2.out, ds_3) = mul(const(3), V_d(X\_REG.out, ds_1))$ (83)-(86)

(88) $V_d(M_1.in2, ds_3) = V_d(MUX_3.out, ds_3)$

(89) $V_d(MUX_3.out, ds_3) = V_d(MUX_3.in2, ds_3)$

(90) $V_d(MUX_3.in2, ds_3) = V_d(T_2.out, ds_3)$

(91) $V_d(T_2.out, ds_3) = mul(const(3), V_d(X\_REG.out, ds_1))$ (87)

(92) $V_d(M_1.in2, ds_3) = mul(const(3), V_d(X\_REG.out, ds_1))$ (88)-(91)

(93) $V_d(T_4.out, ds_5) = V_d(T_4.in, ds_4)$

(94) $V_d(T_4.in, ds_4) = V_d(M_1.out, ds_4)$

(95) $V_d(M_1.out, ds_4) = mul(V_d(M_1.in1, ds_3), V_d(M_1.in2, ds_3))$

(96) $V_d(M_1.out, ds_4) = mul(mul(V_d(U\_REG.out, ds_1), V_d(DX\_REG.out, ds_1)),$
$mul(const(3), V_d(X\_REG.out, ds_1))),$ (80),(92)

(97) $V_d(T_4.out, ds_5) = mul(mul(V_d(U\_REG.out, ds_1), V_d(DX\_REG.out, ds_1)),$
$mul(const(3), V_d(X\_REG.out, ds_1)))$ (93)-(96)

(98) $V_d(S_1.in2, ds_5) = V_d(MUX_5.out, ds_5)$

(99) $V_d(MUX_5.out, ds_5) = V_d(MUX_5.in0, ds_5)$

(100) $V_d(MUX_5.in0, ds_5) = V_d(T_4.out, ds_5)$

(101) $V_d(T_4.out, ds_5) = mul(mul(V_d(U\_REG.out, ds_1), V_d(DX\_REG.out, ds_1)),$
$mul(const(3), V_d(X\_REG.out, ds_1)))$ (97)

(102) $V_d(S_1.in2, ds_5) = mul(mul(V_d(U\_REG.out, ds_1), V_d(DX\_REG.out, ds_1)),$
$mul(const(3), V_d(X\_REG.out, ds_1)))$ (98)-(101)

(103) $V_d(T_6.out, ds_6) = V_d(T_6.in, ds_5)$

(104) $V_d(T_6.in, ds_5) = V_d(S_1.out, ds_5)$

(105) $V_d(S_1.out, ds_5) = sub(V_d(S_1.in1, ds_5), V_d(S_1.in2, ds_5))$

(106) $V_d(S_1.out, ds_5) = sub(V_d(U\_REG.out, ds_1),$
$mul(mul(V_d(U\_REG.out, ds_1),$
$V_d(DX\_REG.out, ds_1)),$
$mul(const(3),$
$V_d(X\_REG.out, ds_1))))$ (62),(102)

(107) $V_d(T_6.out, ds_6) = sub(V_d(U\_REG.out, ds_1),$
$\quad mul(mul(V_d(U\_REG.out, ds_1),$
$\quad\quad V_d(DX\_REG.out, ds_1)),$
$\quad mul(const(3),$
$\quad\quad V_d(X\_REG.out, ds_1))))$ (103)-(106)

(108) $V_d(S_1.in1, ds_6) = V_d(MUX_4.out, ds_6)$
(109) $V_d(MUX_4.out, ds_6) = V_d(MUX_4.in1, ds_6)$
(110) $V_d(MUX_4.in1, ds_6) = V_d(T_6.out, ds_6)$
(111) $V_d(T_6.out, ds_6) = sub(V_d(U\_REG.out, ds_1),$
$\quad mul(mul(V_d(U\_REG.out, ds_1),$
$\quad\quad V_d(DX\_REG.out, ds_1)),$
$\quad mul(const(3),$
$\quad\quad V_d(X\_REG.out, ds_1))))$ (107)
(112) $V_d(S_1.in1, ds_6) = sub(V_d(U\_REG.out, ds_1),$
$\quad mul(mul(V_d(U\_REG.out, ds_1),$
$\quad\quad V_d(DX\_REG.out, ds_1)),$
$\quad mul(const(3),$
$\quad\quad V_d(X\_REG.out, ds_1))))$ (108)-(111)

(113) $V_d(DX\_REG.out, ds_4) = V_d(DX\_REG.out, ds_3)$
(114) $V_d(DX\_REG.out, ds_3) = V_d(DX\_REG.out, ds_2)$
(115) $V_d(DX\_REG.out, ds_2) = V_d(DX\_REG.out, ds_1)$
(116) $V_d(DX\_REG.out, ds_4) = V_d(DX\_REG.out, ds_1)$ (113)-(115)

(117) $V_d(M_1.in1, ds_4) = V_d(MUX_2.out, ds_4)$
(118) $V_d(MUX_2.out, ds_4) = V_d(MUX_2.in3, ds_4)$
(119) $V_d(MUX_2.in3, ds_4) = V_d(DX\_REG.out, ds_4)$
(120) $V_d(DX\_REG.out, ds_4) = V_d(DX\_REG.out, ds_1)$ (116)
(121) $V_d(M_1.in1, ds_4) = V_d(DX\_REG.out, ds_1)$ (117)-(120)

(122) $V_d(M_1.in1, ds_2) = V_d(MUX_2.out, ds_2)$
(123) $V_d(MUX_2.out, ds_2) = V_d(MUX_2.in1, ds_2)$
(124) $V_d(MUX_2.in1, ds_2) = const(3)$
(125) $V_d(M_1.in1, ds_2) = const(3)$

(126) $V_d(M_1.in2, ds_2) = V_d(MUX_3.out, ds_2)$

(127) $V_d(MUX_3.out, ds_2) = V_d(MUX_3.in_1, ds_2)$

(128) $V_d(MUX_3.in_1, ds_2) = V_d(Y\_REG.out, ds_2)$

(129) $V_d(Y\_REG.out, ds_2) = V_d(Y\_REG.out, ds_1)$

(130) $V_d(M_1.in2, ds_2) = V_d(Y\_REG.out, ds_1)$ (126)-(129)

(131) $V_d(T_3.out, ds_4) = V_d(T_3.in, ds_3)$

(132) $V_d(T_3.in, ds_3) = V_d(M_1.out, ds_3)$

(133) $V_d(M_1.out, ds_3) = mul(V_d(M_1.in1, ds_2), V_d(M_1.in2, ds_2))$

(134) $V_d(M_1.out, ds_3) = mul(const(3), V_d(Y\_REG.out, ds_1))$ (125),(130)

(135) $V_d(T_3.out, ds_4) = mul(const(3), V_d(Y\_REG.out, ds_1))$ (131)-(134)

(136) $V_d(M_1.in2, ds_4) = V_d(MUX_3.out, ds_4)$

(137) $V_d(MUX_3.out, ds_4) = V_d(MUX_3.in3, ds_4)$

(138) $V_d(MUX_3.in3, ds_4) = V_d(T_3.out, ds_4)$

(139) $V_d(T_3.out, ds_4) = mul(const(3), V_d(Y\_REG.out, ds_1))$ (135)

(140) $V_d(M_1.in2, ds_4) = mul(const(3), V_d(Y\_REG.out, ds_1))$ (136)-(139)

(141) $V_d(T_5.out, ds_6) = V_d(T_5.in, ds_5)$

(142) $V_d(T_5.in, ds_5) = V_d(M_1.out, ds_5)$

(143) $V_d(M_1.out, ds_5) = mul(V_d(M_1.in1, ds_4), V_d(M_1.in2, ds_4))$

(144) $V_d(M_1.out, ds_5) = mul(V_d(DX\_REG.out, ds_1), mul(const(3), V_d(Y\_REG.out, ds_1)))$ (121),(140)

(145) $V_d(T_5.out, ds_6) = mul(V_d(DX\_REG.out, ds_1), mul(const(3), V_d(Y\_REG.out, ds_1)))$ (141)-(144)

(146) $V_d(S_1.in2, ds_6) = V_d(MUX_5.out, ds_6)$

(147) $V_d(MUX_5.out, ds_6) = V_d(MUX_5.in1, ds_6)$

(148) $V_d(MUX_5.in1, ds_6) = V_d(T_5.out, ds_6)$

(149) $V_d(T_5.out, ds_6) = mul(V_d(DX\_REG.out, ds_1), mul(const(3), V_d(Y\_REG.out, ds_1)))$ (145)

(150) $V_d(S_1.in2, ds_6) = mul(V_d(DX\_REG.out, ds_1), mul(const(3), V_d(Y\_REG.out, ds_1)))$ (146)-(149)

(151) $V_d(U\_REG.out, ds_7) = V_d(U\_REG.in, ds_6)$

(152) $V_d(U\_REG.in, ds_6) = V_d(MUX_1.out, ds_6)$

(153) $V_d(MUX_1.out, ds_6) = V_d(MUX_1.in0, ds_6)$

(154) $V_d(MUX_1.in0, ds_6) = V_d(S_1.out, ds_6)$

(155) $V_d(S_1.out, ds_6) = sub(V_d(S_1.in1, ds_6), V_d(S_1.in2, ds_6)))$

(156) $V_d(S_1.out, ds_6) = sub(sub(V_d(U\_REG.out, ds_1), mul(mul(V_d(U\_REG.out, ds_1), V_d(DX\_REG.out, ds_1)), mul(const(3), V_d(X\_REG.out, ds_1)))), mul(V_d(DX\_REG.out, ds_1) mul(const(3), V_d(Y\_REG.out, ds_1))))$ (112),(150)

(157) $V_d(U\_REG.out, ds_7) = sub(sub(V_d(U\_REG.out, ds_1), mul(mul(V_d(U\_REG.out, ds_1), V_d(DX\_REG.out, ds_1)), mul(const(3), V_d(X\_REG.out, ds_1)))), mul(V_d(DX\_REG.out, ds_1) mul(const(3), V_d(Y\_REG.out, ds_1))))$ (151)-(156)

(158) $V_b(u\_var, bs_8) = V_d(U\_REG.out, ds_7)$ (46),(157)

(159) $Q.E.D.$

Discussion

The main difference in verification of the designs with general and with pipelined multi-cycle resources is due to the differences in the formal descriptions of these components. For a general multi-cycle design to function correctly, the inputs should be held stable during the its operation cycle. This is translated into verification preconditions that should be met for correct operation of a multi-cycle resource. However, this is not required in the case of pipelined resources. During the verification of a multi-cycle design, it needs to be proven that these preconditions are satisfied.

A pipelined multi-cycle component however, may consume multiple sets of data during its operation cycle. There is simply a delay corresponding to the latency of the pipelined resource between the time that a new set of data is input to the component and the time that the response of the component to that set of data appears at the output of the component. Therefore, there are no preconditions for the correct operation of a pipelined multi-cycle design. Due to this difference, during the verification of pipelined and non-pipelined designs, different proof steps are taken. In other words, different proof tactics are used for verification of designs with pipelined versus non-pipelined multi-cycle resources, but the proof strategy remains the same.

In our particular example, the difference between the two implementations is mainly due to the different multipliers that are used in synthesis of the two designs. When comparing the formal descriptions of these multipliers in Figures 9.1.1 and 9.2, the difference in the functional preconditions can be noted. When analyzing the register transfer operations in these designs, the output of the pipelined multiplier at each state is simply the product of its input signals at the previous state. However, for this to be true for the non-pipelined multiplier, it should be additionally verified that the inputs have been stable throughout the 2 cycles of the operation of the multiplier. During the verification process, this is translated into additional proof steps. These additional steps are marked with $*$'s in the proof script.

9.3 Synthesis with Resource Constraints

When deriving schedules from behavioral specifications, there is a trade off between the area (number of resources used) and the latency that may be achieved. In scheduling with no resource constraints, the goal is to generate a schedule with minimum number of resources, with a fixed given latency. To achieve this there should be no limit on the number of the resources that may be used during the synthesis of the design. In scheduling with resource constraints on the other hand, the goal is to generate a schedule with the minimum latency, with a fixed number of resources. For this to be possible, there is no limit on the latency of the design, or if there is, the scheduling may not be possible. In synthesis with resource constraints also, non-pipelined and/or pipelined resources may be used.

Figures 9.13 and 9.14 show two other possible schedules for the partial behavior specification of Figure 9.2. In both these schedules, it is assumed that one multiplier and one subtractor are the only resources available. Figure 9.13 corresponds to a schedule with non-pipelined resources. The formal description of the resources used in this schedule are given in Figure 9.1.1. Figure 9.14 corresponds to a schedule with the same constraints but with pipelined resources. The formal description of the resources used in this schedule are given in Figure 9.2. The advantages of pipelined resources is more evident in the case of scheduling with resource constraints: the schedule with non-pipelined resources has a latency of 11 cycles, while with the same number of pipelined resources a latency of 7 cycles may be achieved in this example.

When resource constraints are present during the synthesis, the schedules are longer in order not to

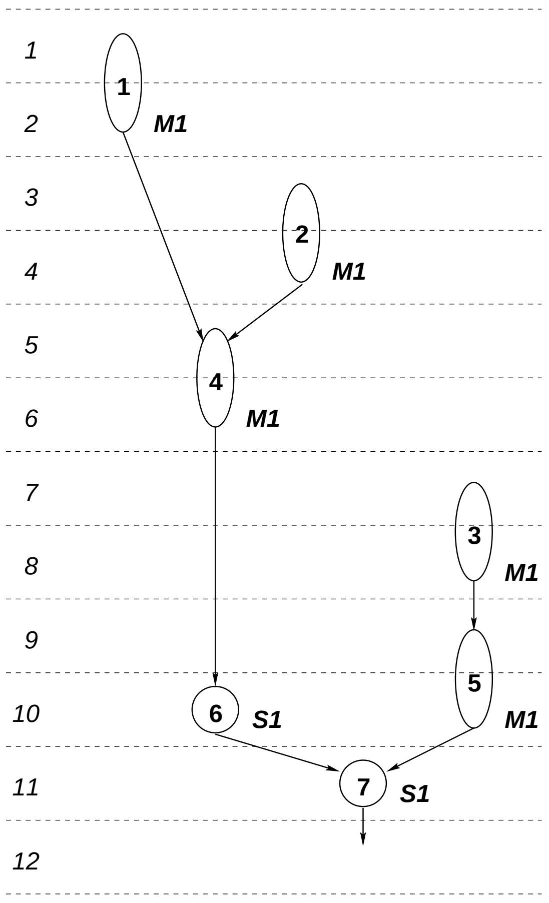


Figure 9.13: **A possible schedule with constrained non-pipelined resources for the partial specification of Figure 9.2**

violate data dependencies. However, scheduling with resource constraints does not introduce new dependencies between the operations within a synthesis basic block and the operations outside the border points of the block. Consequently, our verification method can be safely used for verification of designs with resource constraints. The verification of the designs with non-pipelined constrained resources is similar to the verification of the designs with non-pipelined unconstrained resources, and verification of the designs with pipelined constrained resources is similar to the verification of the designs with pipelined unconstrained resources. Since in the case of the designs with resource constraints, the latency is longer, during the verification process, the number of the states at which the register transfer operations should be analyzed increases and consequently, the correctness proofs will be longer.

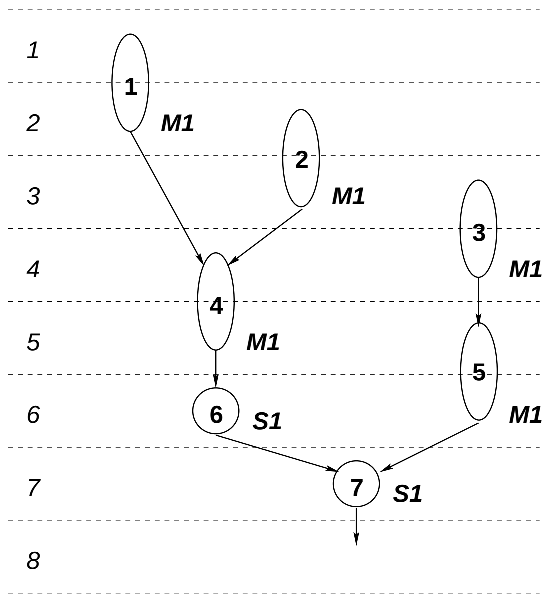


Figure 9.14: **A possible schedule with constrained pipelined resources for the partial specification of Figure 9.2**

9.4 Conclusion

In this chapter we discussed the class of multi-cycle synthesis resources and in particular pipelined multi-cycle resources. Unlike purely combinational synthesis resources that operate on each set of data in a fraction of a clock cycle, the operation of multi-cycle resources spans over multiple clock cycles. We showed that our verification method can be extended to accommodate verification of RTL designs that are synthesized from multi-cycle pipelined or non-pipelined resources. This may be simply done when developing the library of formal descriptions of synthesis resources. The formal descriptions of these components are written such that they capture their particular behavior. This particular behavior is reflected during the verification exercise when the formal descriptions of these components are instantiated.

Chapter 10

Loop Winding and Functional Pipelining

The focus of this chapter is a design technique named *loop winding* or *loop folding*, and special issues in verification of designs that have been synthesized using this technique. Loop winding is used to optimize the execution delay of a loop. When loop winding is performed the consecutive iterations of a loop may execute concurrently. Therefore, the synthesis and verification of such a loop is different from the synthesis and verification of a non-pipelined loop. Loop winding deserves detailed analysis since it is also the basis of functional pipelining of the designs. But, before we discuss this subject we present the related topic of verification of general loop implementations.

10.1 Verification of Non-Pipelined Loops

In previous chapters we discussed that the correctness of a synthesized RTL design with respect to its behavior specification may be verified by proving the equivalence of the corresponding pairs of behavior-design critical paths. We claim that under the condition that no code motion across the border points of basic blocks is performed during the synthesis, the equivalence of corresponding pairs of critical path also implies that each iterative block of the controller of the RTL design along with its data-path correctly implement its corresponding iterative block in the behavior. That is to say in one pass through the body of each iterative block of RTL design, its correctness with respect to its specification may be verified. Let's first consider the non-hierarchical iterative blocks or iterative basic blocks.

Consider the Figure 10.1 representing an iterative block of the controller of a synthesized design, with the condition flag l_d. In this figure, the loop is entered with a single state transition from the

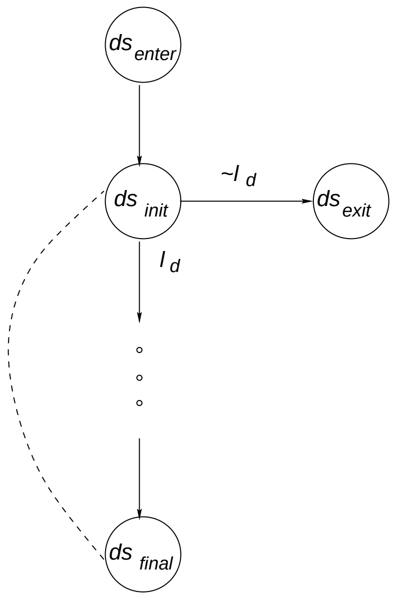


Figure 10.1: **A simple loop basic block**

state ds_{enter}, and it is exited with a single transition to the state ds_{exit}. ds_{init} and ds_{final} are the initial and final states of the body of the loop, respectively. Note that the curved line connecting ds_{init} and ds_{final} in Figure 10.1 means that ds_{init} and ds_{final} are in fact the same state (the initial and final states of a loop are the same). We have assigned two different labels to this state to distinguish between the case when it the is the initial state of an iteration and the case when it is the final state of the same iteration. This loop is an implementation of a loop with a similar control structure in the behavior specification. We define the states bs_{enter}, bs_{init}, bs_{final} and bs_{exit} of the loop specification similar to their counterparts in the controller, and denote the condition variable of the specification loop by l_b. The following nomenclature summarizes the elements of the loop specification:

l_b : loop specification condition variable
$succ_b(s_b)$: the state succeeding s_b in the body of the loop
bs_{init} : the initial state of the loop at each iteration
bs_{enter} : the state preceding the initial state of the loop
bs_{final} : the final state of the body of the loop
bs_{exit} : the state following the final state of the loop
$V_b^i(r_b, s_b)$: the value function representing the value of variable r_b at state s_b of iteration i

The following nomenclature summarizes the elements of the loop implementation:

l_d	loop implementation condition flag
$succ_d(s_d)$	the state succeeding s_d in the body of the loop
bs_{init}	the initial state of the loop at each iteration
bs_{enter}	the state preceding the initial state of the loop
bs_{final}	the final state of the body of the loop
bs_{exit}	the state following the final state of the loop
$V_d^i(r_d, s_d)$	the value function representing the value of register r_d at state s_d of iteration i

When verifying the loop implementation, we assume that just upon a state transition from ds_{enter} to ds_{init}, the initial state of the loop, the critical registers of the design have correct values. If the loop is correctly implemented, these registers will have correct values at the end of each iteration (correctness condition *(2)* below) and also just prior to the control exiting the loop (correctness condition *(3)* below). In addition, the loop will be repeated the same number of times as its behavioral specification (correctness condition *(1)* below). Then assuming that the initial states and final states of the specification of the loop and its implementation are bound together by the state binding function, this may be translated to the following three correctness conditions, for each iteration i of the loop:

1 $\quad V_d(l_d, ds_{init}) \equiv V_b(l_b, bs_{init})$

2 $\quad (V_d(CR_d, ds_{init}) \equiv V_b(CR_b, bs_{init})) \Rightarrow (V_d(CR_d, ds_{final}) \equiv V_b(CR_b, bs_{final}))$

3 $\quad \neg V_b(l_b, bs_{init}) \Rightarrow (V_d(CR_d, ds_{exit}) \equiv V_b(CR_b, bs_{exit}))$

Unlike the case of sequential basic blocks, in iterative basic blocks the content of each register along the critical path corresponding to the block is not simply a function of the states of the path, but it is also a function of the iteration count of the loop, i.e. if bs_j (ds_j) is a state and i is an iteration (where $1 \leq i \leq n_i$) of an iterative basic block then $V_b(CR_b, bs_j) = f_b(i)$ $\quad (V_d(CR_b, ds_j) = f_d(i))$. To be able to address all possible instances of variables at each state of a loop we introduce a superscript to the function V_b (or V_d). V_b^i (or V_d^i) has three arguments: two explicit arguments that are the name of a variable (or register), and a behavior state (or a design state) and an implicit argument i that is the iteration count of a loop. Then $V_b^i(r_b, s_b)$ (or $V_d^i(r_d, s_d)$ stands for the symbolic value of the variable r_b (or register r_d) at state s_b (or s_d) at i-th iteration of the loop. Please note that we have overloaded the term V_b^i (or V_b^i) to a vector version of itself so that:

$$V_b^i(\{a_1, a_2, \cdots, a_n\}, s_b) = \langle V_b^i(a_1, s_b), V_b^i(a_2, s_b), \cdots, V_b^i(a_n, s_b)\rangle$$

Now, the loop correctness condition may be restated as follows:

Theorem 10.1 *The loop implementation is correct iff:*

$$\begin{aligned}
(ds_{init} \ &= \ B_s(bs_{init}) \ \wedge \\
ds_{final} \ &= \ B_s(bs_{final}) \ \wedge \\
V_d^1(CR_d, ds_{init}) \ &\equiv \ V_b^1(CR_b, bs_{init}, 1)) \quad \Rightarrow
\end{aligned}$$

$$\begin{aligned}
(\forall i, \ i \in N^+ \ : \ & (V_b^i(l_b, bs_{init}) = V_d^i(l_d, ds_{init})) \ \wedge \\
& (V_d^i(l_d, ds_{init}) \ \Rightarrow \ (V_d^i(CR_d, ds_{final}) \equiv V_b^i(CR_b, bs_{final}))) \ \wedge \\
& (\neg V_d^i(l_d, ds_{init}) \ \Rightarrow \ (V_d^i(CR_d, ds_{exit}) \equiv V_b^i(CR_b, bs_{exit}))))
\end{aligned}$$

Then, the correctness of the loop may be stated as the conjunction of three correctness conditions that should be met at each iteration of the loop. One approach to verification of the loops is to *unroll* the loop and verify the correctness conditions at each iteration by instantiating i. Unrolling is referred to the process of replacing a loop that has a fixed number of iterations n_i, with a sequence of n_i copies of it (Figure 10.2). Unrolling a loop simplifies its analysis considerably, but verification of the loops through this approach is not attractive. The reason is that it is not applicable to the cases where the number of iterations of the loop is not known before its execution (e.g. a *while* loop in which the loop control variable is read from the input during the execution of the loop), even though this type of loops comprises a big percentage of the loops in the controller of synthesized designs in general. Besides, even for verification of the loops with fixed number of iterations, this approach is not efficient.

We believe that more efficient approaches for verification of a loop exist. Let's first consider a non-hierarchical non-pipelined iterative block. Various iterations of such a loop have symmetrical control flow. [1] This symmetry may be efficiently utilized for verification of the loop. We claim that due to the control flow symmetry, the loop may be verified by just one single pass through its body. The implementation of the loop is considered correct, if under the same assumptions as above, and assuming that the design registers have correct (symbolic) values at the initial state of an arbitrary iteration of the loop, we can prove that at the final state of that iteration they have correct (symbolic) values too. More formally the correctness condition of the loop may be states as follows:

[1] Since there is no nesting of iterative and conditional basic blocks within each other, and no overlap in execution of different iterations of a loop, at each iteration of the loop the exact same register transfer operations occur.

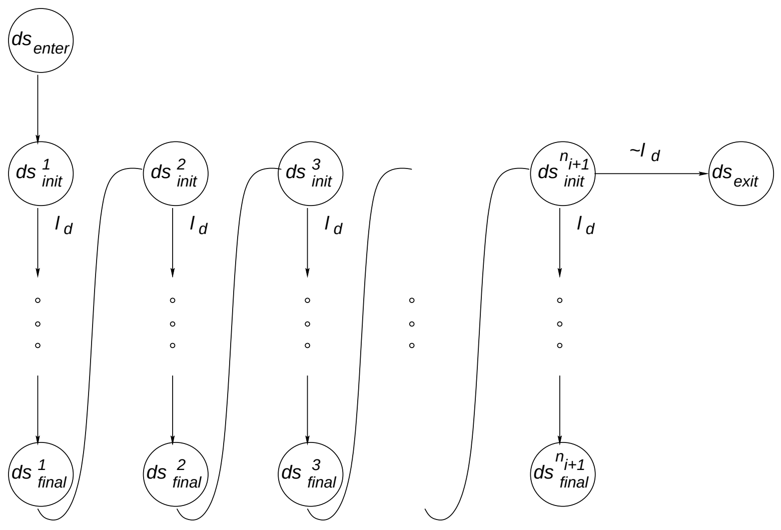


Figure 10.2: **The unrolling of a simple loop basic block**

Theorem 10.2 *The loop implementation is correct iff for an arbitrary iteration $k \in N^+$ of the loop we have:*

$$\begin{aligned}
&(ds_{init} \;=\; B_s(bs_{init}) \;\;\wedge \\
&\qquad ds_{final} \;=\; B_s(bs_{final}) \;\;\wedge \\
&\qquad\qquad V_d^1(CR_d, ds_{init}) \;\equiv\; V_b^1(CR_b, bs_{init})) \quad\Rightarrow \\
&(V_d^k(l_d, ds_{init}) \quad\Rightarrow \\
&\qquad V_d^k(CR_d, ds_{init}) \;\equiv\; V_b^k(CR_b, bs_{init}) \quad\Rightarrow\quad (V_d^k(CR_d, ds_{final}) \;\equiv\; V_b^k(CR_b, bs_{final})))
\end{aligned}$$

In order to show that Theorem 10.2 is a valid definition for the correctness of the loop, we need to prove that each of the three correctness conditions of the loop mentioned in Theorem 10.1 is a direct implication of that. Since both theorems are implications with the same antecedent, to prove the above implication it is enough to show that the consequent of Theorem 10.1 is a direct consequence of the consequent of Theorem 10.2, or that each of the three correctness conditions of the loop is a direct implication of the consequent of the Theorem 10.2. Before presenting this proof, we need to mention a few helpful properties of a loop in the form of premises:

Premise 10.1 $V_d^i(l_d, ds_{init}) \;\Rightarrow\; V_d^{i-1}(l_d, ds_{init})$

Explanation: Iteration i only occurs if the iteration $i-1$ has occurred.

Premise 10.2 $V_b^i(CR_b, bs_{init}) = V_b^{i-1}(CR_b, bs_{final})$ and $V_d^i(CR_d, ds_{init}) = V_b^{i-1}(CR_d, ds_{final})$.

Explanation: Based on definition bs_{init} at iteration i is the same as bs_{final} at iteration $i-1$, and ds_{init} at iteration i is the same as ds_{final} at iteration $i-1$.

Premise 10.3

$$\begin{aligned} &(\exists m,\ m \in N^+ : \\ &\qquad V_d^{m-1}(l_d, ds_{init}) = \text{“}T\text{”} \ \wedge \\ &\qquad V_d^{m}(l_d, ds_{init}) = \text{“}F\text{”}) \\ \Rightarrow\ &(\forall i,\ i \in N^+,\ i > m : \\ &\qquad V_d^{i}(l_d, ds_{init}) = V_d^{m}(l_d, ds_{init}) \ \wedge \\ &\qquad V_d^{i}(CR_d, ds_{init}) = V_d^{m}(CR_d, ds_{init})) \end{aligned}$$

Explanation: It is obvious that once the loop is exited, the iteration condition variable of the loop will remain false. The Premise 10.4 is an immediate implication of the Premise 10.3.

Premise 10.4 $V_d^i(l_d, ds_{init}) = \text{“}F\text{”} \quad \Rightarrow \quad V_d^{i+1}(l_d, ds_{init}) = \text{“}F\text{”}$.

Premise 10.5 $V_b^i(\langle bs_{init}) = \text{“}F\text{”} \ \Rightarrow\ (V_b(CR_b, bs_{exit}) = V_b^i(CR_b, bs_{init}))$ *and*
$V_d^i(\langle ds_{init}) = \text{“}F\text{”} \ \Rightarrow\ (V_d(CR_d, ds_{exit}) = V_d^i(CR_d, ds_{init}))$

Explanation: There are no behavior or register transfer operations corresponding to the initial state of a loop, therefore, the variables and registers preserve their values in a transfer from the initial state of the loop to the state following the loop.

Now we show through formal proof that the Theorem 10.1 is a direct consequence of the Theorem 10.2. The proof is based on the following premise.

Premise- We assume that it has been successfully proven that for an arbitrary iteration of the loop, if the registers have correct values at the initial state then they have correct values at the final state.

Then proof steps may be informally stated as follows: At the initial state of the first iteration of the loop, the registers hold correct values. Based on the above premise, the registers hold correct values at the final state of the first iteration of the loop. The contents of registers at the final state of the first iteration of the loop, is the same as their contents at the initial state of the second iteration of the loop. Among these registers is a register that its output is the control signal serving as the loop control flag. Then the content of this register will be the same as the value of its corresponding loop variable in the specification of the loop. This means that either both loops go through a second iteration or they both terminate. In the first case based on the above premise at the final state of the second iteration and consequently at the initial state of the third iteration of the loop, the registers hold correct values. In the second case, in both loops transitions to the state following the loops (bs_{exit} or ds_{exit}) occur. Since no operation corresponds to the initial state of the loop, then the contents of the variables and registers in a state transition from the initial state of the loop to the state following the loop is preserved (bs_{exit} or ds_{exit}). This means that at the state at which the loop is exited the registers hold correct values, also. The argument for the next iterations of the loop is similar. Therefore, the proof of the theorem is completed. The following steps present these proof steps more formally.

Assumption 10.1

For an arbitrary iteration k of the loop, where $k \in N^+$ the following is true:

$$\begin{aligned}(V_d^i(l_d, ds_{init}) \quad \wedge \quad (V_b^i(CR_b, bs_{init}) \;\equiv\; V_d^i(CR_d, ds_{init}))) \quad \Rightarrow& \\ (V_b^i(CR_b, bs_{final}) \;\equiv\; V_d^i(CR_d, ds_{final}))&\end{aligned}$$

Lemma 10.1

$$\begin{aligned}(\forall i,\; i \in N^+ \;:\; & V_b^i(l_b, bs_{init}) \;=\; V_d^i(l_d, ds_{init}) \;\wedge \\ & V_d^i(l_d, ds_{init}) \;\Rightarrow\; V_b^i(CR_b, bs_{final}) \;\equiv\; V_d^i(CR_d, ds_{final}) \;\wedge \\ & \neg V_d^i(l_d, ds_{init}) \;\Rightarrow\; V_b(CR_b, bs_{exit}) \;\equiv\; V_d(CR_d, ds_{exit}))\end{aligned}$$

Proof 10.1

Step1 - As the first step, we prove that assuming registers have correct values at the initial state of the first iteration of the loop, then if the loop goes through an iteration (if the loop condition flag of a particular iteration has true value) then the registers will have correct values at the initial state of the following iteration of the loop. This statement is the basis of the Sub-lemma 10.1 given below. A proof of this sub-lemma through the induction on the number of iterations of the loop follows.

Sub Lemma 10.1 Assuming that $(V_b^1(CR_b, bs_{init}) \equiv V_d^1(CR_d, ds_{init}))$, then:

$$\forall i \in N^+ \;:\; V_d^{i-1}(l_d, ds_{init}) \;\Rightarrow\; (V_b^i(CR_b, bs_{init}) \;\equiv\; V_d^i(CR_d, ds_{init}))$$

induction basis: It is obvious that for $i = 1$ the above proposition holds:

$$V_d^0((l_d, ds_{init}) \;\Rightarrow\; (V_b^1(CR_b, bs_{init}) \;\equiv V_d^1(CR_d, ds_{init}))$$

Even though $V_d^0(l_d, ds_{init})$ is 'false', since the consequent of the above proposition is 'true', the proposition will be 'true'.

induction hypothesis:

$$V_d^{k-1}(l_d, ds_{init}) \;\Rightarrow\; (V_b^k(CR_b, bs_{init}) \;\equiv\; V_d^k(CR_d, ds_{init}))$$

inductive step: We need to show that:

$$V_d^k(l_d, ds_{init}) \;\Rightarrow\; (V_b^{k+1}(CR_b, bs_{init}) \;\equiv\; V_d^{k+1}(CR_d, ds_{init}))$$

Proof:

assumption	$V_d^k(l_d, ds_{init})$	(1)
Premise 10.1	$V_d^{k-1}(l_d, ds_{init})$	(2)
induction hypothesis & (2)	$V_b^k(CR_b, bs_{init}) \;\equiv\; V_d^k(CR_d, ds_{init})$	(3)
Theorem 10.2 & (1) & (3)	$V_b^k(CR_b, bs_{final}) \;\equiv\; V_d^k(CR_d, ds_{final})$	(4)
Premise 10.1	$V_b^{k+1}(CR_b, bs_{init}) \;\equiv\; V_d^{k+1}(CR_d, ds_{init})$	(5)
(1) & (5)	$V_d^k(l_d, ds_{init}) \;\Rightarrow$ $(V_b^{k+1}(CR_b, bs_{init}) \;\equiv\; V_d^{k+1}(CR_d, ds_{init}))$	(6)
	$Q.E.D.$	(7)

The Sub-lemma 10.2 can be immediately deduced from the Assumption 10.1 and the Sub-lemma 10.1:

Sub Lemma 10.2

$$\forall i,\ i \in N^{+}\ :\ V_d^i(l_d, ds_{init})\ \Rightarrow\ V_d^i(CR_d, ds_{final}) \equiv V_b^i(CR_b, bs_{final})$$

Step2 - As the second step of the proof we need to show that the implementation and specification of the loop always repeat the same number of times, or in other words at the beginning of each iteration the value of the condition flag of the loop implementation is the same as the value of the condition variable of its specification. This is stated as the Sub-Lemma 10.3.

Sub Lemma 10.3

$$\forall i, i \in N^{+} : V_b^i(l_b, bs_{init}) \quad = \quad V_d^i(l_d, ds_{init})$$

Proof - Let's first consider the case where $i = 1$. We know that $V_b^1(CR_b, bs_{init}) \equiv V_d^1(CR_d, ds_{init})$, and since l_d is one of the critical registers, then $V_b^1(l_b, bs_{init}) \equiv V_d^1(l_d, ds_{init})$.

Now we need to prove the lemma when $i \neq 1$. We consider this in two separate cases, the case where $V_d^i(l_d, ds_{init}) =$ "T" and the case where $V_d^1(l_d, ds_{init}) =$ "F". Let's first assume that $i \neq 1$ and $V_d^i(l_d, ds_{init}) =$ "T". Then the proof for this case is as follows:

assumption	$V_d^i(l_d, ds_{init}) = \text{"}T\text{"}$	(1)
assumption	$i \neq 1$	(2)
Premise 10.1	$\forall j,\ j \in N^{+},\ j < i\ :\ V_d^j(l_d, ds_{init}) = \text{"}T\text{"}$	(3)
Sub-Lemma 10.2	$\forall j,\ j \in N^{+},\ j < i\ :\ V_b^j(CR_b, bs_{final})\ \equiv\ V_d^j(CR_d, ds_{final})$	(4)
substituting $i-1$ for j	$V_b^{i-1}(CR_b, bs_{final})\ \equiv\ V_d^{i-1}(CR_d, ds_{final})$	(5)
Premise 10.2	$V_b^i(CR_b, bs_{init})\ \equiv\ V_d^i(CR_d, ds_{init})$	(5)
$l_d \in CR_d$	$V_d^i(l_d, ds_{init}) = V_b^i(l_b, bs_{init})$	(6)
	$Q.E.D.$	(7)

In the final case $i \neq 1$ and $V_d^i(l_d, ds_{init}) =$ "F". In this case, if $V_d^1(l_d, ds_{init}) =$ "F", then we know that $V_b^1(l_b, bs_{init}) =$ "F" also (at the initial state of the first iteration, all the registers including l_d have correct values), and based on the Premise 10.4 we have:

$$\forall j > 1 \;:\; V_d^j(l_d, ds_{init}) = \text{“}F\text{”} \;\wedge\; V_b^j(l_b, bs_{init}) = \text{“}F\text{”}$$

and since $i > 1$, i may be substituted in the above formula for j:

$$V_d^i(l_d, ds_{init}) = \text{“}F\text{”} \;\wedge\; V_b^i(l_b, bs_{init}) = \text{“}F\text{”}$$

However, if $V_d^1(l_d, ds_{init}) = \text{“}T\text{”}$ but $V_d^i(l_d, ds_{init}) = \text{“}F\text{”}$ (the loop is executed at least once), then at some iteration the value of the condition flag l_d should have been changed from ‘true’ to ‘false’, i.e.:

$$\exists m, m \in N^+ \;\wedge\; m \leq i \;:\; V_d^m(l_d, ds_{init}) = \text{“}F\text{”} \;\wedge\; V_d^{m-1}(l_d, ds_{init}) = \text{“}T\text{”}$$

From the above statement and Premise 10.1 it can be concluded:

$$\forall j < m \;:\; V_d^j(l_d, ds_{init}) = \text{“}T\text{”}$$

Then based on Sub-Lemma 10.1:

$$\forall j < m \;:\; V_d^{j+1}(CR_d, ds_{init}) \;\equiv\; V_b^{j+1}(CR_b, bs_{init})$$

$(m-1)$ may be substituted for j in the above formula, that results in:

$$V_d^m(CR_d, ds_{init}) \;\equiv\; V_b^m(CR_b, bs_{init}) \tag{10.1}$$

and since $l_d \in CR_d$, then:

$$V_d^m(l_d, ds_{init}) = V_b^m(l_b, bs_{init}) = \text{“}F\text{”}$$

Then based on the Premise 10.3 the following is true:

$$\begin{aligned}\forall j > m \;:\; & V_d^j(CR_d, ds_{init}) = V_d^m(CR_d, ds_{init}) \;\wedge \\ & V_b^j(CR_b, bs_{init}) = V_b^m(CR_b, bs_{init}, m)\end{aligned}$$

substituting i for j in the above formula:

$$\begin{aligned} V_d^i(CR_d, ds_{init}) &= V_d^m(CR_d, ds_{init}, m) \ \wedge \\ V_b^i(CR_b, bs_{init}) &= V_b^m(CR_b, bs_{init}) \end{aligned}$$

and from the Equation 10.1 we have:

$$V_d^i(CR_d, ds_{init}) \ \equiv \ V_b^i(CR_b, bs_{init})$$

Since $l_d \in CR_d$:

$$V_d^i(l_d, ds_{init}) = V_b^i(l_b, bs_{init})$$

Hence, the proof of Sub-Lemma 10.3 is completed.

Step3 - At this step we prove that under the given assumptions when the loop is exited, the registers hold correct values. This is stated as the Sub-Lemma 10.4 and its proof is given below.

Sub Lemma 10.4

$$\forall i, i \in N^+ : \neg V_d^i(l_d, ds_{init}) \ \Rightarrow \ V_d(CR_d, ds_{exit}) \ \equiv \ V_b(CR_b, bs_{exit})$$

But we know that:

$$\forall i, i \in N^+ : \neg V_d^i(l_d, ds_{init}) \ \Rightarrow \ V_d(CR_d, ds_{exit}) = V_d^i(CR_d, ds_{init})$$

$$\forall i, i \in N^+ : \neg V_b^i(l_b, bs_{init}) \ \Rightarrow \ V_b(CR_b, bs_{exit}) = V_b^i(CR_b, bs_{init})$$

Also from the proof of the previous lemma we know that:

$$\neg V_d^i(l_d, ds_{init}) \ \Rightarrow \ \neg V_b^i(l_b, bs_{init})$$

Then:

$$\begin{aligned} \forall i, i \in N^+ : \neg V_d^i(l_d, ds_{init}) \ \Rightarrow \ (V_d(CR_d, ds_{exit}) &= V_d^i(CR_d, ds_{init}) \ \wedge \\ V_b(CR_b, bs_{exit}) &= V_b^i(CR_b, bs_{init})) \end{aligned}$$

So, to prove the Sub-Lemma 10.4, it is enough to show that:

$$\forall i, i \in N^+ : \neg V_d^i(l_d, ds_{init}) \;\Rightarrow\; V_d^i(CR_d, ds_{init}) \;\equiv\; V_b^i(CR_b, bs_{init})$$

The proof is very similar to the last part of the proof of the previous Sub-Lemma. Let's first consider the case when the loop does not execute at all, and the loop condition flag is false at the initial state of the first iteration of the loop. In this case:

$$V_d^1(l_d, ds_{init}) = \text{``}F\text{''}$$

but we know that:

$$V_d^1(CR_d, ds_{init}) \;\equiv\; V_b^1(CR_b, bs_{init}) \tag{10.2}$$

and since $l_d \in CR_d$, then:

$$V_d^1(l_d, ds_{init}) = V_b^1(l_b, bs_{init}) = \text{``}F\text{''}$$

Then based on the Premise 10.3 the following is true:

$$\begin{aligned}\forall j > 1 \;:\; & V_d^j(CR_d, ds_{init}) = V_d^1(CR_d, ds_{init}) \;\wedge \\ & V_b^j(CR_b, bs_{init}) = V_b^1(CR_b, bs_{init})\end{aligned}$$

substituting i for j in the above formula:

$$\begin{aligned}& V_d^i(CR_d, ds_{init}) = V_d^1(CR_d, ds_{init}) \;\wedge \\ & V_b^i(CR_b, bs_{init}) = V_b^1(CR_b, bs_{init})\end{aligned}$$

and from the Equation 10.2 we have:

$$V_d^i(CR_d, ds_{init}) \;\equiv\; V_b^i(CR_b, bs_{init})$$

This means that:

$$(\forall i \in N^+ \;:\; \neg V_d^1(l_d, ds_{init}) \;\wedge\; \neg V_d^i(l_d, ds_{init}) \;\Rightarrow\; V_d^i(CR_d, ds_{init}) \;\equiv\; V_b^i(CR_b, bs_{init}))$$

However, if $V_d^1(l_d, ds_{init}) =$ "T" but $V_d^i(l_d, ds_{init}) =$ "F" (the loop is executed at least once), then at some iteration the value of the condition flag l_d should have been changed from 'true' to 'false', i.e.:

$$\exists m, m \in N^+ \ \wedge \ m \leq i \ : \ V_d^m(l_d, ds_{init}, m) = \text{"}F\text{"} \ \wedge \ V_d^{m-1}(l_d, ds_{init}) = \text{"}T\text{"}$$

From the above statement and Premise 10.1 it can be concluded:

$$\forall j < m \ : \ V_d^j(l_d, ds_{init}) = \text{"}T\text{"}$$

Then based on Sub-Lemma 10.1:

$$\forall j < m \ : \ V_d^{j+1}(CR_d, ds_{init}) \ \equiv \ V_b^{j+1}(CR_b, bs_{init})$$

$(m-1)$ may be substituted for j in the above formula, that results in:

$$V_d^m(CR_d, ds_{init}) \ \equiv \ V_b^m(CR_b, bs_{init}) \tag{10.3}$$

and since $l_d \in CR_d$, then:

$$V_d^m(l_d, ds_{init}) = V_b^m(l_b, bs_{init}) = \text{"}F\text{"}$$

Then based on the Premise 10.3 the following is true:

$$\begin{aligned}\forall j > m \ : \ & V_d^j(CR_d, ds_{init}) = V_d^m(CR_d, ds_{init}) \ \wedge \\ & V_b^j(CR_b, bs_{init}) = V_b^m(CR_b, bs_{init})\end{aligned}$$

substituting i for j in the above formula:

$$\begin{aligned}& V_d^i(CR_d, ds_{init}) = V_d^m(CR_d, ds_{init}) \ \wedge \\ & V_b^i CR_b, bs_{init}) = V_b^m(CR_b, bs_{init})\end{aligned}$$

and from the Equation 10.3 we have:

$$V_d^i(CR_d, ds_{init}) \;\equiv\; V_b^i(CR_b, bs_{init})$$

This means that:

$$(\forall i \in N^+ \;:\; V_d^1(l_d, ds_{init}, 1) \;\wedge\; \neg V_d^i(l_d, ds_{init}) \;\Rightarrow\; V_d^i(CR_d, ds_{init}) \;\equiv\; V_b^i(CR_b, bs_{init}))$$

This completes the proof of the last sub-lemma. The proof of these three sub-lemmas collectively is a proof of the main lemma. This means that the non-pipelined non-hierarchical loops may be verified by one single pass through the body of the loop.

With a similar reasoning the correctness of an arbitrary non-pipeline iterative block (hierarchical or non-hierarchical) may be verified, if under the same assumptions as before we can show that in every iteration of the loop, if the critical registers have correct values at the initial state of the iteration, then at the end of the iteration they have correct values, too. Since in a hierarchical loop there is no control flow symmetry among the iterations, in this case it is not enough to prove this property for an arbitrary iteration, but in fact it has to be proven for each iteration of the loop:

$$(\forall i \in N^+ \;:\; V_d^i(CR_d, ds_{init}) = V_b^i(CR_b, bs_{init}) \;\Rightarrow\; V_d^i(CR_d, ds_{final}) = V_b^i(CR_b, bs_{final}))$$

However, if each basic block of the RTL design correctly implement a basic block of the behavior specification, it is straight forward to show that for an arbitrary non-pipelined iterative block the above property holds. This may be done in a bottom up fashion. An iterative block at each level of hierarchy is a sequence of synthesis blocks at the lower level. The iterative blocks at the lowest level of hierarchy are basic blocks. At the second lowest level of hierarchy, each iterative block is composed of a sequence of basic blocks. At this level, if each basic block is correctly implemented, then it is obvious that if the registers have correct values at the initial state of each iteration, they will have correct values at the final state of that iteration, too. This reasoning may be recursively applied to the iterative blocks at higher levels of hierarchy. Therefore, by one pass through the control flow graph of each iterative block its correctness may be verified. In the next section we will present a discussion on verification of pipelined non-hierarchical iterative blocks.

10.1.1 Pipelined Loops

To optimize the execution delay of a loop a technique named *loop winding* or *loop folding* is used. During the scheduling of a loop, it is usually assumed that the execution of one iteration should start only when the execution of the previous iteration terminates. If pipelining is performed during the scheduling of a loop, the execution of one iteration of the loop partially overlaps the execution of previous and/or next iterations [17]. The implementation of a loop with a fixed number of iterations n_i, has a total delay of $\lambda \times n_i$. If the same loop has a pipelined implementation with a data introduction interval of δ_0, then its total delay is $\lambda + (n_i - 1) \times \delta_0$, where $\delta_0 < \lambda$. Note that a non-pipelined loop is the special case where $\delta_0 = \lambda$.

Consider Figure 10.3 showing an iterative basic block. This block has been divided into six sub-blocks that are executed consecutively. Each sub-block corresponds to a subset of operations in the body of the loop, and the order of the loop operations is maintained within the sub-blocks.

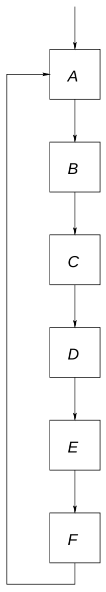


Figure 10.3: **An iterative basic block**

Let's assume the following data-dependencies between the operations of the loop: $\{A_i \mapsto C_i,\ B_i \mapsto D_i,\ D_{i-1} \mapsto C_i,\ E_{i-1} \mapsto E_i\}$. These dependencies are shown in Figure 10.4. A directed arrow from a sub-block b_i to a sub-block b_j indicates that the operations in b_j consume the results of the operations in b_i, or that b_j is data-dependent on b_i. Also, for simplicity of discussion, we assume that each block represents the operations in a single statement of the behavior specification.

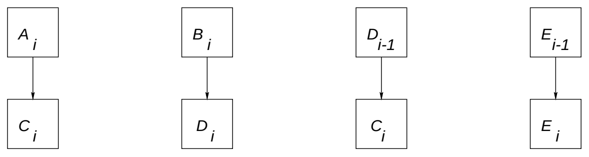


Figure 10.4: **Sub-block dependencies**

During the scheduling of the loop, the execution delay of the loop body is optimized in two ways: (1) The operations that are not data-dependent on each other may be scheduled to execute concurrently. This will improve the execution delay of each iteration of the loop λ, and consequently, the overall execution delay of the loop λ_l, is improved. For example, in the loop of Figure 10.3 there are no data-dependencies between the sub-blocks A and B and between the sub-blocks D and E, therefore, A may be executed in parallel to B and D in parallel to E. (2) As much as data-dependencies permit, the execution of consecutive iterations of the loop may be scheduled with partial overlaps, i.e. an iteration starts execution before the execution of its previous iteration terminates. For example, the execution of a new iteration of the loop in Figure 10.3 may be initiated at the same time as the third sub-block (sub-block C) of the previous iteration, without violating any of the above mentioned data-dependencies. Even though in this optimization scheme, the execution delay of each iteration of the loop stays the same, the overall execution delay of the loop may significantly improve. This latter scheme for performance optimization of a loop introduces the idea behind the loop winding technique or pipelining the implementation of a loop. Figure 10.5 shows a possible pipelined implementation of the loop of Figure 10.3.

In a pipelined loop, the loop initiation interval δ_l is defined as the number of cycles between the initiations of two consecutive iterations of the pipelined loop. For a loop with execution delay λ, the maximum number of stages in the pipeline is defined as $n_s = \lceil \lambda/\delta_l \rceil$. This number is also the maximum number of iterations that may execute concurrently.

When the implementation of a loop is pipelined, the control flow symmetry among different iterations of the loop no longer exists. Each pipelined loop has three phases of operation (Figure 10.6): *prologue* or the cycles between the time the loop starts execution and the time the pipeline is filled; *steady phase* or the cycles at which the pipeline is operating in full capacity (n_s iterations of the loop execute concurrently); and *epilogue* or the cycles between the time the new iterations stop entering the pipeline and the cycle when the last operation of the final iteration of the loop completes its execution (the pipeline is flushed). There is no symmetry between two iterations starting at different phases. Also, there is no symmetry between any two iterations starting at the

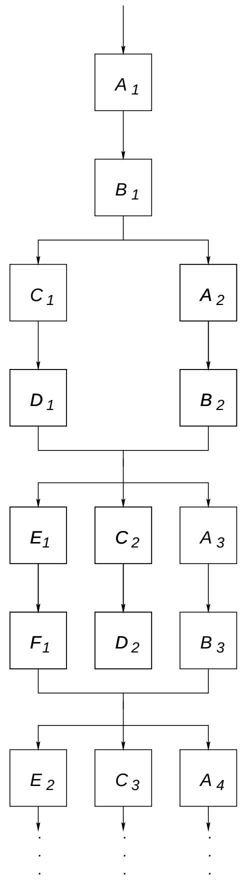


Figure 10.5: **A possible pipelined version of the loop of Figure 10.3**

prologue phase or ending at the epilogue phase.

Due to lack of operational symmetry, that is the basis of our one pass verification approach of the loops, this method is not applicable to the verification of pipelined loops, where different operations are performed in different iterations of a loop. One method of verification of a pipelined loop is to analyze the cycle to cycle behavior of the loop from start to finish to verify its correctness. However, this is only possible for the loops in which the number of iterations is known before the execution of the loop (e.g. *for* loops). Verification of the general loops (such as *while* loops with unknown number of iterations in which the loop variable is updated at each iteration) is not possible through these methods. Even in the cases where the pipelined loops can be verified, this method is inefficient. It is reasonable to demand more inclusive and efficient solutions for the problem of verification of pipelined loops.

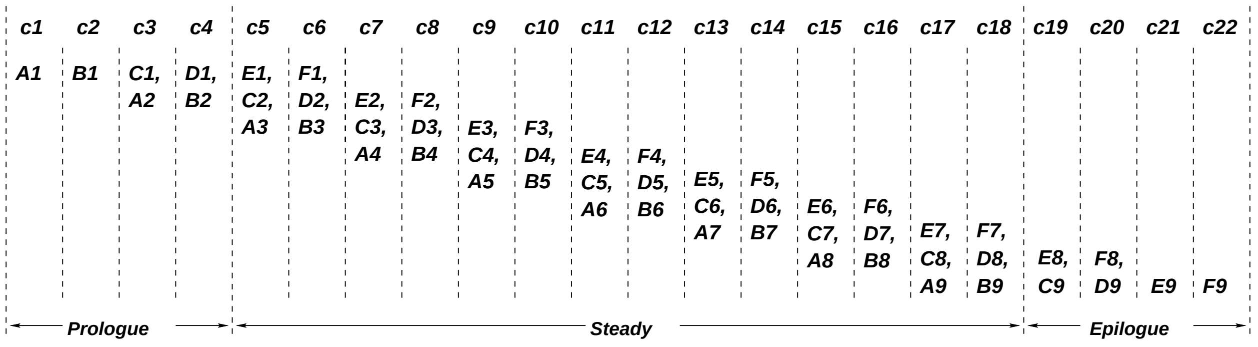


Figure 10.6: **A pipelined implementation of the loop**

10.1.2 Verification of Pipelined Loops

As it was mentioned in the previous section, in a pipelined loop the execution of two consecutive iterations are scheduled δ_l cycles apart. The loop goes through three phases during its execution, and n_l iterations of the loop are executed throughout these phases. The three phases of a pipelined loop are:

1. Prologue - The iterations 1 to $n_s - 1$ start at this phase (cycles c_1 to c_4 in Figure 10.6). The first iteration starts at the first cycle and a new iteration starts at every δ_l cycles then after. For a loop with the execution delay of λ, the loop initiation interval δ_l, and number of stages n_s, the prologue phase consists of the first $\lambda - \delta_l$ first cycles of the execution of the loop. There is no operational symmetry between any two iterations starting at this phase. When the first iteration starts, there is no concurrency. After δ_l cycles, when the second iteration starts, two iterations execute concurrently. After $2 \times \delta_l$ cycles when the third iteration starts, three iterations execute concurrently. This continues until after $(n_s - 1) \times \delta_l$ cycles from the first cycle, when n_s iterations execute concurrently, at which time the prologue phase terminates and the steady state or pipeline phase starts. There is no operational symmetry between any pair of the first $n_s - 1$ iterations. Therefore, the correctness of the operations in each iteration starting at this phase should be individually verified.

2. Steady Phase - The Steady state of a pipelined loop starts $(n_s - 1) \times \delta_l$ cycles after the first cycle, and continues until $(n_s - 1) \times \delta_l$ cycles before the final cycle (cycles c_5 to c_{18} in Figure 10.6). A new iteration starts execution every δ_l cycles. In this phase the maximum number of iterations (n_s) execute concurrently. There is operational symmetry between any two iterations that both start and complete their execution at this phase. Therefore, the verification of the behavior of the loop at this phase is similar to verification of the non-pipelined loops. To fully verify the operation of a loop at its steady state, it is enough to verify an arbitrary iteration that both starts and terminates at this phase. Due to the operational symmetry the control flow of the operation of the loop at this phase may be captured by the control flow graph of an arbitrary iteration of the loop at this phase. Besides, since in the steady state the loop goes through only δ_l distinct states, an iterative block of δ_l distinct states and $n_l - (n_s - 1)$ iterations (Figure 10.9). At each state of either control flow graph up to n_s sets of operations (one set of operations for each of the iterations executing concurrently) may be executed in parallel. Figure 10.7 shows the δ_l distinct states of the control graph of the steady phase for out example pipelined loop.

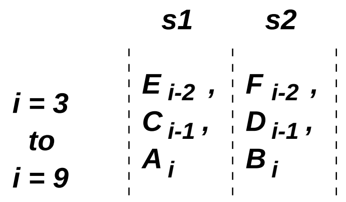


Figure 10.7: **Control flow graph of the pipelined loop at the steady state**

3. Epilogue - This phase corresponds to the time when the pipeline starts flushing, i.e. the last $\lambda - \delta_l$ cycles of the execution of the loop (cycles c_{19} to c_{22} in Figure 10.6). At this phase, no new iterations are initiated and only the execution of the last $n_s - 1$ final iterations of the loop, initiating in the steady phase continue to completion. There are $n_s - 1$ iterations of the loop that start in the steady state and terminate in this phase. There is no operational symmetry between any pair of iterations from these last $n_s - 1$ iterations. At the last δ_l cycles of the loop only the final iteration of the loop executes. In the period from $2 \times \delta_l$ to δ_l cycles before the last, the final 2 iterations of the loop execute, and so on. Due to lack of operational symmetry, to verify the functional correctness of the pipelined loop at the epilogue phase, the correctness of each of the $n_s - 1$ final iterations terminating at this phase should be individually verified.

Now, we will prove that a pipelined loop can be fully verified in $2 \times n_s - 1$ passes through its body, that is one pass for each iteration with a distinctive control flow graph. The distinctive iterations of a pipelined loop comprise of the $n_s - 1$ iterations starting in prologue phase, an arbitrary iteration both starting and ending at the steady phase, and $n_s - 1$ iterations terminating at the epilogue phase. Figure 10.8 shows the 5 distinctive iterations of our example pipelined loop. Considering the fact that pipelining is a tradeoff between the throughput and area (resources), it only make sense to pipeline a loop when n_l, the number of iterations of the loop, is relatively large compared

		c1	c2	c3	c4	c5	c6
Prologue:	*i = 1*	*A1*	*B1*	*C1, A2*	*D1, B2*	*E1, C2, A3*	*F1, D2, B3*
	i = 2	*C1, A2*	*D1, B2,*	*E1, C2, A3*	*F1, D2, B3*	*E2, C3, A4*	*F2, D3, B4*

		c1	c2	c3	c4	c5	c6
Steady State:	*i = 3 to i = 7*	E_{i-2}, C_{i-1}, A_i	F_{i-2}, D_{i-1}, B_i	E_{i-1}, C_i, A_{i+1}	F_{i-1}, D_i, B_{i+1}	E_i, A_{i+1}, C_{i+2}	F_i, B_{i+1}, D_{i+2}

		c1	c2	c3	c4	c5	c6
Epilogue:	*i = 8*	*C7, A8*	*D7, B8,*	*E7, C8, A9*	*F7, D8, B9*	*E8, C9*	*F8 D9*
	i = 9	*C8, A9*	*D8, B9,*	*E8, C9*	*F8, D9*	*E9*	*F9*

Figure 10.8: **All the iterations of the pipelined loop with distinctive CFGs**

to the number of additional sets of resources ($n_l \gg n_s$). In such a case verification in $2 \times n_s - 1$ passes is considerably more efficient compared to a cycle by cycle verification approach (an effort equivalent to n_l/n_s passes through the body of the loop).

When verifying the implementation of a loop, we need to show that at the end of each iteration of the loop, the registers hold correct values, under the condition (assumption) that they have correct values at the initial state of the initial iteration of the loop:

$$(V_b^1(CR_b, bs_{init}) \;\equiv\; V_d^1(CR_d, ds_{init})) \;\Rightarrow$$
$$(\forall\, i,\;\; 1 \leq i \leq n_l \;\;:\;\; V_b^i(CR_b, bs_{final}) \;\equiv\; V_d^i(CR_d, ds_{final}))$$

In previous section, we introduced a strategy for verification of non-pipelined loops, under the given assumption, and based on the above definition of the correctness for the loops. We proved that based on the definition of correctness, to verify a non-pipelined loop, it is enough to show that if at

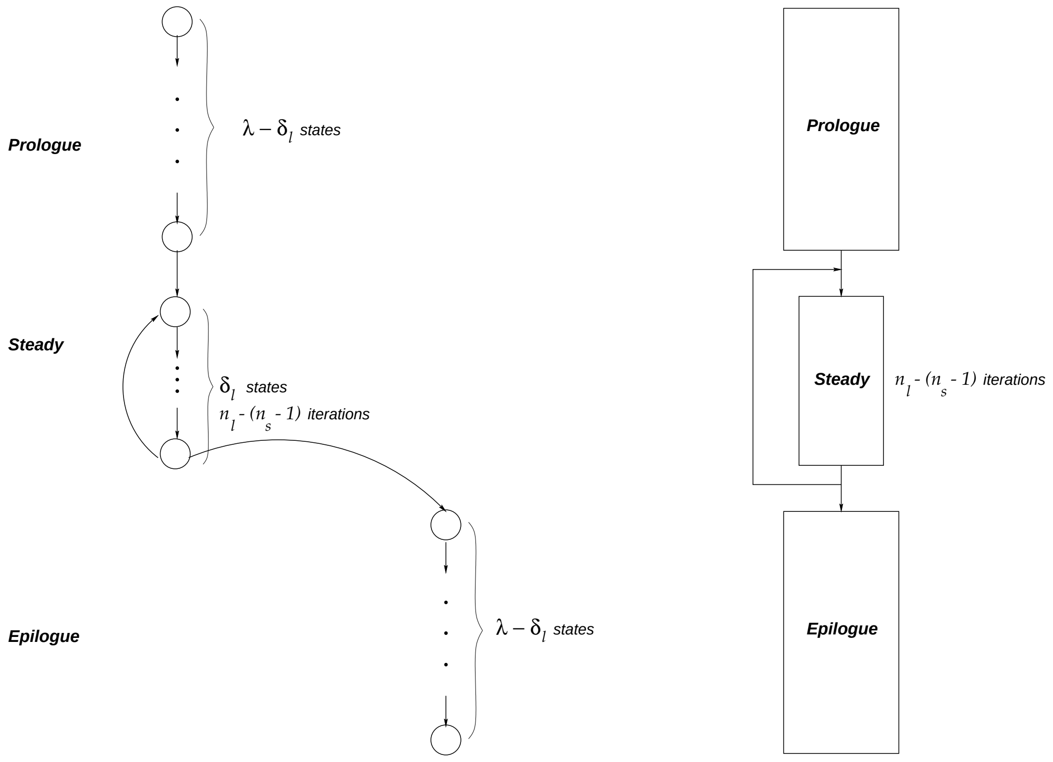

Prologue
$\lambda - \delta_l$ states
Steady
δ_l states
n_l - (n_s - 1) iterations
Epilogue
$\lambda - \delta_l$ states
Prologue
Steady
n_l - (n_s - 1) iterations
Epilogue

Figure 10.9: **The control graph of a pipelined loop**

the initial state of each iteration the registers have correct values, then at the end of the iteration they have correct values too:

$$
\begin{aligned}
&(V_b^1(CR_b, bs_{init}) \;\equiv\; V_d^1(CR_d, ds_{init})) \;\Rightarrow \\
&\qquad (\forall\, i,\;\; 1 \le i \le n_l \;\; : \\
&\qquad\qquad V_b^i(CR_b, bs_{init}) \;\equiv\; V_d^i(CR_d, ds_{init}) \;\Rightarrow\; V_b^i(CR_b, bs_{final}) \;\equiv\; V_d^i(CR_d, ds_{final}))
\end{aligned}
$$

This new definition of the correctness of the loops was the basis of our strategy for verification of non-pipelined loops. During the verification process, we try to prove that at the final states of the specification and implementation loops the content of each register is the same as the value of its specification variable, under the assumption that each register has the same value as its specification variable at the initial states of the loops. Given a behavioral specification of the loop, the value of each specification variable at the final state of each iteration may be symbolically calculated in terms of the values of the specification variables at the initial state of the loop. In a non-pipelined implementation of the loop, the content of each critical register at the final state of each iteration may be evaluated in terms of the values of the critical implementation registers at the initial state of that iteration. Then a comparison between these values can be made, and by exploiting the assumption on the equality of the initial values, a conclusion about the correctness of the final values, and consequently, the correctness of implementation of the loop can be drawn. This is not possible in the case of a pipelined loop.

Let's assume it is possible to extract an n_s stage pipelined implementation of the loop. The control graph of a pipelined loop is partitioned into blocks with δ_l cycles, each corresponding to one stage of the pipeline. An iteration of the loop consists of n_s stages and therefore, is spread over n_s such partitions (Figure 10.10). Two consecutive iterations are spread over $n_s + 1$ partitions (when the steady phase begins, one iteration terminates at the end of each partition and two consecutive iterations share $(n_s - 1)$ of the partitions). The value of a register at the final cycle of the partition $i + (n_s - 1)$ (at the final stage of the i-th iteration of the pipelined loop) corresponds to the value of its specification variable at the final state of iteration i of the specification loop, and its value at a similar cycle of the next partition corresponds to the value of its specification variable at the end of iteration $i + 1$ of the specification loop. Based on the definition of the correctness of the loop, it is the validity of these final values of the registers that at the end of each iteration of the loop should be verified (with respect to the final values of the specification variables).

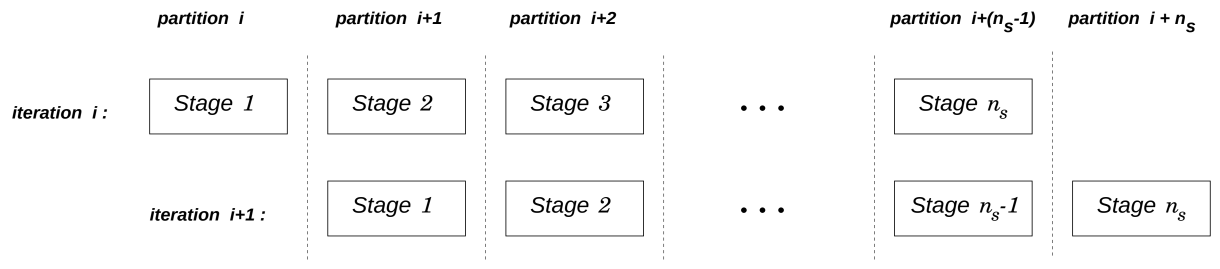


Figure 10.10: **The life span of two consecutive iterations of a pipelined loop**

In each iteration of the loop a set of variables (registers) $AR_b \subseteq CR_b$ ($AR_d \subseteq CR_d$) assume new values. These variables (registers) are referred to as *active variables of the loop* (*active registers*). Active variables are those critical variables that are the target of at least one assignment at each iteration of the loop. Active registers are those critical registers that are bound to an active variable of the loop. Consider the specification variable $r_b \in AR_b$, that assumes a new value in every iteration of a non-hierarchical loop. In a non-pipelined non-hierarchical implementation of the loop, iterations of the loop execute one at a time and throughout the execution of the loop exactly one register $r'_d \in AR_d$ corresponds to r_b. In a pipelined implementation of the same loop on the other hand, each iteration executes in parallel with up to $n_s - 1$ other iterations. This means that the pipelined design should accommodate simultaneous storage of up to n_s different values of r_b (one corresponding to each concurrently executing iteration if necessary). [2] Therefore, in place of register $r'_d \in AR_d$ of a non-pipelined design, there is a queue of up to n_s registers $r_d[1]$ to $r_d[n_s]$ in a pipelined design. During the execution of each iteration of the loop, each register $r_d[j]$ $(1 \leq j \leq n_s)$ holds the value of r'_d at one stage of the pipeline, and $r_d = r_d[n_s]$ holds the value of r'_d at the final stage of the pipeline, i.e. $r_d[1]$ corresponds to the value of r'_d at the first stage of the iteration, $r_d[2]$ corresponds to the value of r'_d at the second stage of the iteration, $\ldots$, and $r_d = r_d[n_s]$ corresponds to the value of r'_d at the final (n_s-th) stage of the iteration. Of all the registers $r_d[1]$ to $r_d[n_s]$, only $r_d = r_d[n_s]$ is a critical register and the rest of the registers are temporaries. During all stages of the life span of an iteration of a pipelined loop but the last, each active critical register may hold a value corresponding to the final stage of some previous iteration of the loop. It is apparent from this discussion that at any given iteration of the pipelined loop, it is only at the final stage that the contents of critical registers of the design are representative of the values of the critical specification variables at that iteration. So, it is only at the final stage of an iteration of a pipelined loop that the values held by critical registers of the design are valid and may be compared to the values of the specification variables at that iteration.

In a pipelined implementation of the loop, at the initial state of each iteration, the execution of the previous iteration is not completed yet, and some of critical registers do not hold valid values. Therefore, unlike the case of the non-pipelined implementations, we cannot assume that at the beginning of each iteration, the critical registers hold correct values; in the contrary, we know that they don't. Also, the contents of the critical registers at the final state of an iteration are defined by the contents of the critical registers and/or temporary registers at the initial state of the loop. It is usually not possible to evaluate the contents of the critical registers at the final state of an iteration solely in terms of the contents of critical registers at the initial state of an iteration. Since the strategy we adopted for verification of the non-pipelined loops is based on calculating the contents of critical registers at the final state of an iteration in terms of their contents at the initial state of that iteration. Apparently, this strategy cannot be used for verification of pipelined designs.

The pipelined loop implementations are verified through a new strategy that is based on a slightly different, but equivalent formulation of correctness. We explained in previous sections that the only valid reasoning about the value of a critical register in a particular iteration of a pipelined implementation of a loop is limited to this value at the final stage of that iteration. Therefore, we should construct a formulation of the correctness that is solely based on the contents of the registers at the final state of each iteration. Let's consider the definition of correctness one more

[2] A more detailed discussion of the exact number of these temporary registers will be presented later in this chapter.

time:

$$(V_b^1(CR_b, bs_{init}) \ \equiv \ V_d^1(CR_d, ds_{init})) \ \Rightarrow$$
$$(\forall\, i, \ 1 \leq i \leq n_l \ : \ V_b^i(CR_b, bs_{final}) \ \equiv \ V_d^i(CR_d, ds_{final}))$$

Since it is the assumption that in a pipelined implementation of the loop, the antecedent of the above proposition or $(V_b^1(CR_b, bs_{init}) \ \equiv \ V_d^1(CR_d, ds_{init}))$ is true, then to prove the correctness of the loop we need to show that under this assumption the following proposition is true:

$$(\forall\, i, \ 1 \leq i \leq n_l \ : \ V_b^i(CR_b, bs_{final}) \ \overset{B_r}{\equiv} \ V_d^i(CR_d, ds_{final}))$$

It can be shown by induction on iteration count of the loop i, that the above proposition is equivalent to the conjunction of the following two propositions.

(1) $V_b^1(CR_b, bs_{final}) \equiv \ V_d^1(CR_d, ds_{final})$
(2) $\forall\, i, \ 1 \leq i \leq n_l - 1 :$
$$V_b^i(CR_b, bs_{final}) \equiv \ V_d^i(CR_d, ds_{final}) \quad \Rightarrow \quad V_b^{i+1}(CR_b, bs_{final}) \ \equiv \ V_d^{i+1}(CR_d, ds_{final})$$

Then to prove the correctness of the pipelined implementation of a loop, it is enough to prove the validity of the above propositions for that particular loop. This new formulation of correctness is the basis of our strategy for verification of the pipelined loop.

The proof of the first proposition is straight forward. Since we know that $(V_b^1(CR_b, bs_{init}) \ \equiv \ V_d^1(CR_d, ds_{init}))$ is true, then to prove the first proposition we show that:

$$(V_b^1(CR_b, bs_{init}) \ \equiv \ V_d^1(CR_d, ds_{init})) \ \Rightarrow \ (V_b^1(CR_b, bs_{final}) \ \equiv \ V_d^1(CR_d, ds_{final}))$$

At the first iteration, the contents of the temporaries are loaded with the initial values of the critical registers (the contents of the critical registers at the initial state of the first iteration of the loop). Then, at the first iteration the contents of the temporaries may be calculated in terms of the contents of critical registers at the initial state. At the final state of the first iteration, the contents of the critical registers may be calculated in terms of the contents of the critical registers and temporaries. This means that at the final state of the first iteration, the contents of the critical registers may be calculated solely in terms of the contents of critical registers at the initial state.

Now, a comparison between the values of the specification variables and critical registers at the final states can be made (exactly in the same way as in the case of a non-pipelined implementation of the loop).

The proof of the second proposition is done in three steps. The second proposition is equivalent to the conjunction of three other propositions. At each step of the proof the correctness of one of these propositions is verified:

$$(2-a)\quad \forall\, i,\ 1 \le i < n_s : \quad V_b^i(CR_b, bs_{final}) \;\equiv\; V_d^i(CR_d, ds_{final}) \;\Rightarrow\; V_b^{i+1}(CR_b, bs_{final}) \;\equiv\; V_d^{i+1}(CR_d, ds_{final})$$

$$(2-b)\quad \forall\, i,\ n_s \le i \le n_l - n_s : \quad V_b^i(CR_b, bs_{final}) \equiv V_d^i(CR_d, ds_{final}) \;\Rightarrow\; V_b^{i+1}(CR_b, bs_{final}) \equiv V_d^{i+1}(CR_d, ds_{final})$$

$$(2-c)\quad \forall\, i,\ n_l - n_s < i \le n_l - 1 : \quad V_b^i(CR_b, bs_{final}) \;\equiv\; V_d^i(CR_d, ds_{final}) \;\Rightarrow\; V_b^{i+1}(CR_b, bs_{final}) \;\equiv\; V_d^{i+1}(CR_d, ds_{final})$$

The construction of the proof in all the three cases is done using the same tactics. Then the explanation below applies to construction of the proof in each of these cases. Consider an arbitrary iteration i of a pipelined loop, and its consecutive iteration $i+1$. The lifetime of these two iterations together spans over (n_s+1) out of the $n_l+(n_s-1)$ stages of the lifetime of the loop (Figure 10.10). To prove the correctness conditions of the loop, it should be established through mathematical proof that if the contents of the critical registers at the final state of the iteration i have correct values, then at the final state of iteration $i+1$ they have correct values too. Let's assume that the contents of the critical registers at the final state of iteration i are correct:

$$\forall r_b \in CR_b,\ \forall r_d \in CR_d \;:\; r_d = B_r(r_b) \;\Rightarrow\; (V_b^i(r_b, bs_{final}) \;\equiv\; V_d^i(r_d, ds_{final})) \tag{10.4}$$

and since we know that $(V_b^i(r_b, bs_{final}) = (V_b^{i+1}(r_b, bs_{init})$, then:

$$\forall r_b \in CR_b,\ \forall r_d \in CR_d \;:\; r_d = B_r(r_b) \;\Rightarrow\; (V_b^{i+1}(r_b, bs_{init}) \equiv V_d^i(r_d, ds_{final}))$$

This means that the contents of the specification variables at the initial state of the iteration $i+1$ may be defined in terms of the contents of critical registers at the final state of the iteration i. Also, we know that the values of the specification variables at the final state of iteration $i+1$ are defined in terms of the values of these variables at the initial state of iteration $i+1$:

$$\exists f \ : \ V_b^{i+1}(CR_b, bs_{final}) \ = \ f(V_b^i(CR_b, bs_{final}))$$

Then, the values of the specification variables at the final state of the iteration $i+1$ may be defined in terms of the contents of critical registers at the final state of iteration i:

$$V_b^{i+1}(CR_b, bs_{final}) \ \equiv \ f(V_d^i(CR_d, ds_{final}))$$

Then to prove the correctness of the loop implementation, we should show that:

$$V_d^{i+1}(CR_d, ds_{final}) \ = \ f(V_d^i(CR_d, ds_{final})) \tag{10.5}$$

To construct this proof the contents of the critical registers at the final state of the iteration $i+1$ should be somehow calculated in terms of the contents of the registers at the final state of the iteration i. The value of a register at the final state of the iteration $i+1$ may be calculated in terms of the values of the non-active critical registers as well as temporaries at the initial state of this iteration. Then the values of the non-active critical registers and temporaries at the beginning of iteration $i+1$ should be calculated in terms of the values of the critical registers at the final state of the iteration $i+1$.

The values of non-active critical registers remain the same from one iteration to the next. Therefore, the values of non-active critical registers at iteration $i+1$ may be calculated in terms of their values at the final state of iteration i:

$$\begin{aligned}
\forall ds, \forall r_d \in CR_b,\ \ r_d \notin AR\ V_d^i(r_d, ds) \ &= V_d^i(r_d, ds_{final}) \\
\forall ds, \forall r_d \in CR_b,\ \ r_d \notin AR\ V_d^{i+1}(r_d, ds) \ &= V_d^i(r_d, ds_{final})
\end{aligned}$$

Now it remains to find a tactic for evaluating the contents of temporaries at the initial state of iteration $i+1$ in terms of the contents of the critical registers at the final state of iteration i. We know that the values of temporary registers at initial state of iteration $i+1$ may be calculated in terms of the values of temporary registers at the initial state of iteration i. Now, the problem may be restated as given $V_d^i(CR_d, ds_{final})$, the values of the critical registers at the final state of iteration i, calculate the contents of the temporary registers from the final state of that iteration back to the initial state of iteration in terms of the contents of critical registers at the final state of iteration i. We know that the opposite is possible and the value of a critical register at the final state of the iteration i, may be calculated in terms of the critical registers and temporaries at the initial state of iteration i. Given the values of the critical registers at the final state of iteration i, the contents of the temporaries may be calculated from the final state of iteration i back to initial state of that iteration through a procedure called *backward value propagation.*

Backward value propagation is used when the values at the output registers of a circuit at a certain state are known and it is desirable to calculate the values at the input registers at that state or at a previous state. So, the backward propagation of values may be in spatial or temporal domain of a design, or both. It is not always possible to calculate the values at the inputs and outputs of all the registers in terms of the values of the output registers. The backward propagation of values is usually most successful when a sequence of registers are chained together. Usually, when the input of a registers is connected to an ALU unit, the backward traversal stops. The backward propagation is done through a procedure that takes five parameters as input: (1) a state s_d when the backward traversal starts, (2) a register r where the backward traversal starts, (3) a value v that appears at the output of the register $r.out$ at state s_d, (4) the set of registers whose output values need to be evaluated, C, and (5) the state when the backward propagation terminates. The Backward value propagation algorithm is given in Figure 10.11.

The critical registers are desirable candidates for backward value propagation. In the case of critical registers (specifically in the case of the active critical registers), we know that there is a chain of temporaries that define the content of an active critical register at a particular state. Then, this value may be easily propagated backward up to the temporary registers. Then the values of the temporary registers at the iteration i may be calculated in terms of the final values of critical registers at this iteration through backward value propagation. These values may then be propagated forward to calculate the values of the temporaries at the iteration $i+1$, and finally to calculate the contents of the active critical registers at the final state of this iteration:

$$\exists g \;:\; V_b^{i+1}(CR_b, bs_{final}) \;=\; g(V_b^i(CR_b, bs_{final}) \tag{10.6}$$

Then comparing the Equations 10.5 and 10.6 the correctness of the contents of critical registers at the final state of iteration $i+1$ may be verified, by investigating that the two functions f and g map each critical register to the same values. As we have mentioned before this process has to be repeated once for each iteration with a distinct control flow graph and one for all the iteration with a symmetrical control flow graphs.

Verification of the pipelined loops through the above mentioned strategy, and in $2 \times n_s - 1$ passes is considerably more efficient than verification through unrolling the specification loop. Intuitively, this new approach may seem the most efficient possible for verification of the pipelined loop. In

```
propagate_back ($r_d$ : $REGISTER$,
                $s_i$ : $RTL\_STATE$,
                $v$ : $VALUE$,
                $R$ : $\{r \mid r : REGISTER\}$,
                $s_f$ : $RTL\_STATE$)
{
    if ($r_d \in R$)
        add_known_values ($r_d.out$, $s_d$, $c$);
    else
        return;
    if ($s_i = s_f$)
        return;

    if ($(\neg V_d\,(r_d.ld,\ pred(s_d))$)
        propagate_back ($r_d$, $pred(s_d)$, $c, R, s_f$);
    else        /*  $V_d\,(r_d.ld,\ pred(s_d)$  */
        if ($(V_d\,(r.in, pred(s_d)) = V_d\,(r'_d.out,\ pred(s_d))$)
            propagate_back ($r'_d$, $pred(s_d)$, $c, R, s_f$);
    return;
}
```

Figure 10.11: **Backward Value Propagation Algorithm**

what follows we present a modification of our method that offers an even better solution to the problem.

10.1.3 Verification of Pipelined Loops : An Alternative Approach

The behavior graph (specification) of a pipelined loop may be transformed into an equivalent graph with n_s partitions, where each partition represents one stage of the pipelined loop (the equivalence of the original behavior graph and the transformed behavior graph may be verified through our observation equivalence checking approach discussed earlier). Now, each partition of the behavior graph may be considered a critical path and the correctness of the pipelined implementation of the loop may be verified through comparing the operation of each behavior partition (critical path) and its corresponding stage of the pipelined loop.

A set of operations are performed at each iteration of the loop. The results of these operations are accumulated in some temporary registers at each stage of the execution of the loop, and before the execution of the final stage (stage n_s) of an iteration terminates, the final results of the operations are assigned to appropriate critical registers of the design.

Definitions

We define the *active variables of stage* j as the set of all the critical specification variables that assume a new value at the stage j of a pipelined loop. AR_b^j denotes the set of active variables of stage j of the loop. The active registers of the stage j of a pipelined loop AR_d^j, are those registers that correspond to an active variable at stage j. In a non-pipelined implementation of a (non-hierarchical) loop, exactly one register represents each specification variable throughout the execution of the loop. As it was explained before, this is not true in the case of a pipelined loop.

Consider the i-th and $(i+1)$-th iterations of the loop as before. Suppose $r'_d \in AR_d^1$ is an active register of the stage 1 in a non-pipelined implementation of the loop. In a pipelined implementation of the loop, a temporary register $t_d[1]$ is introduced to the circuit in place of r'_d. The temporary register $t_d[1]$ represents r'_d at the first stage of iteration i. All the operations performed on r'_d at the first stage of iteration i are performed on $t_d[1]$ in a pipelined implementation of the loop. Since a total of $n_1 = |AR_d^1|$ such registers are active at stage 1, a total of n_1 temporary registers are required in place of the active critical registers of the first stage. These registers are the temporary registers of the first stage and are denoted by $T_d[1]$. At any time during the execution of a pipelined loop, these registers are assigned to the iteration executing at its first stage.

When iteration i of the pipelined loop starts the registers in $T_d[1]$ are assigned to this iteration and hold the results of the operations at the first stage of this iteration. But, when iteration i enters its second stage, the iteration $i+1$ starts its first stage and the registers $T_d[1]$ should be assigned to iteration $i+1$ (Figure 10.12. Therefore, $n_1 = |T_d[1]|$ other temporary registers should exist in the circuit to hold the previously calculated values of $T_d[1]$ corresponding to the first stage

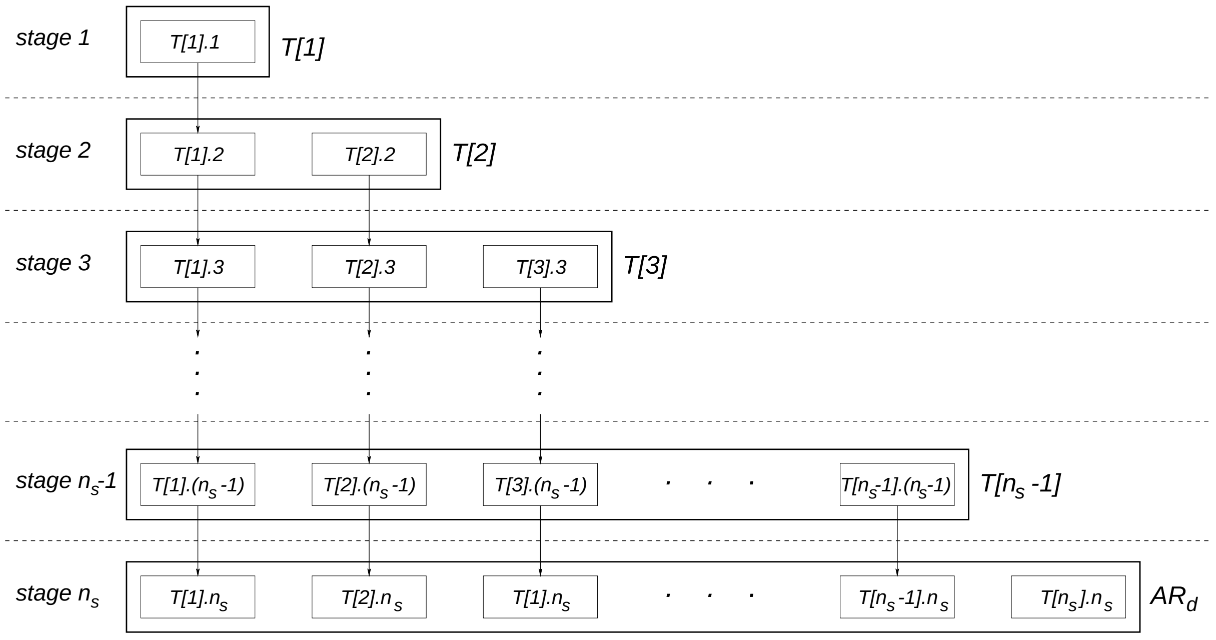


Figure 10.12: **Register assignment in a pipelined implementation of a loop**

of iteration i, as these values may be consumed at the succeeding stages of that iteration. Besides, at stage 2 of iteration i some more operations are performed. At a non-pipelined implementation of the loop, the results of these operations are assigned to the active registers of the second stage, AR_d^2. Then, in the pipelined implementation of the loop, some more temporary registers should be introduced to the circuit to represent the active registers of the second stage, AR_d^2. Therefore, a total of $n_2 = |T_d[1] \cup AR_d^2| = |AR_d^1 \cup AR_d^2|$ temporary registers are assigned to an iteration executing at its second stage of operation. These are the temporary registers of stage 2 and are denoted by $T_d[2]$. With a similar analogy we can show that a total of $n_j = |\cup_{k=1}^{j} AR_d^k|$ temporary registers are assigned to an iteration executing at its j-th stage, where $1 \leq j < n_s$. These are the temporary registers of stage j and are denoted by $T_d[j]$. At the final stage of execution of an iteration, the operations are performed on the critical registers of the design and $T_d[n_s] = AR_d$. Since the non-active critical registers are not the target of an assignment operation, and their values remain the same throughout the execution of the loop, only read operations may be performed on them. Therefore, the contents of these registers may be directly read from the output of these registers and no temporary registers are needed in place of these registers.

Just after the execution of a particular iteration at the stage j terminates, and before that iteration enters its $(j+1)$-th stage of execution, the contents of the temporary registers $T_d[j]$ are written to the appropriate temporary registers of stage $j+1$. This means that at the beginning of stage $j+1$ the following is true:

$$\forall k,\ 1 \leq k \leq n_j \ : \ t_d[j+1][k] \ = \ t_d[j][k]$$

where $(n_j = | \cup_{k=1}^{j} AR_d^k|)$.

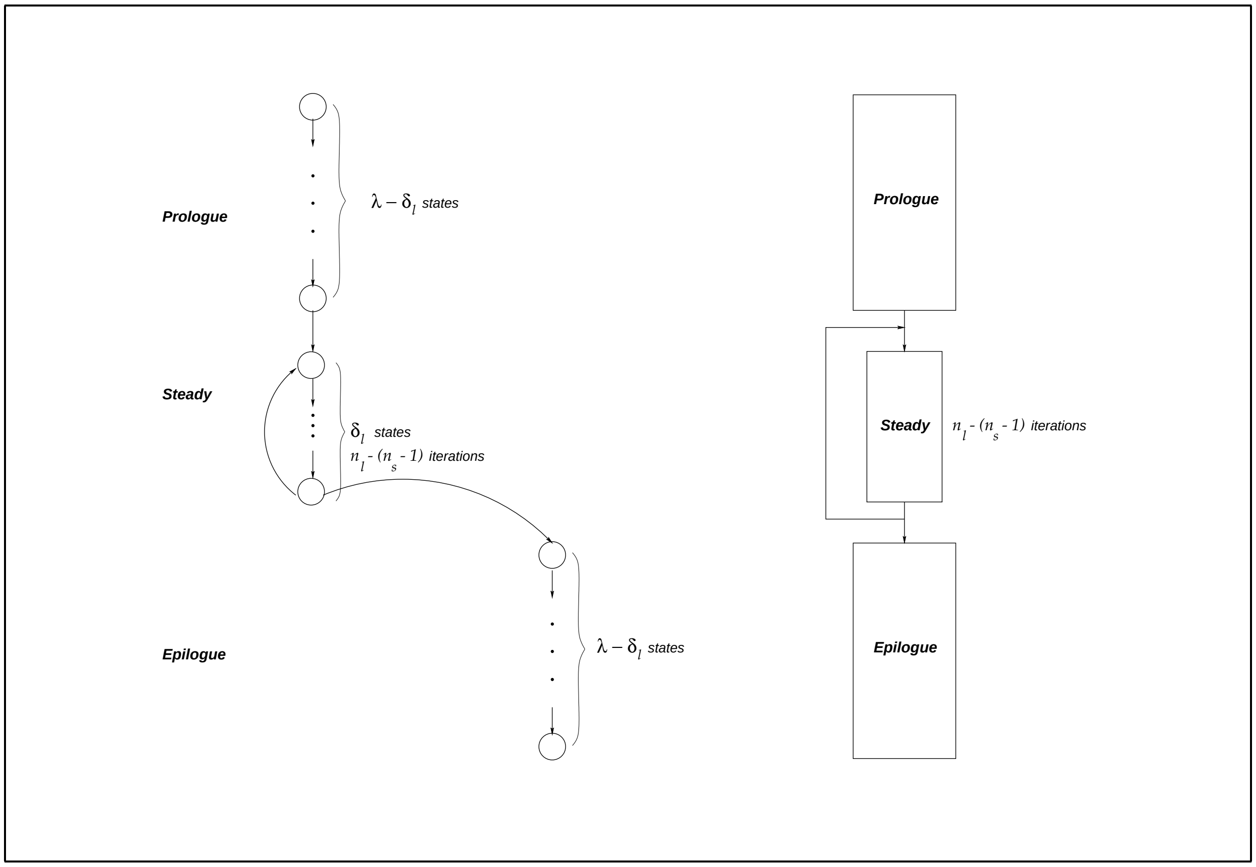

Prologue
$\lambda - \delta_l$ states
Steady
δ_l states
$n_l - (n_s - 1)$ iterations
Epilogue
$\lambda - \delta_l$ states
Prologue
Steady
$n_l - (n_s - 1)$ iterations
Epilogue

Figure 10.13: **The control graph of a pipelined loop revisited**

Verification Method

In this section we will show that the pipelined implementation of the loop may be verified through criticality masking technique and by defining a dynamic register binding function. We first consider the verification of the behavior of the pipelined loop at the steady state. As we mentioned before the behavior graph of the loop-body cp_b is partitioned into n_s subgraphs $cp_b[1]$, $cp_b[2]$, …, $cp_b[n_s]$, so that each subgraph $cp_b[j]$ corresponds to a state j of the pipelined loop. Then, in place of the single critical path of the non-pipelined design, the non-pipelined design has n_s critical paths. We know that for an arbitrary iteration i of the loop that both starts and terminates in the steady phase (i.e. $(n_s - 1) \leq i \leq (n_l - (n_s - 1))$), the verification is performed in n_s steps. In each step, the functional correctness of the pipelined loop at one phase of its operation is verified.

We know that in a pipelined implementation of the loop up to n_s iterations may execute concurrently. Also, we know that the values of the registers at one iteration of the loop, depends on the values of the registers at the previous iteration. That is to say an iteration executing at stage j may consume the contents of some of the temporary registers at stage $j + 1$. If the pipelined loop is correctly implemented, the concurrent execution of n_s iterations should not have any other side-effects. This means that the result of the execution of an iteration i in parallel with up to $n_s - 1$ other iterations should be the same as the execution of the iteration i alone.

Also, if the loop is correctly implemented, at the end of the first stage of the execution of iteration i the temporary registers $T_d[1]$ have correct values, and at the end of the second stage of the execution of iteration i the temporary registers $T_d[2]$ have correct values. With the same analogy at the end of the j-th stage of the execution of iteration i the temporary registers $T_d[j]$ have correct values, where $1 \leq j \leq n_s$. This ensures that at the end of the n_s-th stage of the execution of iteration i the temporary registers $T_d[n_s] = AR_d$ have correct values (Figure 10.14).

The non-active or passive variables (registers) of the loop are denoted by PR_b (PR_d), where $PR_b = CR_b - AR_b$ ($PR_d = CR_d - AR_d$) and we know that:

$$\forall bs,\ \forall i,\ 1 \leq i \leq n_l \ :\ V_b^i(AR_b, bs_{init}) \ \equiv\ V_b^1(AR_b, bs_{init})$$

In our verification approach, at a particular iteration i of the pipelined loop, the criticality of the registers AR_b (AR_d) at all the stages of the pipeline except the last (n_s) is masked. At each stage j of an iteration i, these registers are critical for a concurrently executing iteration $i - (n_s - j)$ (if it exist). The critical variables along each critical path $cp_b[j]$ are $CR_b[j] = \cup_{k=1}^{j} AR_d^k \cup PR_b$, and the critical registers at the stage j of iteration i are $CR_d[j] = T_d[j] \cup PR_d$. This means that for the critical path cp_{n_s} the critical variables are defined as:

$$CR_b[n_s] = \bigcup_{k=1}^{n_s} AR_b^k \cup PR_b = AR_b \cup PR_b = CR_b \tag{10.7}$$

and the critical registers at the stage n_s of iteration i are defined as:

	iteration i	iteration $i+1$	iteration $i+2$		iteration $i+n_s$-3	iteration $i+n_s$-2	iteration $i+n_s$-1
stage 1	T_d [1]						
stage 2	T_d [2]	T_d [1]					
stage 3	T_d [3]	T_d [2]	T_d [1]				
. . .							
stage n_s - 2	T_d [n_s - 2]	T_d [n_s - 3]	T_d [n_s - 4]	. . .	T_d [1]		
stage n_s - 1	T_d [n_s -1]	T_d [n_s - 2]	T_d [n_s - 3]	. . .	T_d [2]	T_d [1]	
stage n_s	AR_d	T_d [n_s -1]	T_d [n_s -2]	. . .	T_d [3]	T_d [2]	T_d [1]

Figure 10.14: **Critical registers at each stage of execution of an iteration** i

$$CR_d[n_s] = T_d[n_s] \cup PR_d = AR_d \cup PR_d = CR_d \tag{10.8}$$

Now we revisit the definition of the correctness once again and give a new interpretation of it, as a conjunction of three correctness conditions. In the previous sections, the correctness of the loop was defined by the following proposition:

$$(V_b^1(CR_b, bs_{init}) \;\equiv\; V_d^1(CR_d, ds_{init})) \;\Rightarrow\; (\forall\, i,\;\; 1 \leq i \leq n_l \;\;:\;\; V_b^i(CR_b, bs_{final}) \;\equiv\; V_d^i(CR_d, ds_{final}))$$

Since it is the assumption that in a pipelined implementation of the loop, the antecedent of the above proposition or $(V_b^1(CR_b, bs_{init}) \;\equiv\; V_d^1(CR_d, ds_{init}))$ is true, then correctness condition of a loop is reduced to the following proposition:

$$(\forall\, i,\ 1 \leq i \leq n_l \ :\ V_b^i(CR_b, bs_{final}) \ \equiv\ V_d^i(CR_d, ds_{final}))$$

In what follows three new correctness conditions for a pipelined loop are presented. We will show that these three correctness conditions collectively imply the above correctness condition of the loop. Each of three correctness conditions assures the functional correctness of the loop at one of the three phases of its operation.

Prologue Correctness Condition

Informally, this condition states that if the critical registers have correct values at the initial state of the first iteration of the loop, then, at the final state of the prologue phase the critical registers CR_d, and temporary registers of this phase $T_d = \cup_{k=1}^{n_s-1} T_d[k]$, have correct values, too:

$$\begin{array}{l}
(V_b^1(CR_b, bs_{init}) \ \equiv\ V_d^1(CR_d, ds_{init})) \ \Rightarrow \\
\qquad\qquad \begin{array}{rcl}
(V_b^1(CR_b[n_s-1], bs_{final}[n_s-1]) & \equiv & V_d^1(CR_d[n_s-1], ds_{final}[n_s-1]) \ \wedge \\
V_b^2(CR_b[n_s-2], bs_{final}[n_s-2]) & \equiv & V_d^1(CR_d[n_s-2], ds_{final}[n_s-1]) \ \wedge \\
\vdots & & \\
V_b^{n_s-2}(CR_b[2], bs_{final}[2]) & \equiv & V_d^1(CR_d[2], ds_{final}[n_s-1]) \ \wedge \\
V_b^{n_s-1}(CR_b[1], bs_{final}[1]) & \equiv & V_d^1(CR_d[1], ds_{final}[n_s-1]) \ \wedge \\
V_b^1(CR_b, bs_{init}) & \equiv & V_d^1(CR_d, ds_{final}))
\end{array}
\end{array}$$

By considering the Equations 10.7 and 10.8 and the assumption that $(V_b^1(CR_b, bs_{init}) \ \equiv\ V_d^1(CR_d, ds_{init}))$, then prologue correctness condition is reduced to:

$$\begin{array}{l}
CC_p \ : \\
\qquad \begin{array}{rcl}
(V_b^1(CR_b[n_s-1], bs_{final}[n_s-1]) & \equiv & V_d^1(CR_d[n_s-1], ds_{final}[n_s-1]) \ \wedge \\
V_b^2(CR_b[n_s-2], bs_{final}[n_s-2]) & \equiv & V_d^1(CR_d[n_s-2], ds_{final}[n_s-1]) \ \wedge \\
\vdots & & \\
V_b^{n_s-2}(CR_b[2], bs_{final}[2]) & \equiv & V_d^1(CR_d[2], ds_{final}[n_s-1]) \ \wedge \\
V_b^{n_s-1}(CR_b[1], bs_{final}[1]) & \equiv & V_d^1(CR_d[1], ds_{final}[n_s-1]) \ \wedge \\
V_b^1(CR_b, bs_{init}) & \equiv & V_d^1(CR_d, ds_{final}))
\end{array}
\end{array}$$

The prologue correctness condition ensures that when the steady phase starts the critical registers of each of each stage have correct values, and also that the contents of critical registers have not changed. That is to say, each iteration i from the first (n_s-1) iterations, up to its (n_s-i)-th stage of operation, is functionally correct.

Steady Correctness Condition

We mentioned before that the behavior graph of the loop may be partitioned into n_s critical paths $cp_b[1]$, $cp_b[2]$, $\ldots$, $cp_b[n_s]$. Each of these critical paths corresponds to on stage of operation of the loop at each of its iterations.

The control flow graph of the steady phase of the pipelined implementation of the loop is also a loop with $n_l-(n_s-1)$ iterations, δ_l distinct states and n_s parallel branches (Figure 10.9). Therefore, at each iteration of the loop of the steady phase n_s stages of the loop are executed concurrently. In other words, at each one of the δ_l states of the control graph of the steady phase n_s different operations, each corresponding to one stage of the loop are executed simultaneously. Each set of the temporary registers $T[j]$ is assigned to a branch j of the steady phase. At the end of each iteration i of the control graph of the steady phase, the iteration i of the pipelined loop is also completed. Since the final state of the iteration i of the loop $ds_{final}[n_s]$ and the final state of the iteration i of the steady phase are the same, we denote both these states by ds_{final}.

The steady correctness condition ensures that at the end of each iteration of the control flow graph of the steady phase the registers have correct values (*stage* n_s in Figure 10.14 corresponds to one iteration of the control flow graph of the steady phase). Since at the end of each iteration of the control flow graph of the steady phase, one iteration of the loop terminates, this ensures that at the end of each iteration terminating at the steady phase, the critical registers have correct values. The steady correctness condition states that at each arbitrary iteration i of the loop, that both starts and end at the steady phase, if the values of the registers at the end of stage $j-1$ of the iteration are correct, then at the end of the next stage (stage j) of the iteration i they are correct, too.

$$\forall i, 1 \leq i \leq n_l-(n_s-1) \ :$$

$$\begin{aligned}
(V_b^{i-1}(CR_b[n_s], bs_{final}[n_s]) \ &\equiv \ V_d^{i-1}(CR_d[n_s], ds_{final}) \ \wedge \\
V_b^{i}(CR_b[n_s-1], bs_{final}[n_s-1]) \ &\equiv \ V_d^{i-1}(CR_d[n_s-1], ds_{final}) \ \wedge \\
V_b^{i+1}(CR_b[n_s-2], bs_{final}[n_s-2]) \ &\equiv \ V_d^{i-1}(CR_d[n_s-2], ds_{final}) \ \wedge \\
&\vdots \\
V_b^{i+(n_s-3)}(CR_b[2], bs_{final}[2]) \ &\equiv \ V_d^{i-1}(CR_d[2], ds_{final}) \ \wedge \\
V_b^{i+(n_s-2)}(CR_b[1], bs_{final}[1]) \ &\equiv \ V_d^{i-1}(CR_d[1], ds_{final}) \ \wedge
\end{aligned}$$

$$
\begin{aligned}
&V_b^1(PR_b, bs_{init}) \;\equiv\; V_d^{i-1}(PR_d, ds_{final})) \;\Rightarrow \\
&\qquad\qquad (V_(^i CR_b[n_s], bs_{final}[n_s]) \;\equiv\; V_d^i(CR_d[n_s], ds_{final}) \;\wedge \\
&\qquad\qquad\; V_b^{i+1}(CR_b[n_s-1], bs_{final}[n_s-1]) \;\equiv\; V_d^i(CR_d[n_s-1], ds_{final}) \;\wedge \\
&\qquad\qquad\; V_b^{i+2}(CR_b[n_s-2], bs_{final}[n_s-2]) \;\equiv\; V_d^i(CR_d[n_s-2], ds_{final}) \;\wedge \\
&\qquad\qquad\qquad\qquad \vdots \\
&\qquad\qquad\; V_b^{i+(n_s-2)}(CR_b[2], bs_{final}[2]) \;\equiv\; V_d^i(CR_d[2], ds_{final}) \;\wedge \\
&\qquad\qquad\; V_b^{i+(n_s-1)}(CR_b[1], bs_{final}[1]) \;\equiv\; V_d^i(CR_d[1], ds_{final}) \;\wedge \\
&\qquad\qquad\; V_b^1(CR_b, bs_{init}) \;\equiv\; V_d^i(CR_d, ds_{final}))
\end{aligned}
$$

that considering Equations 10.7 and 10.8 may be reduced to:

$$
\begin{aligned}
&CC_s \;:\; \forall i, 1 \leq i \leq n_l - (n_s - 1) \;: \\
&(V_b^{i-1}(CR_b, bs_{final}) \;\equiv\; V_d^{i-1}(CR_d, ds_{final}) \;\wedge \\
&\; V_b^i(CR_b[n_s-1], bs_{final}[n_s-1]) \;\equiv\; V_d^{i-1}(CR_d[n_s-1], ds_{final}) \;\wedge \\
&\; V_b^{i+1}(CR_b[n_s-2], bs_{final}[n_s-2]) \;\equiv\; V_d^{i-1}(CR_d[n_s-2], ds_{final}) \;\wedge \\
&\qquad\qquad \vdots \\
&\; V_b^{i+(n_s-3)}(CR_b[2], bs_{final}[2]) \;\equiv\; V_d^{i-1}(CR_d[2], ds_{final}) \;\wedge \\
&\; V_b^{i+(n_s-2)}(CR_b[1], bs_{final}[1]) \;\equiv\; V_d^{i-1}(CR_d[1], ds_{final})) \;\Rightarrow \\
&\qquad\qquad (V_b^i(CR_b, bs_{final}) \;\equiv\; V_d^i(CR_d, ds_{final}) \;\wedge \\
&\qquad\qquad\; V_b^{i+1}(CR_b[n_s-1], bs_{final}[n_s-1]) \;\equiv\; V_d^i(CR_d[n_s-1], ds_{final}) \;\wedge \\
&\qquad\qquad\; V_b^{i+2}(CR_b[n_s-2], bs_{final}[n_s-2]) \;\equiv\; V_d^i(CR_d[n_s-2], ds_{final}) \;\wedge \\
&\qquad\qquad\qquad\qquad \vdots \\
&\qquad\qquad\; V_b^{i+(n_s-2)}(CR_b[2], bs_{final}[2]) \;\equiv\; V_d^i(CR_d[2], ds_{final}) \;\wedge \\
&\qquad\qquad\; V_b^{i+(n_s-1)}(CR_b[1], bs_{final}[1]) \;\equiv\; V_d^i(CR_d[1], ds_{final}))
\end{aligned}
$$

Please note that throughout this discussion $V_b^0(CR_b[n_s], bs_{final}[n_s])$ refers to $V_b^1(CR_b,\, bs_{init})$ and $V_d^0(CR_d[n_s], ds_{final}[n_s])$ refers to $V_d^1(CR_d, ds_{init})$, and these terms are used interchangeably throughout the text to denote the values of the critical variables prior to the execution of the first iteration of the loop.

Epilogue Correctness Condition

Of n_l iterations of the loop, $n_l - (n_s - 1)$ iterations end at the steady phase of execution of the loop. The prologue and steady correctness conditions ensure that at the end of each of these iterations, the critical registers have correct values. The epilogue correctness condition ensures that at the end of each of the remaining $(n_s - 1)$ iteration of the loop, that end at epilogue phase, the critical registers have correct values too:

$$
\begin{array}{l}
(V_b^{n_l-(n_s-1)}(CR_b[n_s], bs_{final}[n_s]) \;\equiv\; V_d^{n_l-(n_s-1)}(CR_d[n_s], ds_{final})) \;\wedge \\
\;V_b^{n_l-(n_s-2)}(CR_b[n_s-1], bs_{final}[n_s-1])) \;\equiv\; V_d^{n_l-(n_s-1)}(CR_d[n_s-1], ds_{final})) \;\wedge \\
\;V_b^{n_l-(n_s-3)}(CR_b[n_s-2], bs_{final}[n_s-2])) \;\equiv\; V_d^{n_l-(n_s-1)}(CR_d[n_s-2], ds_{final})) \;\wedge \\
\quad\vdots \\
\;V_b^{(n_l-1)}(CR_b[2], bs_{final}[2]) \;\equiv\; V_d^{n_l-(n_s-1)}(CR_d[2], ds_{final})) \;\wedge \\
\;V_b^{n_l}(CR_b[1], bs_{final}[1]) \;\equiv\; V_d^{n_l-(n_s-1)}(CR_d[1], ds_{final})) \;\wedge \\
\;V_b^{1}(PR_b, bs_{init}) \;\equiv\; V_d^{n_l-(n_s-1)}(PR_d, ds_{final}[1])) \;\Rightarrow \\
\\
\qquad (\forall i,\; n_l - (n_s - 2) \le i \le n_l \;:\; V_b^{i}(CR_b, bs_{final}[n_s]) \;\equiv\; V_d^{n_l}(CR_d, ds_{final}[n_s - (n_l - i)])))
\end{array}
$$

that considering Equations 10.7 and 10.8 may be reduced to:

$$
\begin{array}{ll}
CC_e \;: & ((V_b^{n_l-(n_s-1)}(CR_b, bs_{final}) \;\equiv\; V_d^{n_l-(n_s-1)}(CR_d, ds_{final})) \;\wedge \\
& \;V_b^{n_l-(n_s-2)}(CR_b[n_s-1], bs_{final}[n_s-1]) \;\equiv\; V_d^{n_l-(n_s-1)}(CR_d[n_s-1], ds_{final})) \;\wedge \\
& \;(V_b^{n_l-(n_s-3)}(CR_b[n_s-2], bs_{final}[n_s-2]) \;\equiv\; V_d^{n_l-(n_s-1)}(CR_d[n_s-2], ds_{final})) \;\wedge \\
& \quad\vdots \\
& \;(V_b^{n_l-1}(CR_b[2], bs_{final}[2]) \;\equiv\; V_d^{n_l-(n_s-1)}(CR_d[2], ds_{final})) \;\wedge \\
& \;(V_b^{n_l}(CR_b[1], bs_{final}[1]) \;\equiv\; V_d^{n_l-(n_s-1)}(CR_d[1], ds_{final})) \;\Rightarrow \\
& \\
& \qquad (\forall i,\; n_l - (n_s - 2) \le i \le n_l \;:\; V_b^{i}(CR_b, bs_{final}[n_s]) \;\equiv\; V_d^{n_l}(CR_d, ds_{final}[n_s - (n_l - i)]))
\end{array}
$$

Validation of the Verification Method

This approach to verification of pipelined loops is correct, if we can prove that the three correctness conditions that are the basis of this method collectively imply the original correctness condition of the loop:

$$\forall i,\ 1 \leq i \leq n_l \ :\ V_b^i(CR_b, bs_{final}) \ \equiv\ f(V_d^i(CR_d, ds_{final})$$

This proof is done in two steps. At the first step we will show that the prologue and steady correctness conditions guarantee that the pipelined implementation of the loop at its first $(n_l - (n_s - 1))$ initial iterations (that all end at the steady phase of the operation of the loop) is functionally correct:

$$\forall i,\ 1 \leq i \leq (n_l - (n_s - 1)) \ :\ V_b^i(CR_b, bs_{final}) \ \overset{B_r}{\equiv}\ V_d^i(CR_d, ds_{final})$$

Then, at the second step we will show that the functional correctness of the loop at the prologue and steady phases of its operation, together with epilogue correctness conditions guarantee that the pipelined implementation of the loop at its last $(n_s - 1)$ final iterations (that all end at the epilogue phase of the operation of the loop) is functionally correct:

$$\forall i,\ (n_l - (n_s - 1)) \leq i \leq n_l \ :\ V_b^i(CR_b, bs_{final}) \ \overset{B_r}{\equiv}\ V_d^i(CR_d, ds_{final})$$

Step 1: For brevity we use the following denotations in the steady phase correctness condition:

$$CC_s \ :\ (\forall i,\ 1 \leq i \leq (n_l - (n_s - 1)) \ :\ CC_a(i) \ \Rightarrow\ CC_c(i))$$

where $CC_a(i)$ is the antecedent of the steady correctness condition, and $CC_c(i)$ is the consequent of the steady correctness condition. Now, as the first step of the proof, we show that:

$$\forall i,\ 1 \leq i \leq n_l - (n_s - 1) \ :\ CC_c(i)$$

This proof is by induction on i the iteration count of the loop, and it goes as follows:

Induction Basis

We mentioned before that $V_b^0(CR_b[n_s], bs_{final}[n_s])$ refers to $V_b^1(CR_b,\ bs_{init})$ and $V_d^0(CR_d[n_s], ds_{final}[n_s])$ refers to $V_d^1(CR_d, ds_{init})$ then, $CC_p = CC_a(1)$. Then, the proof goes as follows:

premise	CC_p	(1)
premise	CC_s	(2)
expanding (2)	$\forall i,\ 1 \leq i \leq (n_l - (n_s - 1))\ :\ CC_a(i) \ \Rightarrow\ CC_c(i)$	(3)
substituting 1 for i	$CC_p = CC_a(1)$	(4)
(1) & (4)	$CC_a(1)$	(5)
$\forall$-elimination (3)	$CC_c(1)$	(6)
	$Q.E.D.$	

Induction Hypothesis

$$\forall k,\ 1 \leq k < n_l - (n_s - 1)\ :\ CC_c(k)$$

Inductive Step

At this step we will show that:

$$\forall k,\ 1 \leq k < n_l - (n_s - 1)\ :\ CC_c(k) \ \Rightarrow\ CC_c(k+1)$$

premise	$CC_c(k)$	(1)
premise	CC_s	(2)
expanding (2)	$\forall i,\ 1 \leq i \leq (n_l - (n_s - 1))\ :\ CC_a(i) \ \Rightarrow\ CC_c(i)$	(3)
substituting $(k+1)$ for i	$CC_a(k+1) \ \Rightarrow\ CC_c(k+1)$	(4)
	$CC_c(k) = CC_a(k+1)$	(5)
(1) & (4) substitution	$CC_a(k+1)$	(5)
(3) & (5) $\Rightarrow$-elimination	$CC_c(k+1)$	(6)
	$Q.E.D.$	

This completes the proof that:

$$\forall i,\ 1 \leq i \leq n_l - (n_s - 1)\ :\ CC_c(i) \tag{10.9}$$

By investigation of $CC_c(i)$ it is obvious that:

$$\forall i,\ CC_c(i) \ \Rightarrow\ V_b^i(CR_b, bs_{final}) \ \equiv\ V_d^i(CR_d, ds_{final}) \tag{10.10}$$

Then, the propositions 10.9 and 10.10 imply that:

$$\forall i,\ 1 \leq i \leq (n_l - (n_s - 1)) \ : \ V_b^i(CR_b, bs_{final}) \ \equiv \ V_d^i(CR_d, ds_{final}) \qquad (10.11)$$

and therefore, the first step of the proof is completed.

Step 2: At this step we will assume that the epilogue correctness condition is valid:

$$CC_e = \text{“}True\text{”}$$

In addition, we will use the proof of the Step 1 and show that:

$$\forall i,\ (n_l - (n_s - 1)) \leq i \leq n_l \ : \ V_b^i(CR_b, bs_{final}) \ \equiv \ V_d^i(CR_d, ds_{final})$$

Let's consider the proposition 10.9 that was proved to be valid in previous section:

$$\forall i,\ 1 \leq i \leq n_l - (n_s - 1) \ : \ CC_c(i) \qquad (10.12)$$

By substituting $n_l - (n_s - 1)$ for i, we know that the following is true:

$$CC_c(n_l - (n_s - 1)) = \text{“}True\text{”}$$

Since $CC_c(n_l - (n_s - 1))$ is the antecedent of the epilogue correctness condition and considering that CC_e is valid, it is a valid conclusion that the consequent of the epilogue correctness condition is valid, i.e.:

$$\forall i,\ n_l - (n_s - 2) \leq i \leq n_l \ : \ V_b^i(CR_b, bs_{final}[n_s]) \ \equiv \ V_d^{n_l}(CR_d, ds_{final}[n_s - (n_l - i)]) \qquad (10.13)$$

However, we know that the state $ds_{final}[n_s - (n_l - i)]$ at iteration n_l is the same as the state $ds_{final}[n_s] \ = \ ds_{final}$ at iteration i, therefore, the proposition 10.13 is the same as:

$$\forall i,\ n_l - (n_s - 2) \leq i \leq n_l \ : \ V_b^i(CR_b, bs_{final}) \ \equiv \ V_d^i(CR_d, ds_{final}) \tag{10.14}$$

The propositions 10.11 and 10.14 imply that:

$$\forall i,\ 1 \leq i \leq n_l \ : \ V_b^i(CR_b, bs_{final}) \ \equiv \ V_d^i(CR_d, ds_{final}) \tag{10.15}$$

This means that the three correctness conditions imply the general correctness condition of the design and our verification method is correct.

10.1.4 Discussion

When the loop-unfolding technique is used to verify the implementation of the pipelined loops, the values of $n = |CR_d = AR_d \cup PR_d|$ registers at each of the n_l iterations of the loop are computed (in the cases where the approach can be used). Therefore, the complexity of this verification algorithm is $O_1(n_l) = n \times n_l$.

In our verification technique based on backward value propagation, the values of $n = |CR_d = AR_d \cup PR_d|$ registers at $2n_s$ iterations of the loop are computed (these iterations consist of $(n_s - 1)$ iterations starting in prologue phase, $(n_s - 1)$ iterations ending in epilogue phase, an arbitrary iteration i both starting and ending in steady phase, where the values are propagated backwards, and its consecutive iteration where the values are propagated forward. Therefore, the complexity of this verification algorithm is defined by $O_2(n_l) = 2n_s \times n$. It is apparent that the complexity of this algorithm is a constant for a loop, regardless of its number of iterations n_l.

In our verification technique introduced in previous section (based on criticality masking technique), the complexity of the verification algorithm is also a constant and is not affected by the number of loop iteration n_l. However, even the worse case complexity of this method is better than the verification technique based on backward propagation.

Consider the proofs of the prologue correctness condition CC_p and epilogue correctness condition CC_e. As the first step in proving CC_p the correctness of the values of the critical registers at the end of the first iteration of the loop should be calculated, so, all the n_s stages of the loop are considered in the verification exercise. For the second iteration the values of the $n_{n_s-1} = |T[n_s - 1]|$ $(n_s - 1 < n)$ at the end of stage $n_s - 1$ should be calculated and so the first $(n_s - 1)$ stages of the loop are considered in the verification exercise, For the third iteration the values of $n_{n_s-2} = |T[n_s - 2]|$ at the end of stage $n_s - 2$ should be calculated, so the first $(n_s - 2)$ stages of the loop are considered in verification exercise. This trend continues so that for the n_s-th iteration of the loop, only the first stage of the loop is considered in verification exercise.

For proving the correctness of CC_e on the other hand, at iteration $(n_l - (n_s - 1)$, we know that the values of the temporaries prior to the final stage of that iteration are correct and based on that we need to show that at the end of the final stage of this iteration critical registers have correct values. Therefore, for iteration $(n_l - (n_s - 1)$ only stage n_s of the loop is considered in the verification exercise. For iteration $(n_l - (n_s - 2)$ the last two stages of the loop is considered in the verification exercise. This trend continues so that for iteration $n_l - 1$ all the stages of the loop but the first two, and for iteration n_l all the stages of the loop but the first are considered in the verification exercise. This means that the complexity of the verification algorithm for prologue and epilogue phases combined is smaller than $(1 + (n_s - 1)/2) \times n = (n_s + 1)/2 \times n$.

At the steady phase, the operation of each stage of the loop is considered once. At each stage $j < n_s$, the values of $n_j = |T[j]|$ (where $n_j < n$) registers are computed, and at the stage n_s the values of all the registers are calculated, therefore, the complexity of the steady phase verification algorithm is less than $1 \times n$. Then, the complexity of the algorithm for complete verification of the loop is defined as $O_3(n_l) < (n_s + 3)/2 \times n$. We know that pipelining is used when the number of loop iterations n_l is relatively large (at least in the order of hundred). While for pipelined designs the optimal number of stages is defined to be in the range $[4, 8]$, and so $n_s << n_l$ [26]. [3] Considering these facts, it is apparent that the verification techniques presented in this chapter, offer considerable improvements in the time required for verification of pipelined designs.

10.1.5 Concluding Remarks

We considered the verification of pipelined loops in detail, since it is the basis of verification of functionally pipelined designs that is the subject of next section. In functional pipelining a global data introduction interval δ_0 is assumed for the circuit. This means that the initial state of the control flow graph of the design $S0_d$ is executed (fired) once every δ_0 cycles. If we assume an outer loop for the control flow graph of this design, the problem is the same as loop winding where $\delta_l = \delta_0$. Functional pipelining will be discussed in more detail in the next section.

[3] "It is a fallacy that increasing the number of pipeline stages always increases the performance. Pipelining increases the throughput - the number of operations completed per unit of time - but it does not reduce the execution time of each operation. In fact it usually slightly increases the execution time of each operation due to overhead in the control of the pipeline. The increase in throughput lowers the total execution time, even though no single operation is executed faster. The fact that the execution time of each operation and consequently, the overall execution delay of one iteration does not decrease puts limits on the practical depth of a pipeline. The performance flattens out when the number of pipeline stages reaches 4 and actually drops when the execution portion is pipelined 16 deep". [26]

10.2 Functional Pipelining in Design Synthesis

In the case of functional pipelining, the sequencing graph is generated under the assumption of a global data introduction interval (δ_0) constraint [17]. This means that there is a δ_0 interval between two consecutive executions of the operations corresponding to each state of the controller. In a pipelined design, the data introduction interval δ_0 is smaller than the latency λ. In the case of a non-pipelined design, the data introduction interval is equal to the latency: $\delta_0 = \lambda$.

The synthesis of the pipelined circuits is very complicated and is considerably more restricted than the manual design of pipelined circuits. In what follows, we consider the verification of the class of functionally pipelined designs that may be generated through synthesis. Throughout this discussion, we assume that out of order execution is permitted only under the condition that it doesn't result in out of order completion.

10.2.1 Sequential Specifications

In this section we consider the functional pipelining of sequential specifications. A sequential specification consists of a single critical path, and no iterative or conditional constructs are embedded in its flow graph.

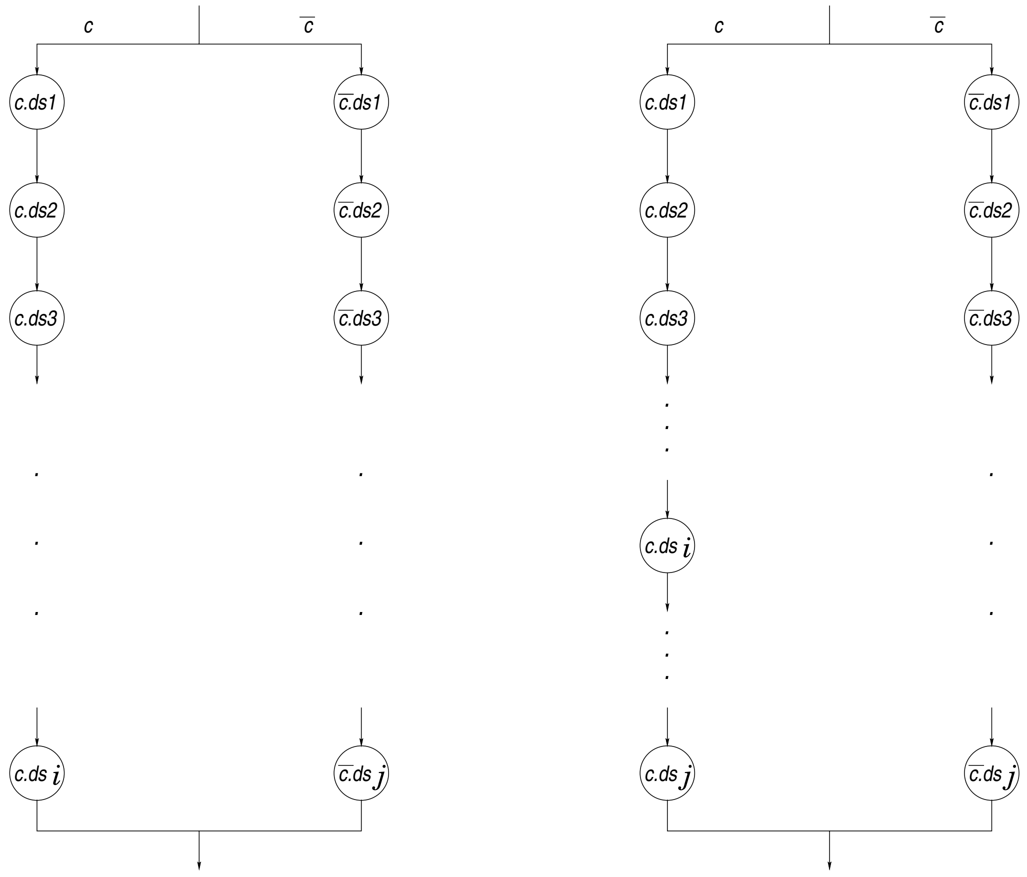
Now let's consider a loop that its body consists of a single critical path of the above specification, and which is repeated indefinitely ($n_l = \infty$). Implementation of a sequential specification such of this kind, with execution delay λ and data introduction interval δ_0, is synthesized under the same rule as a pipelined implementation of a loop such as above with the loop initiation interval $\delta_l = \delta_0$. It is obvious that the problem of synthesis and verification of functionally pipelined sequential blocks is the same as the problem of synthesis and verification of a pipelined loop.

10.2.2 Specifications with Conditional Constructs

This section discusses the functional pipelining of the specifications that are composed of sequential or conditional constructs (the iterative blocks are not allowed). Before presenting more detailed explanation on synthesis and verification of these specifications, a few definitions are necessary.

Definition 10.1 Balancing A Conditional Basic Block A conditional block is said to be *balanced* if there are equal number of states in the two branches of that block. Even when a conditional block is not balanced, it may be transformed into one.

Figure 10.15 shows a conditional basic block. c is the condition variable, and we represent the states of the 'true' branch by $c.ds_1$ to $c.ds_i$ and the states of the 'false' branch by $\overline{c}.ds_1$



(a) **An unbalanced conditional construct** (b) **A balanced conditional construct**

Figure 10.15: **Balancing a Conditional Basic Block where $(\mathbf{i} < \mathbf{j})$**

to $\overline{c}.ds_j$ to emphasize that these states are executed conditionally. If CTL is the set of all control signal lines from the controller to the control inputs of the components in the data-path, then $c.cs_k \subseteq CTL$ listed besides the state $c.ds_k$ $(\overline{c}.ds_k)$ is the set of all control signals that are asserted high at that state.

If $j < i$ $(i < j)$, then to balance the conditional basic block, the balancing states $\overline{c}.ds_{j+1}$ to $\overline{c}.ds_i$ $(\overline{c}.ds_{i+1}$ to $\overline{c}.ds_j)$ should be added at the end of 'false' ('true') branch, where for $j < k \leq i$ $(i < k \leq j)$ the labels of these states $c.cs_k = \emptyset$ ($\overline{c}.cs_k = \emptyset$). This means that at the balancing states no register transfer operations occur.

Definition 10.2 Merging the Conditional Branches One of the synthesis transformations is merging the conditional branches of a balanced conditional block. In this procedure the two conditional branches are merged into one. The operations of the 'true' and 'false' branches are executed simultaneously, and at the final state of the branch the results of the operations are assigned to appropriate registers, conditionally. This transformation affects the flow of the operation of the controller, as well as the structure of the data-path, but maintains the overall functionality of the circuit the same. This transformation is possible under the condition that the conditional block is balanced (if not it has to be transformed into one).

The execution of the branches of a conditional block is a mutually exclusive process, therefore, in synthesis of the conditional blocks resources may be shared between the two branches. When merging the branches, the operations of the two branches are assigned to concurrent execution cycles, therefore, exclusive sets of resources should be dedicated to each branch. This is especially true in the case of the design registers.

Let's assume that $AR_d \subseteq CR_d$ is the set of active registers of the conditional block (the active registers of a conditional block are defined similar to the active registers of an iterative block. The active registers of a conditional block are those registers that are loaded at least in once within the conditional block they are loaded with new values. Corresponding to each active register r_d of a control block, two new temporary registers $c.r_d$ and $\overline{c}.r_d$ are introduced to the circuit. These two registers assume the contents of r_d at the initial state of the control block. In all the register transfer operations in the 'true' branch of the conditional block $c.r_d$, and in the 'false' branch $\overline{c}.r_d$ replaces r_d. At the final state of the block the contents of the temporary registers are conditionally transfered to the critical registers. If the value of c at the initial state of the block is true then the 'true' set of the temporary registers and otherwise the 'false' set of temporary registers are written to the critical registers.

Merging the branches of a conditional block is a transformation that is performed in synthesis of a pipelined implementation [49], because at the initial state of a conditional block the value of the condition flag c may not be known (execution of the previous iteration is not terminated, and the initial values of the registers in current iteration are not yet available). Then the operations of the two branches are concurrently performed and at the final state of the block, when the value of the condition flag is surely available, the appropriate values are written to the registers.

After merging the branches of all the conditional blocks, a specification with conditional blocks is transformed into a sequential specification. The synthesis and verification of a functionally pipelined implementation of such a specification is as discussed in the case of sequential blocks.

10.2.3 Specifications with Conditional and Iterative Constructs

The synthesis of functionally pipelined implementations of the specifications with iterative constructs is a hard problem. Most synthesis systems restrict the iterative constructs in the specification of pipelined circuits to those with known number of iterations (such as *for* loops). Prior to the synthesis the iterative blocks are unrolled, and then, the sequencing graph is subjected to scheduling [49]. After merging the branches of all the conditional blocks and unrolling all the iterative constructs, a specification with conditional and iterative blocks is transformed into a sequential specification. The synthesis and verification of a functionally pipelined implementation of such a specification are as discussed in the case of sequential blocks.

10.3 Conclusion

In this chapter we presented the verification of synthesized implementations of iterative constructs in general, and pipelined iterative constructs, in particular. We discussed loop-winding, a design technique used to optimize the execution delay of a loop, and on that basis functional pipelining, a design technique used to optimize the overall execution of a general design. We showed that synthesis and therefore verification of pipelined designs are different from synthesis and verification of non-pipelined designs. We pointed out these differences through a detailed analysis, and presented extensions of our verification method to account for loop-winding and functional pipelining in a synthesized designs. Finally, we presented a set of three correctness conditions for pipelined synthesized designs, and proved that these three correctness conditions are sufficient for verifying these deigns.

Chapter 11

Implementation and Results

The method discussed in the previous sections has been implemented in a correctness condition generator module integrated with the ASSERTA high level synthesis system. ASSERTA has been in development for several years and is relatively mature. ASSERTA accepts behavioral specifications in an intermediate HDL format (that is directly extracted from behavioral VHDL) and generates RTL designs also in an intermediate HDL format (that is directly translated into structural VHDL). Using parallel synthesis algorithms, ASSERTA searches through vast regions of design space [18]. ASSERTA uses enhancements of force-directed list scheduling [24, 23] and a hierarchical clique partitioning algorithm for register allocation [32]. ASSERTA has been used to generate numerous designs both in the university and industry and has been thoroughly tested using systematic benchmark development, test generation and simulation [65]. In addition, as a byproduct of the synthesis process, ASSERTA automatically generates control flow properties in CTL logic [13, 43] for verification by the SMV model checker [38].

Figure 8.1 shows the integration of the correctness condition generator (CCG) with the ASSERTA system as explained in the previous section. The CCG component of ASSERTA has helped us to determine how much of the verification effort can be automated. A limitation of the verification condition generator currently is that it can handle a smaller subset of VHDL than that can be synthesized by ASSERTA. [1]

The modified ASSERTA system with this generator produces a PVS file containing declarative spec-

[1] ASSERTA algorithms are capable of handling delta-delayed signal assignments, interacting multiple processes, wait statements and resolution functions. Proper handling of these constructs requires generation of a controller containing multiple interacting finite state machines [44]. The controller embodies not only the explicit control flow in the behavior specification but also the implicit control flow due to VHDL semantics. We are currently investigating methods to include correctness conditions to handle such control flow within the framework of the technique discussed in this dissertation.

Design Name	Critical Paths	Spec Vars	RTL Regs	Beh States	RTL States	Axioms	Lemmas to Prove	PG Time (sec)	TC Time (sec)	Proof Time (mins)
MAX & SUM	5	5	4	11	18	87	14	0.04	12.39	1.22
Stack	9	10	8	27	32	188	24	0.11	16.91	5.16
Compress	18	10	3	19	42	244	20	0.08	4.40	3.09
Decompress	18	10	3	22	40	258	29	0.07	12.22	6.53
Move	8	13	6	20	40	276	50	0.11	26.77	12.96
Video Mom	3	15	9	33	58	168	6	0.19	43.24	17.19
Find	14	23	9	38	90	433	93	0.43	14.40	24.54
TLC	18	19	8	50	70	395	94	0.33	84.07	58.98
DCT	7	37	8	115	146	256	80	0.48	794.42	97.94

Table 11.1: **Experimental Results**

ifications of the behavior and data path and formal specification of the controller. In addition, it produces all of the critical path equivalence lemmas and proof scripts to prove these lemmas. The PVS theories generated are not necessarily very elegant, but are amenable to completely automated verification. PVS system is used to execute these scripts automatically. No manual interaction is necessary to conduct the proof and inspection is necessary only in the event of a failure.

Table 11.1 shows results pertaining to several verification exercises using the ASSERTA synthesis system and the PVS proof checker. [2] All of these exercises were successful in verifying the corresponding designs. Note that behavior states are formed by assuming one assignment or conditional statement per state. In some instances RTL controller may have fewer states than the behavior automaton due to parallelization of operations during synthesis. Also, note that the ASSERTA execution time for generating the designs is a few minutes and the CCG execution time [3] is under one minute for each of these examples. Figures 11.1 to 11.4 show the proof time of each example design as a function of various design parameters.

11.1 Discussion

During our experiments we made the following observations:

1. The time of generating the proof scripts is negligible compared to the time required for

[2]These exercises were conducted on a SUN SPARC5 with OS 5.5.1.

[3]Note that this is the execution time of CCG for generating the proof scripts, and the times presented in the table are the PVS execution times for verifying the proofs.

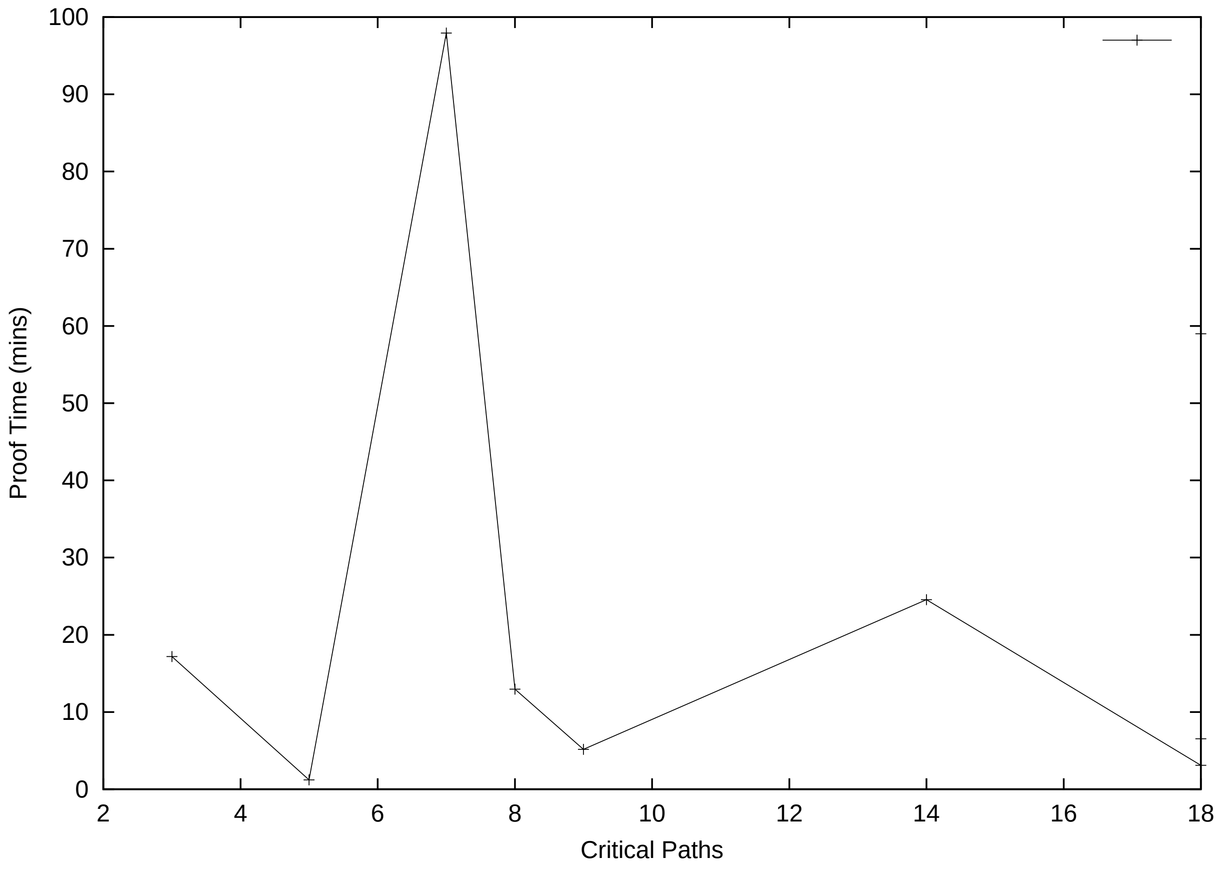


Figure 11.1: **Proof Time as Function of the Number of Critical Paths**

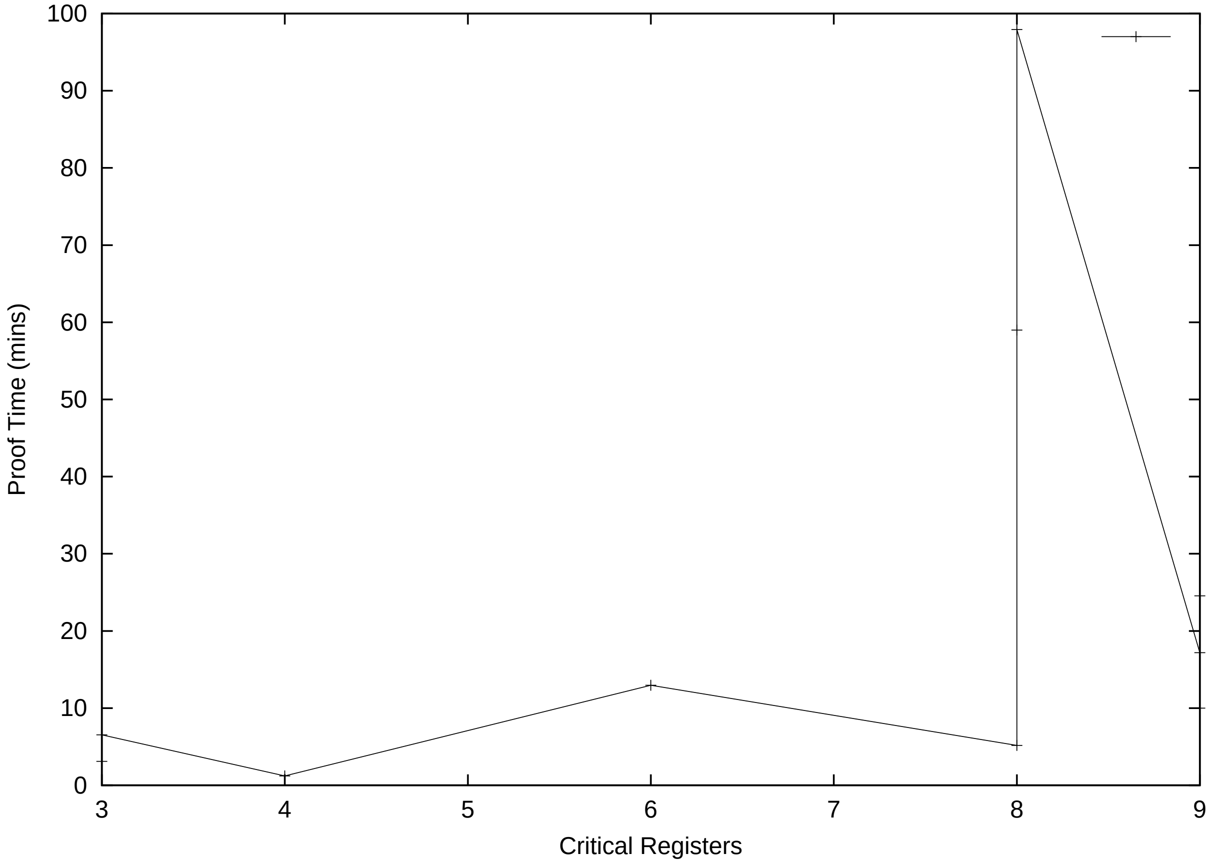


Figure 11.2: **Proof Time as Function of the Number of Critical Registers**

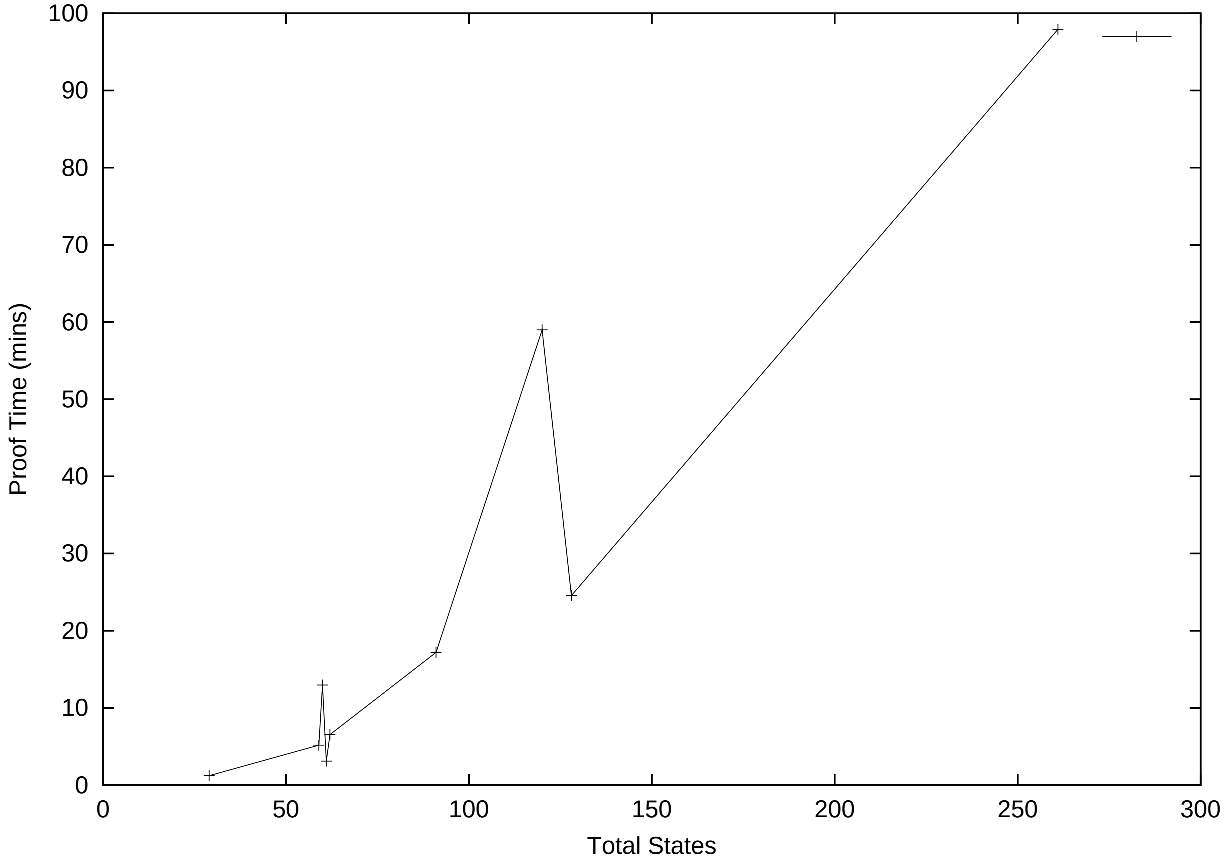


Figure 11.3: **Proof Time as Function of the Total Number of States**

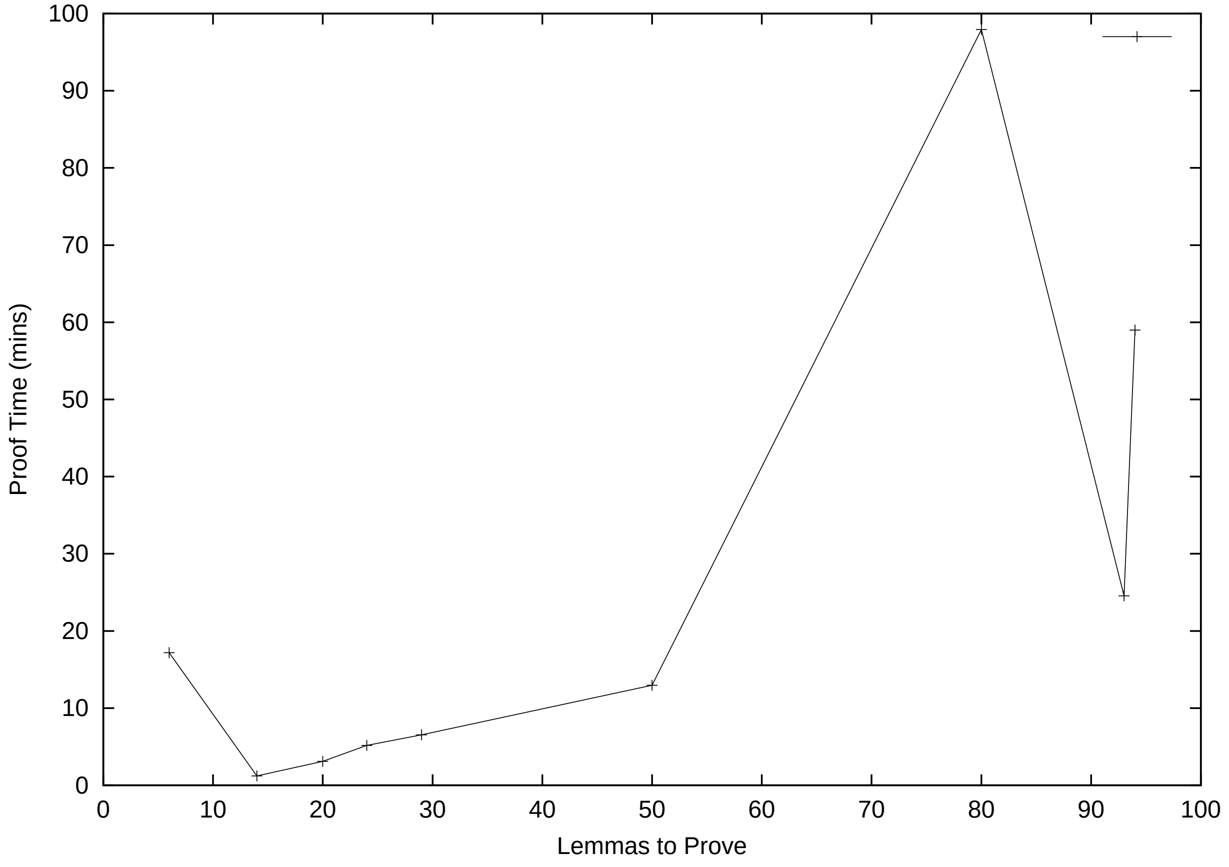


Figure 11.4: **Proof Time as Function of the Number of Lemmas**

verifying the proofs by the proof checker.

2. The CPU time reported by PVS for verifying each proof is minimal. However, the actual time spent on verifying the proofs is dominated by the memory management (garbage collection) algorithms of PVS, that is more significant in comparison to the proof processing time.

3. The verification time is independent from the bit size of variables and registers. This means that for any given design, we can arbitrarily increase the size of the variables and registers without affecting the proof processing time.

   Several researchers [27] have noted that spatial abstraction methods can be effectively exploited to improve the efficiency of the verification algorithms. However, the degree of bit-level reduction that can be achieved in a data-abstraction method is limited. As data-path abstraction is usually used in conjunction with model checking, even the abstracted models may not be verifiable due to the state explosion problem.

   However, in our method that relies on analysis of uninterpreted values, the bit-sizes of all signals (variables or registers) are implicitly abstracted away. This is the best result achievable by any spatial-abstraction algorithm.

4. Our experiments are not indicative of a direct relation between any single parameter and the proof time (Figures 11.1 to 11.4). However, the follwoing observations/conclusions should be pointed out.

   The number of the equivalence lemmas corresponding to a particular critical path is the same as the number of critical registers across that path. Therefore, the time for proving the correctness of a given critical path is directly proportional to the number of critical registers across that path. As the number of the critical registers differ from one critical path of a design to the other, the proof time of a given design may not be simply defined in terms of the number of critical registers or the critical paths of that design.

   For a design that is composed of critical paths of comparable size (i.e. critical paths comprising similar number of states), the processing time is directly proportional to the number of critical paths. However, we noticed a nonlinear dependency between the proof processing time and the number of states in a critical path (Figures 11.5, 11.6, and 11.7). We observed that proof time increases nonlinearly as the number of states in a critical path increases.

   It can be directly concluded from this dependency that in our verification approach, the limiting factor is not the size of the design, rather, it is the maximum number of the states in a critical path of the design. This means that the designs of arbitrarily large size may be verified through the approach presented in this dissertation as long as no critical path with unusually large number of states belong to the control flow graph of the design.

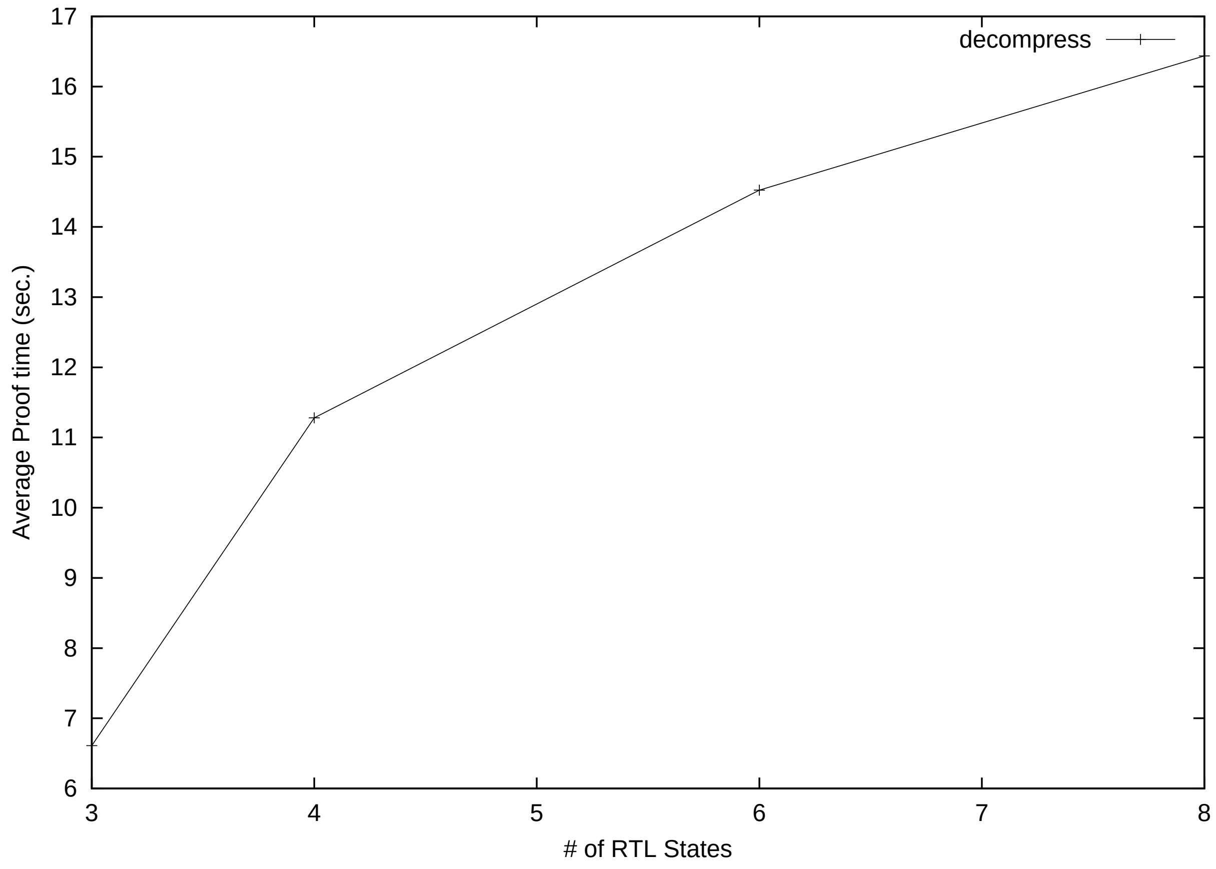


Figure 11.5: **Average Proof Time of DECOMPRESS Equivalence Lemmas as Function of the Number of States per Path**

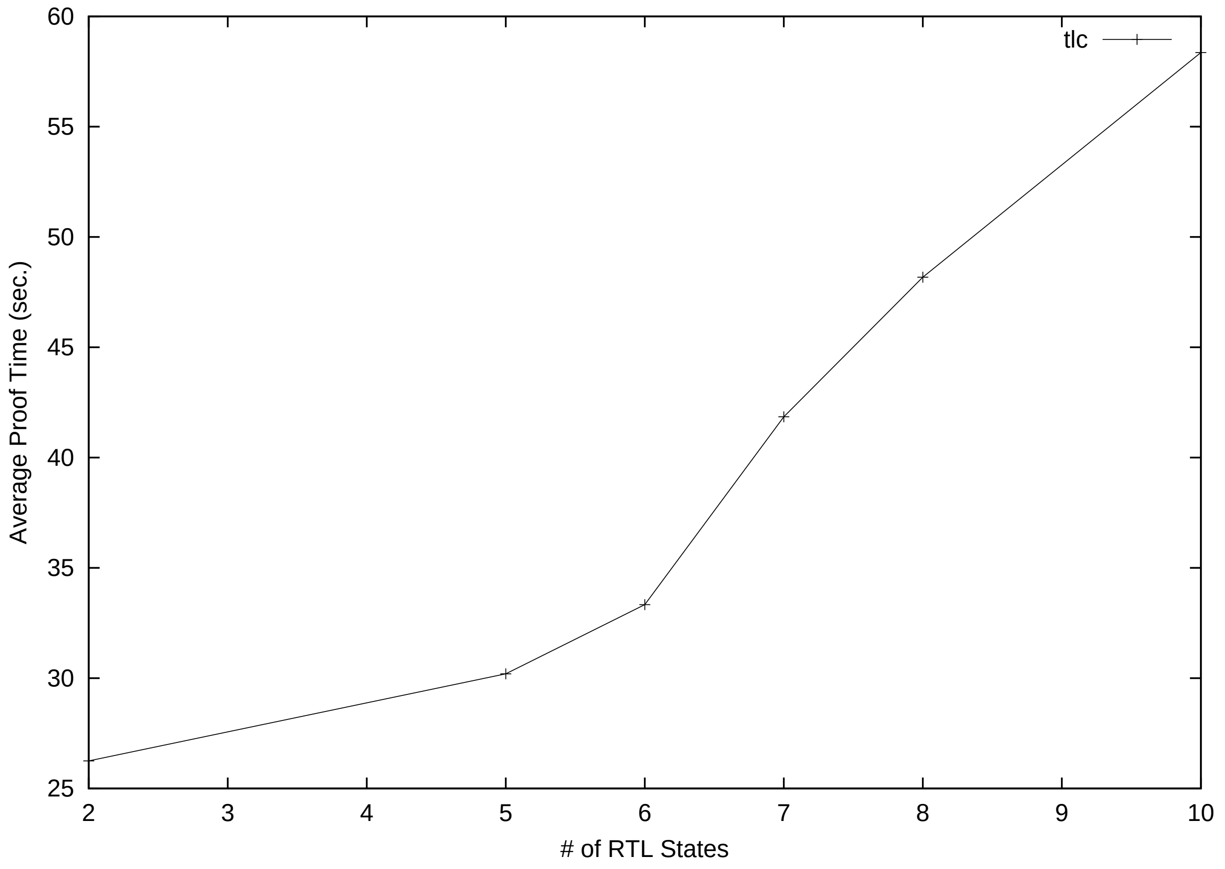


Figure 11.6: **Average Proof Time of TLC Equivalence Lemmas as Function of the Number of States per Path**

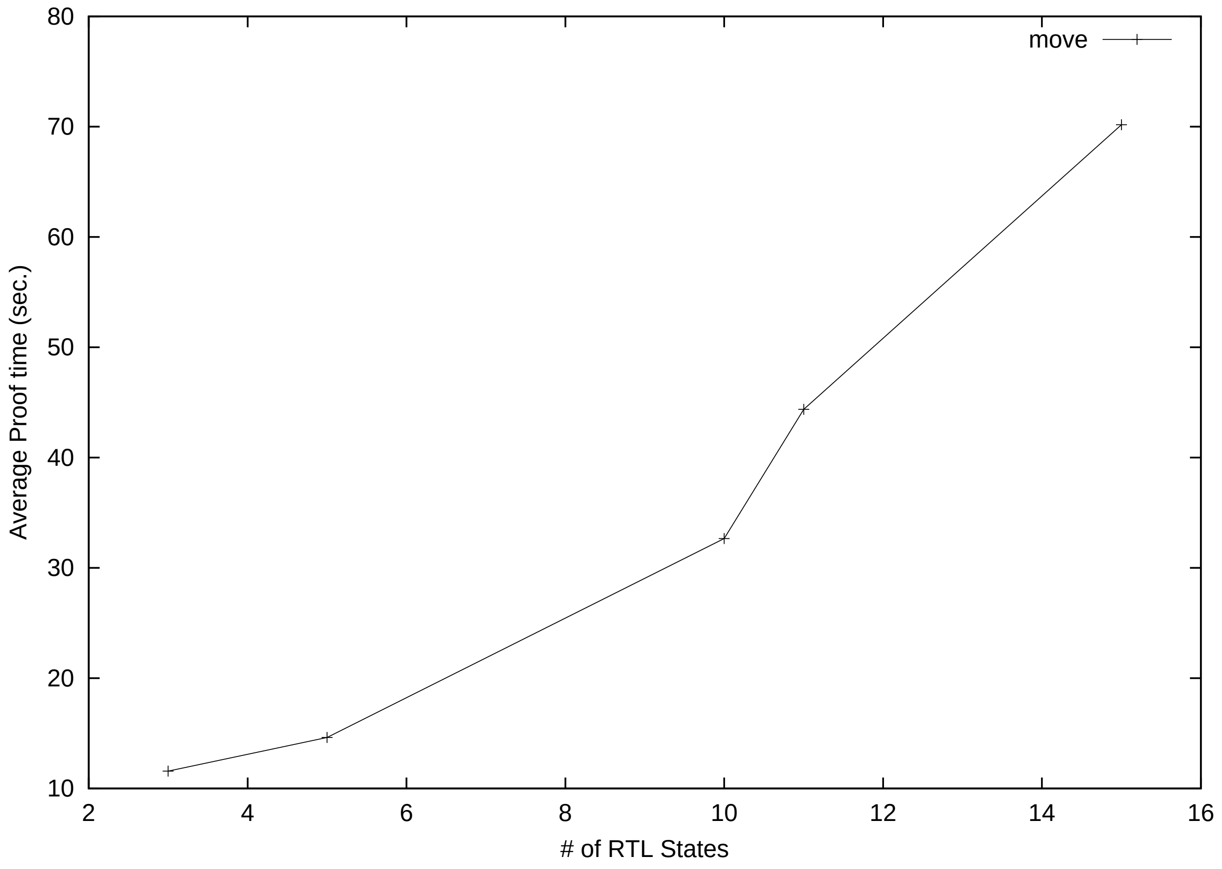


Figure 11.7: **Average Proof Time of MOVE Equivalence Lemmas as Function of the Number of States per Path**

The observations mentioned above indicate that our method is very effective in verification of the designs with wide data-paths. Also, the verification method is very effective in verification of control-intensive designs. The data-intensive designs can be successfully verified through our verification method as long as the critical paths of the design are not excessively large.

Chapter 12

Conclusion and Future Work

In this chapter, we conclude the dissertation with some final remarks. We reiterate our thesis on formal verification of synthesized register transfer level designs, present a summary of the scope and limitations of our approach and take a glance at paths for future research.

12.1 Discussion

The advent of highly sophisticated synthesis systems in recent years has provided the potential for generating synthesized designs of unprecedented size and complexity. As the degree of intricacy of synthesized systems has improved, it has become increasingly difficult to ensure that they won't malfunction due to design errors. This problem can assert as a restraining factor on synthesis systems gaining widespread acceptance. Therefore, a sense of urgency has been raised in computer aided design research community to address the verification of synthesized designs.

Conventional testing methods were commonly used for verification of synthesized designs. However, these techniques do not scale well and are limited in their application domain. As more advanced synthesis systems were employed to generate larger, more sophisticated designs, the necessity of alternative, more systematic approaches to verification became apparent. Besides, in order not to forfeit the benefits of automatic design synthesis, a verification approach used in conjunction with a synthesis system should be susceptible to automation. Formal methods currently offers the most promising vehicle to achieve this goal.

Several researchers have developed formal methods based on theorem proving for verification of synthesized designs. Formal synthesis methods (transformation-based synthesis, incremental verification, etc) have been used to generate designs that are correct by construction. Although effective, these methods are tedious and time consuming and cannot be automated. Post-design formal verification methods have also been applied to verify synthesized designs. Even-though these methods that are mainly based on model checking render to automation, they are limited by the size of the designs they can verify.

In this dissertation, we presented an effective approach for formal verification of synthesized register transfer level designs. Our approach, that is based on theorem proving, effectively exploits the common characteristics of high-level synthesis algorithms to achieve efficient verification. Widely used algorithms in high-level synthesis tools retain the overall control flow structure of the behavioral specification allowing code motion only within basic blocks. Further, high-level synthesis algorithms are in general oblivious to the mathematical properties of arithmetic and logic operators, selecting and sharing RTL library modules solely based on matching uninterpreted function symbols and constants. Many researchers have noted that these features of high-level synthesis algorithms can be exploited to develop efficient verification strategies for synthesized designs. We have effectively exploited these features in a verification methodology that achieves efficient and fully automated verification of synthesized designs.

It should be noted that a similar approach can be applied for the purpose of software verification, provided that no optimizations based on code-motion or interpretation of the functions or variables are performed. However, this may restrict the application of the method for verifying the software.

12.2 Future Work

In what follows we summarize the scope and limitations of our work.

- The application of the method presented in this dissertation is limited to verification of the synthesized designs generated through synthesis processes that perform restricted code motion. In such a synthesis process, the set of all operations within a behavioral synthesis basic block is mapped into the set of operations scheduled in a single block of the controller of the RTL design. However, while performing primitive and restricted code motion that respect the operational dependencies, some synthesis transformations used in certain high-level synthesis tools, violate the restriction of code motion within the boundaries of basic blocks. Currently, our verification tool cannot account for such transformations. For this reason, some correct designs generated by this class of synthesis tools may not be verifiable through our method.
- Our verification method is based on, and limited to the analysis of symbolic and uninterpreted values of the variables and uninterpreted contents of registers. Certain properties of a synthesized design may depend on bit-level characteristics of its signals besides their uninterpreted values. At its present state, our verification tool is not capable to reasoning about such

properties, and we are limited to verification of the synthesized designs that can be verified solely by reasoning about their uninterpreted signals and functional units.

We foresee a number of enhancements that can significantly improve the effectiveness of our method and expand its application domain. These enhancements, that address the above problem, present interesting topics for future research within the framework of the verification approach presented in this dissertation:

1. Extension of the methodology should be considered to accommodate verification of synthesized designs generated through synthesis processes that utilize scheduling transformations that perform code motion across the border points of the basis blocks. This will considerably expand the intersection of the set of synthesized designs and the class of the synthesized designs that are verifiable through this approach.

2. Verification of synthesized designs that require bit-level analysis should be addressed. This can possibly be achieved by integrating the verification tool with a model-checker, where the general lemmas are proven as discussed in this work, and the limited set of bit-level properties are verified through model-checking. In this way the verification of RTL design in its entirety is ensured.

3. The verification methodology presented in this dissertation was developed to target the verification of the RTL designs that are synthesized from single process behavior descriptions. Extending the scope of this approach to encompass the RTL designs that are synthesized from multiple (possibly communicating processes) presents a challenging topic for research.

4. One of our goals in developing this methodology was to remedy the deficiencies of existing methods in dealing with verification of medium to large sized synthesized RTL designs, where the time and memory requirements grow exponentially with increasing size. During our experiments, we noticed that our verification practice is not limited by the time and memory resources available, rather by the memory management (garbage collection) algorithms of the PVS theorem proving system.

   Regardless of the amount of available memory during the verification, the upper-bound of the utilized memory is defined by PVS. This upper-bound is often considerably smaller than the available memory. Also, the memory management algorithms present a considerable overhead on the real time required for verifying the proofs. This overhead is the dominant factor in the overall processing time of the proofs. We believe that alternative theorem proving systems may offer more suitable environments for our verification exercise, by utilizing the resources more efficiently.

12.3 Summary of Contributions

The following are the key contributions of this dissertation:

(1) Introduction of the notion of synthesis aware post-synthesis verification. Presenting discussions on how to apply the knowledge of methodical design to identify system properties that are the consequence of the construction methods, and to exploit these properties for the purpose of verification.

(2) Formalization and Formulation in higher order logic in a theorem proving environment mathematical models for the register transfer level synthesized designs and their behavior specifications, and a set of sufficient correctness conditions for these designs.

(3) Introduction of a formal approach for functional verification of RTL designs, and presenting a notion of correctness of the systems in general, and RTL designs in particular based on observation equivalence of the synthesized design and its behavior specification.

(4) Implementation of the formal verification approach in a verification tool integrated with a high-level synthesis system that performs model extraction, correctness condition generation and proof generation automatically and without user interaction.

(5) Development of a formal method for verifying RTL design that is independent of the bit-size of variables and registers. As a result of considering uninterpreted values for variables (registers) and operators (functional units) in this approach, the bit-size of all design elements may be increased arbitrarily without affecting the verification time.

(6) Development of formal methods for verification of implementations of special behavior constructs such as loops and less conventional structurally or functionally pipelined designs, and identification of a set of sufficient correctness conditions for pipelined designs.

Appendix A

PVS Specification and Proof of Equivalence Theorem

```
EQUIV : THEORY
  BEGIN

  BehState   : TYPE+
  BehCPath   : TYPE+
  BehEPath   : TYPE+ = list[BehState]

  RTLCPath   : TYPE+
  RTLEPath   : TYPE+
  RTLState   : TYPE+

  IMPORTING  LISTS[BehState]

  First_b     : [BehCPath->BehState]
  Last_b      : [BehCPath->BehState]

  S0_b        : BehState

  B_s         : [BehState -> RTLState]
  B_p         : [BehCPath -> RTLCPath]
  B_e         : [BehEPath -> RTLEPath]

  equivalent  : [BehEPath,RTLEPath -> bool]
  equivalent_paths : [BehCPath,RTLCPath -> bool]
  equivalent_states : [BehState,RTLState -> bool]
  CorrectEPath : PRED[BehEPath]

  c_path_equivalence : AXIOM
      (FORALL (p_b: BehCPath) :
          equivalent_paths(p_b,B_p(p_b))
    IMPLIES
      (equivalent_states(First_b(p_b),B_s(First_b(p_b))) AND
       equivalent_states(Last_b(p_b),B_s(Last_b(p_b)))))

  e_path_nonempty : AXIOM
      (FORALL (e_b: BehEPath) : NOT(null?(e_b)))

  e_path_equivalence : AXIOM
      (FORALL (e_b: BehEPath) :
         (equivalent_states(car(e_b),B_s(car(e_b))) AND
          (null?(cdr(e_b)) OR equivalent(cdr(e_b),B_e(cdr(e_b))))) IFF
             equivalent(e_b,B_e(e_b)))

  c_path_existence : AXIOM
      (FORALL (s_b: BehState, e_b: BehEPath, e_b2: BehEPath) :
         e_b = cons(s_b, e_b2) =>
            (EXISTS (p_b: BehCPath) :
               First_b(p_b) = car(e_b2) AND Last_b(p_b) = s_b))

  e_path_correct : AXIOM
      (FORALL (e_b: BehEPath) :
         equivalent(e_b,B_e(e_b)) = CorrectEPath(e_b))
  initial_state_equivalence : AXIOM
       equivalent_states(S0_b,B_s(S0_b))
  initial_state : AXIOM
    (FORALL (s_b: BehState, e_b: BehEPath) :
        e_b = cons(s_b,null) => (s_b = S0_b))
  path_equivalence : THEOREM
        (FORALL (p_b: BehCPath) : equivalent_paths(p_b,B_p(p_b)))
  IMPLIES
        (FORALL (e_b: BehEPath) : equivalent(e_b,B_e(e_b)))
  END  EQUIV
```

Figure A.1: **Execution Path Equivalence Theory**

```
LISTS[T : TYPE+] : THEORY
  BEGIN
  list_induction : AXIOM
    (FORALL (P: PRED[list[T]]) :
     (((FORALL (t: T, l: list[T]) : (l = cons(t,null)) => P(l)) and
       (FORALL (t : T, l : list[T]) : P(l) => P(cons(t,l)))) =>
           FORALL (l : list[T]) : P(l)))
  END LISTS
```

Figure A.2: **List Induction Theory**

```
(EQUIV
 (|base_path_equivalence| "" (SKOLEM!)
  ((""  (FLATTEN)
    ((""  (LEMMA "initial_state" ("s_b" "s_b!1" "e_b" "e_b!1"))
      ((""  (PROP)
        ((""  (REPLACE -1 (-2) LR)
          ((""  (LEMMA "e_path_equivalence" ("e_b" "e_b!1"))
            ((""  (REPLACE -3 (-1) LR)
              ((""  (ASSERT)
                ((""  (LEMMA "initial_state_equivalence")
                  ((""  (LEMMA "null_path_equivalence")
                    ((""  (LEMMA "e_path_correct" ("e_b" "e_b!1"))
                      ((""  (REPLACE -6 (-4) RL)
                        ((""  (REPLACE -1 (-4) LR) ((""  (PROP) NIL NIL))
                          NIL))
                        NIL))
                      NIL))
                    NIL))
                  NIL))
                NIL))
              NIL))
            NIL))
          NIL))
        NIL))
      NIL))
    NIL))
  NIL)
 (|inductive_path_equivalence| "" (SKOLEM!)
  ((""  (FLATTEN)
    ((""  (LEMMA "e_path_correct" ("e_b" "e_b!1"))
      ((""  (REPLACE -1 (-2) RL)
        ((""  (LEMMA "e_path_equivalence" ("e_b" "e_b!1"))
          ((""  (PROP) ((""  (POSTPONE) NIL NIL)) NIL)) NIL))
        NIL))
      NIL))
    NIL))
  NIL)
 (|path_equivalence| "" (FLATTEN)
  ((""  (SKOLEM!)
    ((""  (LEMMA "list_induction" ("P" "CorrectEPath"))
      ((""  (SPLIT)
        (("1" (INST -1 "e_b!1")
          (("1" (LEMMA "e_path_correct" ("e_b" "e_b!1"))
            (("1" (REPLACE -1 (-2) RL) (("1" (PROPAX) NIL NIL)) NIL))
            NIL))
          NIL)
         ("2" (HIDE 2)
          (("2" (SKOLEM 1 ("s_b!2" "e_b!2"))
            (("2" (FLATTEN)
              (("2"
                (LEMMA "initial_state" ("s_b" "s_b!2" "e_b" "e_b!2"))
                (("2" (PROP)
                  (("2" (REPLACE -1 (-2) LR)
                    (("2" (LEMMA "e_path_equivalence" ("e_b" "e_b!2"))
                      (("2" (REPLACE -3 (-1) LR)
                        (("2" (ASSERT)
                          (("2" (LEMMA "initial_state_equivalence")
                            (("2"
                              (LEMMA "e_path_correct" ("e_b" "e_b!2"))
                              (("2"
                                (REPLACE -5 (-3) RL)
                                (("2"
                                  (PROP)
                                  (("2"
                                    (REPLACE -3 (-2) LR)
                                    (("2" (PROPAX) NIL NIL))
                                    NIL))
                                  NIL))
                                NIL))
                              NIL))
                            NIL))
                          NIL))
                        NIL))
                      NIL))
                    NIL))
                  NIL))
                NIL))
              NIL))
            NIL))
          NIL)
         ("3" (SKOLEM 1 ("s_b!2" "e_b!2"))
          (("3" (FLATTEN)
            (("3"
              (LEMMA "c_path_existence"
               ("s_b" "s_b!2" "e_b" "cons(s_b!2, e_b!2)" "e_b2"
                "e_b!2"))
              (("3" (PROP)
                (("3" (SKOLEM!)
                  (("3" (INST -3 "p_b!1")
                    (("3" (LEMMA "c_path_equivalence" ("p_b" "p_b!1"))
                      (("3" (PROP)
                        (("3" (REPLACE -4 (-2) LR)
                          (("3"
                            (LEMMA "e_path_equivalence"
                             ("e_b" "cons(s_b!2, e_b!2)"))
                            (("3" (ASSERT)
                              (("3"
                                (LEMMA
                                 "e_path_correct"
                                 ("e_b" "cons(s_b!2, e_b!2)"))
                                (("3"
                                  (REPLACE -1 (-2) LR)
                                  (("3"
                                    (LEMMA
                                     "e_path_correct"
                                     ("e_b" "e_b!2"))
                                    (("3"
                                      (REPLACE -1 (-8) RL)
                                      (("3" (PROP) NIL NIL))
                                      NIL))
                                    NIL))
                                  NIL))
                                NIL))
                              NIL))
                            NIL))
                          NIL))
                        NIL))
                      NIL))
                    NIL))
                  NIL))
                NIL))
              NIL))
            NIL))
          NIL))
        NIL))
      NIL))
    NIL))
  NIL))
```

Figure A.3: **PVS Proof of Execution Path Equivalence Theorem**

Appendix B

A Comprehensive Example

This chapter presents a design example that is fully verified by the verification system CCG. The design describes the behavior of a traffic light controller (TLC). The original behavior description of the design in VHDL is given in Figure B.1. Before the synthesis process begins, this description is translated into an intermediate HDL format called Basic Block Intermediate Format as given in Figure B.2. The synthesis tool generates a register transfer level implementation of the behavior in the Basic Block Intermediate Format as given in Figure B.3. Also as a by product of synthesis the binding information between the behavior variables and design registers, between the behavior states and design states, and between the behavior critical paths and design critical paths are produced.

CCG generates formal descriptions of the behavior of TLC, the RTL implementation of TLC, and the binding relations between the two using the specification language of the PVS system. The axioms describing the behavior of the TLC are given in Figure B.6. The data path of the RTL design is described by the axioms given in Figure B.7 and its controller by the axioms given in Figure B.8. Formal descriptions of the behavior of the components used in synthesis of TLC are given in Figure B.9. The set of all lemmas required for proving the correctness of the TLC is given in Figure B.10. Finally, as examples, the proofs of the critical path equivalence lemmas $eq\_cp_0\_st\_lemma$ and $eq\_cp_1\_hl\_lemma$ are given in Figures B.11 and B.12, respectively.

```
-------------------------------------------------------------------------------
--
-- Traffic Light Controller (TLC)
--
-- Source:  Hardware C version  written by David Ku on June 8, 1988 at Stanford
--
-- VHDL Benchmark author Champaka Ramachandran
--                       University Of California, Irvine, CA 92717
--                       champaka@balboa.eng.uci.edu
--
-- Developed on Aug 11, 1992
--
-- Verification Information:
--
--                  Verified     By whom?          Date        Simulator
--                  --------   ------------      --------     ------------
--  Syntax            yes    Champaka Ramachandran  Aug 11, 92    ZYCAD
--  Functionality     yes    Champaka Ramachandran  Aug 11, 92    ZYCAD
--
-- Modified By Naren and Nand Kumar: August 20, 1993
--
-------------------------------------------------------------------------------
entity TLC is
port (
               COF_in: in  std_logic;
               TOL_in : in  std_logic;
               TOS_in : in  std_logic;
               ST_out : out  std_logic;
               HL_out : out  std_logic_vector(1 downto 0);
               FL_out : out  std_logic_vector(1 downto 0));
end TLC;
architecture TLC of TLC is
begin
  process
  variable state: std_logic_vector(1 downto 0);
  variable COF, TOL, TOS: std_logic;
  variable ST:  std_logic;
  variable HL, FL:  std_logic_vector(1 downto 0);
begin
--  wait on COF_in, TOL_in, TOS_in;
  COF := COF_in; TOL := TOL_in; TOS := TOS_in;
  IF (state = "00") THEN
     HL := "10";  FL := "11";
     if (COF = '1') and (TOL = '1') then
        state := "10"; ST := '1';
     else
        state := "00"; ST := '0';
     end if;
  ELSIF (state = "10") THEN
     HL := "01"; FL := "11";
     if (TOS = '1') then
        state := "01"; ST := '1';
     else
        state := "10"; ST := '0';
     end if;
  ELSIF (state = "01") THEN
     HL := "11"; FL := "10";
     if (COF = '0') or (TOL = '1') then
        state :="11"; ST := '1';
     else
        state := "01"; ST := '0';
     end if;
  ELSIF (state = "11") THEN
     HL := "11"; FL := "01";
     if (TOS = '1') then
        state := "00"; ST := '1';
     else
        state := "11"; ST := '0';
     end if;
  END IF;
  ST_out <= ST; HL_out <= HL; FL_out <= FL;
end process ;
end TLC;
```

Figure B.1: **VHDL Behavioral Description of a Traffic Light Controller**

```
% Basic Block Intermediate Format (BBIF)
(SPEC tlc)
(INPORT (cof_in 1) (tol_in 1) (tos_in 1))
(OUTPORT (st_out 1) (hl_out 2) (fl_out 2))
(BLOCKS Blk_1 Blk_2 Blk_3 Blk_4 Blk_5 Blk_6 Blk_7 Blk_8 Blk_9 Blk_10 Blk_11 Blk_12 Blk_13
        Blk_14 Blk_15 Blk_16 Blk_17 Blk_18 Blk_19)
(STARTBLK Blk_2)
(BB Blk_2 ( (state 2) (st 1) (hl 2) (fl 2) )
    (LOCAL (cof 1) (tol 1) (tos 1))
1  (GET_PORT (cof_in) (cof)) ()
2  (GET_PORT (tol_in) (tol)) ()
3  (GET_PORT (tos_in) (tos)) ()
(TRUE_BRANCH Blk_3(state cof tol tos st hl fl))
)
(BB Blk_3 ( (state 2) (cof 1) (tol 1) (tos 1) (st 1) (hl 2) (fl 2) )
    (LOCAL (t18 1))
    (CONSTANT (c17 2 00))
4  (bb_eq (state c17) (t18)) ()
 (t18 Blk_4(cof tol) Blk_7(state cof tol tos st hl fl) Blk_19)
)
(BB Blk_4 ( (cof 1) (tol 1) )
    (LOCAL (hl 2) (fl 2) (t23 1) (t24 1) (t25 1))
    (CONSTANT (c20 2 10) (c21 2 11) (c22 1 1))
6  (TRANSFER (c20) (hl)) ()
7  (TRANSFER (c21) (fl)) ()
8  (bb_eq (cof c22) (t23)) ()
9  (bb_eq (tol c22) (t24)) ()
10  (bb_plus (t23 t24) (t25)) ()
 (t25 Blk_5(hl fl) Blk_6(hl fl) Blk_19)
)
(BB Blk_5 ( (hl 2) (fl 2) )
    (LOCAL (state 2) (st 1))
    (CONSTANT (c20 2 10) (c22 1 1))
11  (TRANSFER (c20) (state)) ()
12  (TRANSFER (c22) (st)) ()
(TRUE_BRANCH Blk_19(state st hl fl))
)
(BB Blk_6 ( (hl 2) (fl 2) )
    (LOCAL (state 2) (st 1))
    (CONSTANT (c17 2 00) (c27 1 0))
13  (TRANSFER (c17) (state)) ()
14  (TRANSFER (c27) (st)) ()
(TRUE_BRANCH Blk_19(state st hl fl))
)
(BB Blk_7 ( (state 2) (cof 1) (tol 1) (tos 1) (st 1) (hl 2) (fl 2) )
    (LOCAL (t28 1))
    (CONSTANT (c20 2 10))
15  (bb_eq (state c20) (t28)) ()
 (t28 Blk_8(tos) Blk_11(state cof tol tos st hl fl) Blk_19)
)
(BB Blk_8 ( (tos 1) )
    (LOCAL (hl 2) (fl 2) (t30 1))
    (CONSTANT (c29 2 01) (c21 2 11) (c22 1 1))
16  (TRANSFER (c29) (hl)) ()
17  (TRANSFER (c21) (fl)) ()
18  (bb_eq (tos c22) (t30)) ()
 (t30 Blk_9(hl fl) Blk_10(hl fl) Blk_19)
)
(BB Blk_9 ( (hl 2) (fl 2) )
    (LOCAL (state 2) (st 1))
    (CONSTANT (c29 2 01) (c22 1 1))
19  (TRANSFER (c29) (state)) ()
20  (TRANSFER (c22) (st)) ()
(TRUE_BRANCH Blk_19(state st hl fl))
)
(BB Blk_10 ( (hl 2) (fl 2) )
    (LOCAL (state 2) (st 1))
    (CONSTANT (c20 2 10) (c27 1 0))
21  (TRANSFER (c20) (state)) ()
22  (TRANSFER (c27) (st)) ()
(TRUE_BRANCH Blk_19(state st hl fl))
)
```

```
(BB Blk_11 ( (state 2) (cof 1) (tol 1) (tos 1) (st 1) (hl 2) (fl 2) )
    (LOCAL (t31 1))
    (CONSTANT (c29 2 01))
23  (bb_eq (state c29) (t31)) ()
 (t31 Blk_12(cof tol) Blk_15(state tos st hl fl) Blk_19)
)
(BB Blk_12 ( (cof 1) (tol 1) )
    (LOCAL (hl 2) (fl 2) (t32 1) (t33 1) (t34 1))
    (CONSTANT (c21 2 11) (c20 2 10) (c27 1 0) (c22 1 1))
24  (TRANSFER (c21) (hl)) ()
25  (TRANSFER (c20) (fl)) ()
26  (bb_eq (cof c27) (t32)) ()
27  (bb_eq (tol c22) (t33)) ()
28  (bb_plus (t32 t33) (t34)) ()
 (t34 Blk_13(hl fl) Blk_14(hl fl) Blk_19)
)
(BB Blk_13 ( (hl 2) (fl 2) )
    (LOCAL (state 2) (st 1))
    (CONSTANT (c21 2 11) (c22 1 1))
29  (TRANSFER (c21) (state)) ()
30  (TRANSFER (c22) (st)) ()
(TRUE_BRANCH Blk_19(state st hl fl))
)
(BB Blk_14 ( (hl 2) (fl 2) )
    (LOCAL (state 2) (st 1))
    (CONSTANT (c29 2 01) (c27 1 0))
31  (TRANSFER (c29) (state)) ()
32  (TRANSFER (c27) (st)) ()
(TRUE_BRANCH Blk_19(state st hl fl))
)
(BB Blk_15 ( (state 2) (tos 1) (st 1) (hl 2) (fl 2) )
    (LOCAL (t35 1))
    (CONSTANT (c21 2 11))
33  (bb_eq (state c21) (t35)) ()
 (t35 Blk_16(tos) Blk_19(state st hl fl) Blk_19)
)
(BB Blk_16 ( (tos 1) )
    (LOCAL (hl 2) (fl 2) (t36 1))
    (CONSTANT (c21 2 11) (c29 2 01) (c22 1 1))
34  (TRANSFER (c21) (hl)) ()
35  (TRANSFER (c29) (fl)) ()
36  (bb_eq (tos c22) (t36)) ()
 (t36 Blk_17(hl fl) Blk_18(hl fl) Blk_19)
)
(BB Blk_17 ( (hl 2) (fl 2) )
    (LOCAL (state 2) (st 1))
    (CONSTANT (c17 2 00) (c22 1 1))
37  (TRANSFER (c17) (state)) ()
38  (TRANSFER (c22) (st)) ()
(TRUE_BRANCH Blk_19(state st hl fl))
)
(BB Blk_18 ( (hl 2) (fl 2) )
    (LOCAL (state 2) (st 1))
    (CONSTANT (c21 2 11) (c27 1 0))
39  (TRANSFER (c21) (state)) ()
40  (TRANSFER (c27) (st)) ()
(TRUE_BRANCH Blk_19(state st hl fl))
)
(BB Blk_19 ( (state 2) (st 1) (hl 2) (fl 2) )
42  (PUT_PORT (st st_out) ()) ()
44  (PUT_PORT (hl hl_out) ()) ()
46  (PUT_PORT (fl fl_out) ()) ()
(TRUE_BRANCH Blk_2(state st hl fl))
)
```

Figure B.2: **Intermediate Behavioral HDL Description of the Traffic Light Controller**

```
% Synthesized RTL Design from Asserta
( DESIGN tlc )
( INPORTS (cof_in 1) (tol_in 1) (tos_in 1) )
( OUTPORTS (st_out 1) (hl_out 2) (fl_out 2) )
( CONTROL_WORD 37 )
( STARTBLK Blk_2_1 )
%%% RTL Component Declarations %%%
% Comp Name                      Instance Name      ParametersControl (pos, size)
asserta_eq                       asserta_eq_0              ( 2 )
asserta_eq                       asserta_eq_1              ( 1 )
asserta_eq                       asserta_eq_2              ( 1 )
asserta_adder                 asserta_adder_3              ( 1 )
asserta_adder                 asserta_adder_4              ( 1 )
asserta_register           asserta_register_1              ( 2 )              ( 0 2 )
asserta_register           asserta_register_2              ( 2 )              ( 2 2 )
asserta_register           asserta_register_3              ( 2 )              ( 4 2 )
asserta_register           asserta_register_4              ( 2 )              ( 6 2 )
asserta_register           asserta_register_5              ( 2 )              ( 8 2 )
asserta_register           asserta_register_6              ( 2 )              ( 10 2 )
asserta_register           asserta_register_7              ( 2 )              ( 12 2 )
asserta_register           asserta_register_8              ( 1 )              ( 14 2 )
asserta_mux                     asserta_mux_1              ( 14 2 3 )              ( 16 3 )
asserta_mux                     asserta_mux_2              ( 14 2 3 )              ( 19 3 )
asserta_mux                     asserta_mux_3              ( 22 2 4 )              ( 22 4 )
asserta_mux                     asserta_mux_4              ( 16 2 3 )              ( 26 3 )
asserta_mux                     asserta_mux_5              ( 6 2 2 )              ( 29 2 )
asserta_mux                     asserta_mux_6              ( 6 2 2 )              ( 31 2 )
asserta_mux                     asserta_mux_7              ( 4 2 1 )              ( 33 1 )
asserta_mux                     asserta_mux_8              ( 8 2 2 )              ( 34 2 )
asserta_mux                     asserta_mux_9              ( 2 1 1 )              ( 36 1 )
%RTL Block Description
 (Blk_2_1
   (% StmtId: 1 Function: GET_PORT
    asserta_mux_5     [2 (PAD_X 1) (cof_in 1) ]    [1 (asserta_mux_5_out1 2) ]   01
    asserta_register_5     [1 (asserta_mux_5_out1 2) ]    [1 (asserta_register_5_out1 2) ]   01
   )
   (% StmtId: 2 Function: GET_PORT
    asserta_mux_6     [2 (PAD_X 1) (tol_in 1) ]    [1 (asserta_mux_6_out1 2) ]   01
    asserta_register_6     [1 (asserta_mux_6_out1 2) ]    [1 (asserta_register_6_out1 2) ]   01
   )
   (% StmtId: 3 Function: GET_PORT
    asserta_mux_7     [1 (PAD_X 1) (tos_in 1) ]    [1 (asserta_mux_7_out1 2) ]   0
    asserta_register_7     [1 (asserta_mux_7_out1 2) ]    [1 (asserta_register_7_out1 2) ]   01
   )
 )
 ( BRANCH
   (TRUE (Blk_2_Blk_3_0 ))
 )
 (Blk_2_Blk_3_0
   (% StmtId: branch Function: actual-formal interconnect
    asserta_mux_2     [4 (asserta_register_5_out1 2) ]    [1 (asserta_mux_2_out1 2) ]   011
    asserta_register_2     [1 (asserta_mux_2_out1 2) ]    [1 (asserta_register_2_out1 2) ]   01
   )
   (% StmtId: branch Function: actual-formal interconnect
    asserta_mux_3     [4 (asserta_register_6_out1 2) ]    [1 (asserta_mux_3_out1 2) ]   0011
    asserta_register_3     [1 (asserta_mux_3_out1 2) ]    [1 (asserta_register_3_out1 2) ]   01
   )
   (% StmtId: branch Function: actual-formal interconnect
    asserta_mux_4     [2 (asserta_register_7_out1 2) ]    [1 (asserta_mux_4_out1 2) ]   001
    asserta_register_4     [1 (asserta_mux_4_out1 2) ]    [1 (asserta_register_4_out1 2) ]   01
   )
   (% StmtId: branch Function: actual-formal interconnect
    asserta_mux_5     [3 (asserta_register_2_out1 2) ]    [1 (asserta_mux_5_out1 2) ]   10
    asserta_register_5     [1 (asserta_mux_5_out1 2) ]    [1 (asserta_register_5_out1 2) ]   01
   )
   (% StmtId: branch Function: actual-formal interconnect
    asserta_mux_6     [3 (asserta_register_3_out1 2) ]    [1 (asserta_mux_6_out1 2) ]   10
    asserta_register_6     [1 (asserta_mux_6_out1 2) ]    [1 (asserta_register_6_out1 2) ]   01
   )
```

```
    (% StmtId: branch Function: actual-formal interconnect
     asserta_mux_7     [2 (asserta_register_4_out1 2) ]    [1 (asserta_mux_7_out1 2) ]   1
     asserta_register_7     [1 (asserta_mux_7_out1 2) ]    [1 (asserta_register_7_out1 2) ]   01
    )
  )
  ( BRANCH
    (TRUE (Blk_3_1 ))
  )
  (Blk_3_1
    (% StmtId: 4 Function: bb_eq
     asserta_mux_8     [1 (00 2) ]    [1 (asserta_mux_8_out1 2) ]   00
     asserta_eq_0     [1 (asserta_register_1_out1 2) ]  [2 (asserta_mux_8_out1 2) ]    [1 (asserta_eq_
0_out1 1) ]
     asserta_register_8     [1 (asserta_eq_0_out1 1) ]    [1 (asserta_register_8_out1 1) ]   01
    )
  )
  ( BRANCH
    (asserta_register_8_out1 (Blk_3_Blk_4_0 )(Blk_7_1 ))
  )
  (Blk_3_Blk_4_0
    (% StmtId: branch Function: actual-formal interconnect
     asserta_mux_1     [4 (asserta_register_2_out1 2) ]    [1 (asserta_mux_1_out1 2) ]   011
     asserta_register_1     [1 (asserta_mux_1_out1 2) ]    [1 (asserta_register_1_out1 2) ]   01
    )
    (% StmtId: branch Function: actual-formal interconnect
     asserta_mux_2     [1 (asserta_register_3_out1 2) ]    [1 (asserta_mux_2_out1 2) ]   000
     asserta_register_2     [1 (asserta_mux_2_out1 2) ]    [1 (asserta_register_2_out1 2) ]   01
    )
  )
  ( BRANCH
    (TRUE (Blk_4_1 ))
  )
  (Blk_4_1
    (% StmtId: 6 Function: TRANSFER
     asserta_mux_1     [5 (10 2) ]    [1 (asserta_mux_1_out1 2) ]   100
     asserta_register_1     [1 (asserta_mux_1_out1 2) ]    [1 (asserta_register_1_out1 2) ]   01
    )
    (% StmtId: 7 Function: TRANSFER
     asserta_mux_2     [2 (11 2) ]    [1 (asserta_mux_2_out1 2) ]   001
     asserta_register_2     [1 (asserta_mux_2_out1 2) ]    [1 (asserta_register_2_out1 2) ]   01
    )
    (% StmtId: 8 Function: bb_eq
     asserta_mux_9     [1 (1 1) ]    [1 (asserta_mux_9_out1 1) ]   0
     asserta_eq_1     [1 (asserta_register_1_out1 1) ]  [2 (asserta_mux_9_out1 1) ]    [1 (asserta_eq_
1_out1 1) ]
     asserta_mux_3     [2 (PAD_X 1) (asserta_eq_1_out1 1) ]    [1 (asserta_mux_3_out1 2) ]   0001
     asserta_register_3     [1 (asserta_mux_3_out1 2) ]    [1 (asserta_register_3_out1 2) ]   01
    )
    (% StmtId: 9 Function: bb_eq
     asserta_eq_2     [1 (asserta_register_2_out1 1) ]  [2 (1 1) ]    [1 (asserta_eq_2_out1 1) ]
     asserta_mux_4     [3 (PAD_X 1) (asserta_eq_2_out1 1) ]    [1 (asserta_mux_4_out1 2) ]   010
     asserta_register_4     [1 (asserta_mux_4_out1 2) ]    [1 (asserta_register_4_out1 2) ]   01
    )
  )
  ( BRANCH
    (TRUE (Blk_4_2 ))
  )
  (Blk_4_2
    (% StmtId: 10 Function: bb_and
     asserta_adder_3     [1 (asserta_register_3_out1 1) ]  [2 (asserta_register_4_out1 1) ]    [1 (as
serta_adder_3_out1 1) ]
     asserta_mux_3     [1 (PAD_X 1) (asserta_adder_3_out1 1) ]    [1 (asserta_mux_3_out1 2) ]   0000
     asserta_register_3     [1 (asserta_mux_3_out1 2) ]    [1 (asserta_register_3_out1 2) ]   01
    )
  )
  ( BRANCH
    (asserta_register_3_out1 (Blk_5_1 )(Blk_6_1 ))
  )
  (Blk_5_1
    (% StmtId: 11 Function: TRANSFER
     asserta_mux_3     [5 (10 2) ]    [1 (asserta_mux_3_out1 2) ]   0100
     asserta_register_3     [1 (asserta_mux_3_out1 2) ]    [1 (asserta_register_3_out1 2) ]   01
    )
    (% StmtId: 12 Function: TRANSFER
     asserta_mux_4     [4 (PAD_X 1) (1 1) ]    [1 (asserta_mux_4_out1 2) ]   011
    asserta_register_4     [1 (asserta_mux_4_out1 2) ]    [1 (asserta_register_4_out1 2) ]   01
   )
  )
  ( BRANCH
   (TRUE (Blk_5_Blk_19_0 ))
  )
  (Blk_5_Blk_19_0
   (% StmtId: branch Function: actual-formal interconnect
    asserta_mux_1     [7 (asserta_register_3_out1 2) ]    [1 (asserta_mux_1_out1 2) ]   110
    asserta_register_1     [1 (asserta_mux_1_out1 2) ]    [1 (asserta_register_1_out1 2) ]   01
   )
   (% StmtId: branch Function: actual-formal interconnect
    asserta_mux_2     [6 (asserta_register_4_out1 2) ]    [1 (asserta_mux_2_out1 2) ]   101
    asserta_register_2     [1 (asserta_mux_2_out1 2) ]    [1 (asserta_register_2_out1 2) ]   01
   )
   (% StmtId: branch Function: actual-formal interconnect
    asserta_mux_3     [10 (asserta_register_1_out1 2) ]    [1 (asserta_mux_3_out1 2) ]   1001
    asserta_register_3     [1 (asserta_mux_3_out1 2) ]    [1 (asserta_register_3_out1 2) ]   01
   )
   (% StmtId: branch Function: actual-formal interconnect
    asserta_mux_4     [7 (asserta_register_2_out1 2) ]    [1 (asserta_mux_4_out1 2) ]   110
    asserta_register_4     [1 (asserta_mux_4_out1 2) ]    [1 (asserta_register_4_out1 2) ]   01
   )
  )
  ( BRANCH
   (TRUE (Blk_19_1 ))
  )
  (Blk_6_1
   (% StmtId: 13 Function: TRANSFER
    asserta_mux_3     [3 (00 2) ]    [1 (asserta_mux_3_out1 2) ]   0010
    asserta_register_3     [1 (asserta_mux_3_out1 2) ]    [1 (asserta_register_3_out1 2) ]   01
   )
   (% StmtId: 14 Function: TRANSFER
    asserta_mux_4     [5 (PAD_X 1) (0 1) ]    [1 (asserta_mux_4_out1 2) ]   100
    asserta_register_4     [1 (asserta_mux_4_out1 2) ]    [1 (asserta_register_4_out1 2) ]   01
   )
  )
  ( BRANCH
   (TRUE (Blk_6_Blk_19_0 ))
  )
  (Blk_6_Blk_19_0
   (% StmtId: branch Function: actual-formal interconnect
    asserta_mux_1     [7 (asserta_register_3_out1 2) ]    [1 (asserta_mux_1_out1 2) ]   110
    asserta_register_1     [1 (asserta_mux_1_out1 2) ]    [1 (asserta_register_1_out1 2) ]   01
   )
   (% StmtId: branch Function: actual-formal interconnect
    asserta_mux_2     [6 (asserta_register_4_out1 2) ]    [1 (asserta_mux_2_out1 2) ]   101
    asserta_register_2     [1 (asserta_mux_2_out1 2) ]    [1 (asserta_register_2_out1 2) ]   01
   )
   (% StmtId: branch Function: actual-formal interconnect
    asserta_mux_3     [10 (asserta_register_1_out1 2) ]    [1 (asserta_mux_3_out1 2) ]   1001
    asserta_register_3     [1 (asserta_mux_3_out1 2) ]    [1 (asserta_register_3_out1 2) ]   01
   )
   (% StmtId: branch Function: actual-formal interconnect
    asserta_mux_4     [7 (asserta_register_2_out1 2) ]    [1 (asserta_mux_4_out1 2) ]   110
    asserta_register_4     [1 (asserta_mux_4_out1 2) ]    [1 (asserta_register_4_out1 2) ]   01
   )
  )
  ( BRANCH
   (TRUE (Blk_19_1 ))
  )
  (Blk_7_1
   (% StmtId: 15 Function: bb_eq
    asserta_mux_8     [2 (10 2) ]    [1 (asserta_mux_8_out1 2) ]   01
    asserta_eq_0     [1 (asserta_register_1_out1 2) ]  [2 (asserta_mux_8_out1 2) ]    [1 (asserta_eq_
0_out1 1) ]
    asserta_register_8     [1 (asserta_eq_0_out1 1) ]    [1 (asserta_register_8_out1 1) ]   01
   )
  )
  ( BRANCH
   (asserta_register_8_out1 (Blk_7_Blk_8_0 )(Blk_11_1 ))
  )
  (Blk_7_Blk_8_0
   (% StmtId: branch Function: actual-formal interconnect
    asserta_mux_1     [1 (asserta_register_4_out1 2) ]    [1 (asserta_mux_1_out1 2) ]   000
    asserta_register_1     [1 (asserta_mux_1_out1 2) ]    [1 (asserta_register_1_out1 2) ]   01
   )
  )
  ( BRANCH
   (TRUE (Blk_8_1 ))
  )
  (Blk_8_1
   (% StmtId: 16 Function: TRANSFER
    asserta_mux_1     [2 (01 2) ]    [1 (asserta_mux_1_out1 2) ]   001
    asserta_register_1     [1 (asserta_mux_1_out1 2) ]    [1 (asserta_register_1_out1 2) ]   01
   )
   (% StmtId: 17 Function: TRANSFER
    asserta_mux_2     [2 (11 2) ]    [1 (asserta_mux_2_out1 2) ]   001
    asserta_register_2     [1 (asserta_mux_2_out1 2) ]    [1 (asserta_register_2_out1 2) ]   01
   )
   (% StmtId: 18 Function: bb_eq
    asserta_mux_9     [1 (1 1) ]    [1 (asserta_mux_9_out1 1) ]   0
    asserta_eq_1     [1 (asserta_register_1_out1 1) ]  [2 (asserta_mux_9_out1 1) ]    [1 (asserta_eq_
1_out1 1) ]
    asserta_mux_3     [2 (PAD_X 1) (asserta_eq_1_out1 1) ]    [1 (asserta_mux_3_out1 2) ]   0001
    asserta_register_3     [1 (asserta_mux_3_out1 2) ]    [1 (asserta_register_3_out1 2) ]   01
   )
  )
  ( BRANCH
   (asserta_register_3_out1 (Blk_9_1 )(Blk_10_1 ))
  )
  (Blk_9_1
   (% StmtId: 19 Function: TRANSFER
    asserta_mux_3     [6 (01 2) ]    [1 (asserta_mux_3_out1 2) ]   0101
    asserta_register_3     [1 (asserta_mux_3_out1 2) ]    [1 (asserta_register_3_out1 2) ]   01
   )
   (% StmtId: 20 Function: TRANSFER
    asserta_mux_4     [4 (PAD_X 1) (1 1) ]    [1 (asserta_mux_4_out1 2) ]   011
    asserta_register_4     [1 (asserta_mux_4_out1 2) ]    [1 (asserta_register_4_out1 2) ]   01
   )
  )
  ( BRANCH
   (TRUE (Blk_9_Blk_19_0 ))
  )
  (Blk_9_Blk_19_0
   (% StmtId: branch Function: actual-formal interconnect
    asserta_mux_1     [7 (asserta_register_3_out1 2) ]    [1 (asserta_mux_1_out1 2) ]   110
    asserta_register_1     [1 (asserta_mux_1_out1 2) ]    [1 (asserta_register_1_out1 2) ]   01
   )
   (% StmtId: branch Function: actual-formal interconnect
    asserta_mux_2     [6 (asserta_register_4_out1 2) ]    [1 (asserta_mux_2_out1 2) ]   101
    asserta_register_2     [1 (asserta_mux_2_out1 2) ]    [1 (asserta_register_2_out1 2) ]   01
   )
   (% StmtId: branch Function: actual-formal interconnect
    asserta_mux_3     [10 (asserta_register_1_out1 2) ]    [1 (asserta_mux_3_out1 2) ]   1001
    asserta_register_3     [1 (asserta_mux_3_out1 2) ]    [1 (asserta_register_3_out1 2) ]   01
   )
   (% StmtId: branch Function: actual-formal interconnect
    asserta_mux_4     [7 (asserta_register_2_out1 2) ]    [1 (asserta_mux_4_out1 2) ]   110
    asserta_register_4     [1 (asserta_mux_4_out1 2) ]    [1 (asserta_register_4_out1 2) ]   01
   )
  )
  ( BRANCH
   (TRUE (Blk_19_1 ))
  )
  (Blk_10_1
   (% StmtId: 21 Function: TRANSFER
    asserta_mux_3     [5 (10 2) ]    [1 (asserta_mux_3_out1 2) ]   0100
    asserta_register_3     [1 (asserta_mux_3_out1 2) ]    [1 (asserta_register_3_out1 2) ]   01
   )
   (% StmtId: 22 Function: TRANSFER
    asserta_mux_4     [5 (PAD_X 1) (0 1) ]    [1 (asserta_mux_4_out1 2) ]   100
    asserta_register_4     [1 (asserta_mux_4_out1 2) ]    [1 (asserta_register_4_out1 2) ]   01
   )
  )
  ( BRANCH
   (TRUE (Blk_10_Blk_19_0 ))
  )
  (Blk_10_Blk_19_0
   (% StmtId: branch Function: actual-formal interconnect
    asserta_mux_1     [7 (asserta_register_3_out1 2) ]    [1 (asserta_mux_1_out1 2) ]   110
    asserta_register_1     [1 (asserta_mux_1_out1 2) ]    [1 (asserta_register_1_out1 2) ]   01
   )
   (% StmtId: branch Function: actual-formal interconnect
    asserta_mux_2     [6 (asserta_register_4_out1 2) ]    [1 (asserta_mux_2_out1 2) ]   101
    asserta_register_2     [1 (asserta_mux_2_out1 2) ]    [1 (asserta_register_2_out1 2) ]   01
   )
   (% StmtId: branch Function: actual-formal interconnect
    asserta_mux_3     [10 (asserta_register_1_out1 2) ]    [1 (asserta_mux_3_out1 2) ]   1001
    asserta_register_3     [1 (asserta_mux_3_out1 2) ]    [1 (asserta_register_3_out1 2) ]   01
   )
   (% StmtId: branch Function: actual-formal interconnect
    asserta_mux_4     [7 (asserta_register_2_out1 2) ]    [1 (asserta_mux_4_out1 2) ]   110
    asserta_register_4     [1 (asserta_mux_4_out1 2) ]    [1 (asserta_register_4_out1 2) ]   01
   )
 )
 ( BRANCH
   (TRUE (Blk_19_1 ))
 )
 (Blk_11_1
   (% StmtId: 23 Function: bb_eq
    asserta_mux_8     [3 (01 2) ]    [1 (asserta_mux_8_out1 2) ]   10
    asserta_eq_0     [1 (asserta_register_1_out1 2) ]  [2 (asserta_mux_8_out1 2) ]    [1 (asserta_eq
_0_out1 1) ]
    asserta_register_8     [1 (asserta_eq_0_out1 1) ]    [1 (asserta_register_8_out1 1) ]   01
   )
 )
 ( BRANCH
   (asserta_register_8_out1 (Blk_11_Blk_12_0 )(Blk_11_Blk_15_0 ))
 )
 (Blk_11_Blk_12_0
   (% StmtId: branch Function: actual-formal interconnect
    asserta_mux_1     [4 (asserta_register_2_out1 2) ]    [1 (asserta_mux_1_out1 2) ]   011
    asserta_register_1     [1 (asserta_mux_1_out1 2) ]    [1 (asserta_register_1_out1 2) ]   01
   )
   (% StmtId: branch Function: actual-formal interconnect
    asserta_mux_2     [1 (asserta_register_3_out1 2) ]    [1 (asserta_mux_2_out1 2) ]   000
    asserta_register_2     [1 (asserta_mux_2_out1 2) ]    [1 (asserta_register_2_out1 2) ]   01
   )
 )
 ( BRANCH
   (TRUE (Blk_12_1 ))
 )
 (Blk_11_Blk_15_0
   (% StmtId: branch Function: actual-formal interconnect
    asserta_mux_2     [6 (asserta_register_4_out1 2) ]    [1 (asserta_mux_2_out1 2) ]   101
    asserta_register_2     [1 (asserta_mux_2_out1 2) ]    [1 (asserta_register_2_out1 2) ]   01
   )
   (% StmtId: branch Function: actual-formal interconnect
    asserta_mux_3     [9 (asserta_register_5_out1 2) ]    [1 (asserta_mux_3_out1 2) ]   1000
    asserta_register_3     [1 (asserta_mux_3_out1 2) ]    [1 (asserta_register_3_out1 2) ]   01
   )
   (% StmtId: branch Function: actual-formal interconnect
    asserta_mux_4     [6 (asserta_register_6_out1 2) ]    [1 (asserta_mux_4_out1 2) ]   101
    asserta_register_4     [1 (asserta_mux_4_out1 2) ]    [1 (asserta_register_4_out1 2) ]   01
   )
   (% StmtId: branch Function: actual-formal interconnect
    asserta_mux_5     [1 (asserta_register_7_out1 2) ]    [1 (asserta_mux_5_out1 2) ]   00
    asserta_register_5     [1 (asserta_mux_5_out1 2) ]    [1 (asserta_register_5_out1 2) ]   01
   )
 )
 ( BRANCH
   (TRUE (Blk_15_1 ))
 )
 (Blk_12_1
   (% StmtId: 24 Function: TRANSFER
    asserta_mux_1     [6 (11 2) ]    [1 (asserta_mux_1_out1 2) ]   101
    asserta_register_1     [1 (asserta_mux_1_out1 2) ]    [1 (asserta_register_1_out1 2) ]   01
   )
   (% StmtId: 25 Function: TRANSFER
    asserta_mux_2     [5 (10 2) ]    [1 (asserta_mux_2_out1 2) ]   100
    asserta_register_2     [1 (asserta_mux_2_out1 2) ]    [1 (asserta_register_2_out1 2) ]   01
   )
```

```
    (% StmtId: 26 Function: bb_eq
     asserta_mux_9     [2 (0 1) ]    [1 (asserta_mux_9_out1 1) ]   1
     asserta_eq_1     [1 (asserta_register_1_out1 1) ]  [2 (asserta_mux_9_out1 1) ]    [1 (asserta_eq
_1_out1 1) ]
     asserta_mux_3     [2 (PAD_X 1) (asserta_eq_1_out1 1) ]    [1 (asserta_mux_3_out1 2) ]   0001
     asserta_register_3     [1 (asserta_mux_3_out1 2) ]    [1 (asserta_register_3_out1 2) ]   01
    )
    (% StmtId: 27 Function: bb_eq
     asserta_eq_2     [1 (asserta_register_2_out1 1) ]  [2 (1 1) ]    [1 (asserta_eq_2_out1 1) ]
     asserta_mux_4     [3 (PAD_X 1) (asserta_eq_2_out1 1) ]    [1 (asserta_mux_4_out1 2) ]   010
     asserta_register_4     [1 (asserta_mux_4_out1 2) ]    [1 (asserta_register_4_out1 2) ]   01
    )
  )
  ( BRANCH
    (TRUE (Blk_12_2 ))
  )
  (Blk_12_2
    (% StmtId: 28 Function: bb_or
     asserta_adder_4     [1 (asserta_register_3_out1 1) ]  [2 (asserta_register_4_out1 1) ]    [1 (as
serta_adder_4_out1 1) ]
     asserta_mux_3     [7 (PAD_X 1) (asserta_adder_4_out1 1) ]    [1 (asserta_mux_3_out1 2) ]   0110
     asserta_register_3     [1 (asserta_mux_3_out1 2) ]    [1 (asserta_register_3_out1 2) ]   01
    )
  )
  ( BRANCH
    (asserta_register_3_out1 (Blk_13_1 )(Blk_14_1 ))
  )
  (Blk_13_1
    (% StmtId: 29 Function: TRANSFER
     asserta_mux_3     [8 (11 2) ]    [1 (asserta_mux_3_out1 2) ]   0111
     asserta_register_3     [1 (asserta_mux_3_out1 2) ]    [1 (asserta_register_3_out1 2) ]   01
    )
    (% StmtId: 30 Function: TRANSFER
     asserta_mux_4     [4 (PAD_X 1) (1 1) ]    [1 (asserta_mux_4_out1 2) ]   011
     asserta_register_4     [1 (asserta_mux_4_out1 2) ]    [1 (asserta_register_4_out1 2) ]   01
    )
  )
  ( BRANCH
    (TRUE (Blk_13_Blk_19_0 ))
  )
  (Blk_13_Blk_19_0
    (% StmtId: branch Function: actual-formal interconnect
     asserta_mux_1     [7 (asserta_register_3_out1 2) ]    [1 (asserta_mux_1_out1 2) ]   110
     asserta_register_1     [1 (asserta_mux_1_out1 2) ]    [1 (asserta_register_1_out1 2) ]   01
    )
    (% StmtId: branch Function: actual-formal interconnect
     asserta_mux_2     [6 (asserta_register_4_out1 2) ]    [1 (asserta_mux_2_out1 2) ]   101
     asserta_register_2     [1 (asserta_mux_2_out1 2) ]    [1 (asserta_register_2_out1 2) ]   01
    )
    (% StmtId: branch Function: actual-formal interconnect
     asserta_mux_3     [10 (asserta_register_1_out1 2) ]    [1 (asserta_mux_3_out1 2) ]   1001
     asserta_register_3     [1 (asserta_mux_3_out1 2) ]    [1 (asserta_register_3_out1 2) ]   01
    )
    (% StmtId: branch Function: actual-formal interconnect
     asserta_mux_4     [7 (asserta_register_2_out1 2) ]    [1 (asserta_mux_4_out1 2) ]   110
     asserta_register_4     [1 (asserta_mux_4_out1 2) ]    [1 (asserta_register_4_out1 2) ]   01
    )
  )
  ( BRANCH
    (TRUE (Blk_19_1 ))
  )
  (Blk_14_1
    (% StmtId: 31 Function: TRANSFER
     asserta_mux_3     [6 (01 2) ]    [1 (asserta_mux_3_out1 2) ]   0101
     asserta_register_3     [1 (asserta_mux_3_out1 2) ]    [1 (asserta_register_3_out1 2) ]   01
    )
    (% StmtId: 32 Function: TRANSFER
     asserta_mux_4     [5 (PAD_X 1) (0 1) ]    [1 (asserta_mux_4_out1 2) ]   100
     asserta_register_4     [1 (asserta_mux_4_out1 2) ]    [1 (asserta_register_4_out1 2) ]   01
    )
  )
  ( BRANCH
    (TRUE (Blk_14_Blk_19_0 ))
  )
```

```
 (Blk_14_Blk_19_0
   (% StmtId: branch Function: actual-formal interconnect
    asserta_mux_1     [7 (asserta_register_3_out1 2) ]    [1 (asserta_mux_1_out1 2) ]   110
    asserta_register_1     [1 (asserta_mux_1_out1 2) ]    [1 (asserta_register_1_out1 2) ]   01
   )
   (% StmtId: branch Function: actual-formal interconnect
    asserta_mux_2     [6 (asserta_register_4_out1 2) ]    [1 (asserta_mux_2_out1 2) ]   101
    asserta_register_2     [1 (asserta_mux_2_out1 2) ]    [1 (asserta_register_2_out1 2) ]   01
   )
   (% StmtId: branch Function: actual-formal interconnect
    asserta_mux_3     [10 (asserta_register_1_out1 2) ]    [1 (asserta_mux_3_out1 2) ]   1001
    asserta_register_3     [1 (asserta_mux_3_out1 2) ]    [1 (asserta_register_3_out1 2) ]   01
   )
   (% StmtId: branch Function: actual-formal interconnect
    asserta_mux_4     [7 (asserta_register_2_out1 2) ]    [1 (asserta_mux_4_out1 2) ]   110
    asserta_register_4     [1 (asserta_mux_4_out1 2) ]    [1 (asserta_register_4_out1 2) ]   01
   )
 )
 ( BRANCH
   (TRUE (Blk_19_1 ))
 )
 (Blk_15_1
   (% StmtId: 33 Function: bb_eq
    asserta_mux_8     [4 (11 2) ]    [1 (asserta_mux_8_out1 2) ]   11
    asserta_eq_0     [1 (asserta_register_1_out1 2) ]  [2 (asserta_mux_8_out1 2) ]    [1 (as
serta_eq_0_out1 1) ]
    asserta_mux_6     [1 (PAD_X 1) (asserta_eq_0_out1 1) ]    [1 (asserta_mux_6_out1 2) ]   00
    asserta_register_6     [1 (asserta_mux_6_out1 2) ]    [1 (asserta_register_6_out1 2) ]   01
   )
 )
 ( BRANCH
   (asserta_register_6_out1 (Blk_15_Blk_16_0 )(Blk_15_Blk_19_0 ))
 )
 (Blk_15_Blk_16_0
   (% StmtId: branch Function: actual-formal interconnect
    asserta_mux_1     [4 (asserta_register_2_out1 2) ]    [1 (asserta_mux_1_out1 2) ]   011
    asserta_register_1     [1 (asserta_mux_1_out1 2) ]    [1 (asserta_register_1_out1 2) ]   01
   )
 )
 ( BRANCH
   (TRUE (Blk_16_1 ))
 )
 (Blk_15_Blk_19_0
   (% StmtId: branch Function: actual-formal interconnect
    asserta_mux_2     [1 (asserta_register_3_out1 2) ]    [1 (asserta_mux_2_out1 2) ]   000
    asserta_register_2     [1 (asserta_mux_2_out1 2) ]    [1 (asserta_register_2_out1 2) ]   01
   )
   (% StmtId: branch Function: actual-formal interconnect
    asserta_mux_3     [11 (asserta_register_4_out1 2) ]    [1 (asserta_mux_3_out1 2) ]   1010
    asserta_register_3     [1 (asserta_mux_3_out1 2) ]    [1 (asserta_register_3_out1 2) ]   01
   )
   (% StmtId: branch Function: actual-formal interconnect
    asserta_mux_4     [8 (asserta_register_5_out1 2) ]    [1 (asserta_mux_4_out1 2) ]   111
    asserta_register_4     [1 (asserta_mux_4_out1 2) ]    [1 (asserta_register_4_out1 2) ]   01
   )
 )
 ( BRANCH
   (TRUE (Blk_19_1 ))
 )
 (Blk_16_1
   (% StmtId: 34 Function: TRANSFER
    asserta_mux_1     [6 (11 2) ]    [1 (asserta_mux_1_out1 2) ]   101
    asserta_register_1     [1 (asserta_mux_1_out1 2) ]    [1 (asserta_register_1_out1 2) ]   01
   )
   (% StmtId: 35 Function: TRANSFER
    asserta_mux_2     [7 (01 2) ]    [1 (asserta_mux_2_out1 2) ]   110
    asserta_register_2     [1 (asserta_mux_2_out1 2) ]    [1 (asserta_register_2_out1 2) ]   01
   )
   (% StmtId: 36 Function: bb_eq
    asserta_mux_9     [1 (1 1) ]    [1 (asserta_mux_9_out1 1) ]   0
    asserta_eq_1     [1 (asserta_register_1_out1 1) ]  [2 (asserta_mux_9_out1 1) ]    [1 (asserta_eq_
1_out1 1) ]
    asserta_mux_3     [2 (PAD_X 1) (asserta_eq_1_out1 1) ]    [1 (asserta_mux_3_out1 2) ]   0001
    asserta_register_3     [1 (asserta_mux_3_out1 2) ]    [1 (asserta_register_3_out1 2) ]   01
   )
 )
```

```
( BRANCH
  (asserta_register_3_out1 (Blk_17_1 )(Blk_18_1 ))
)
(Blk_17_1
  (% StmtId: 37 Function: TRANSFER
   asserta_mux_3     [3 (00 2) ]    [1 (asserta_mux_3_out1 2) ]   0010
   asserta_register_3     [1 (asserta_mux_3_out1 2) ]    [1 (asserta_register_3_out1 2) ]   01
  )
  (% StmtId: 38 Function: TRANSFER
   asserta_mux_4     [4 (PAD_X 1) (1 1) ]    [1 (asserta_mux_4_out1 2) ]   011
   asserta_register_4     [1 (asserta_mux_4_out1 2) ]    [1 (asserta_register_4_out1 2) ]   01
  )
)
( BRANCH
  (TRUE (Blk_17_Blk_19_0 ))
 )
(Blk_17_Blk_19_0
  (% StmtId: branch Function: actual-formal interconnect
   asserta_mux_1     [7 (asserta_register_3_out1 2) ]    [1 (asserta_mux_1_out1 2) ]   110
   asserta_register_1     [1 (asserta_mux_1_out1 2) ]    [1 (asserta_register_1_out1 2) ]   01
  )
  (% StmtId: branch Function: actual-formal interconnect
   asserta_mux_2     [6 (asserta_register_4_out1 2) ]    [1 (asserta_mux_2_out1 2) ]   101
   asserta_register_2     [1 (asserta_mux_2_out1 2) ]    [1 (asserta_register_2_out1 2) ]   01
  )
  (% StmtId: branch Function: actual-formal interconnect
   asserta_mux_3     [10 (asserta_register_1_out1 2) ]    [1 (asserta_mux_3_out1 2) ]   1001
   asserta_register_3     [1 (asserta_mux_3_out1 2) ]    [1 (asserta_register_3_out1 2) ]   01
  )
  (% StmtId: branch Function: actual-formal interconnect
   asserta_mux_4     [7 (asserta_register_2_out1 2) ]    [1 (asserta_mux_4_out1 2) ]   110
   asserta_register_4     [1 (asserta_mux_4_out1 2) ]    [1 (asserta_register_4_out1 2) ]   01
  )
)
( BRANCH
  (TRUE (Blk_19_1 ))
)
(Blk_18_1
  (% StmtId: 39 Function: TRANSFER
   asserta_mux_3     [8 (11 2) ]    [1 (asserta_mux_3_out1 2) ]   0111
   asserta_register_3     [1 (asserta_mux_3_out1 2) ]    [1 (asserta_register_3_out1 2) ]   01
  )
  (% StmtId: 40 Function: TRANSFER
   asserta_mux_4     [5 (PAD_X 1) (0 1) ]    [1 (asserta_mux_4_out1 2) ]   100
   asserta_register_4     [1 (asserta_mux_4_out1 2) ]    [1 (asserta_register_4_out1 2) ]   01
  )
)
( BRANCH
  (TRUE (Blk_18_Blk_19_0 ))
)
(Blk_18_Blk_19_0
  (% StmtId: branch Function: actual-formal interconnect
   asserta_mux_1     [7 (asserta_register_3_out1 2) ]    [1 (asserta_mux_1_out1 2) ]   110
   asserta_register_1     [1 (asserta_mux_1_out1 2) ]    [1 (asserta_register_1_out1 2) ]   01
  )
  (% StmtId: branch Function: actual-formal interconnect
   asserta_mux_2     [6 (asserta_register_4_out1 2) ]    [1 (asserta_mux_2_out1 2) ]   101
   asserta_register_2     [1 (asserta_mux_2_out1 2) ]    [1 (asserta_register_2_out1 2) ]   01
  )
  (% StmtId: branch Function: actual-formal interconnect
   asserta_mux_3     [10 (asserta_register_1_out1 2) ]    [1 (asserta_mux_3_out1 2) ]   1001
   asserta_register_3     [1 (asserta_mux_3_out1 2) ]    [1 (asserta_register_3_out1 2) ]   01
  )
  (% StmtId: branch Function: actual-formal interconnect
   asserta_mux_4     [7 (asserta_register_2_out1 2) ]    [1 (asserta_mux_4_out1 2) ]   110
   asserta_register_4     [1 (asserta_mux_4_out1 2) ]    [1 (asserta_register_4_out1 2) ]   01
  )
)
( BRANCH
  (TRUE (Blk_19_1 ))
)
```

```
(Blk_19_1
  (% StmtId: 42 Function: PUT_PORT
   WIRE     [1 (asserta_register_2_out1 1) ]    [1 (st_out 1) ]
  )
  (% StmtId: 44 Function: PUT_PORT
   WIRE     [1 (asserta_register_3_out1 2) ]    [1 (hl_out 2) ]
  )
  (% StmtId: 46 Function: PUT_PORT
   WIRE     [1 (asserta_register_4_out1 2) ]    [1 (fl_out 2) ]
  )
)
( BRANCH
  (TRUE (Blk_2_1 ))
)
```

Figure B.3: **Intermediate Structural HDL Description of the Traffic Light Controller**

```
(Blk_2_1
  (asserta_register_5 cof)
  (asserta_register_6 tol)
  (asserta_register_7 tos)
)
(Blk_2_Blk_3_0
  (asserta_register_2 cof)
  (asserta_register_3 tol)
  (asserta_register_4 tos)
  (asserta_register_5 st)
  (asserta_register_6 hl)
  (asserta_register_7 fl)
)
(Blk_3_1
  (asserta_register_8 t18)
)
(Blk_3_Blk_4_0
  (asserta_register_1 cof)
  (asserta_register_2 tol)
)
(Blk_4_1
  (asserta_register_1 hl)
  (asserta_register_2 fl)
  (asserta_register_3 t23)
  (asserta_register_4 t24)
)
(Blk_4_2
  (asserta_register_3 t25)
)
(Blk_5_1
  (asserta_register_3 state)
  (asserta_register_4 st)
)
(Blk_5_Blk_19_0
  (asserta_register_1 state)
  (asserta_register_2 st)
  (asserta_register_3 hl)
  (asserta_register_4 fl)
)
(Blk_6_1
  (asserta_register_3 state)
  (asserta_register_4 st)
)
(Blk_6_Blk_19_0
  (asserta_register_1 state)
  (asserta_register_2 st)
  (asserta_register_3 hl)
  (asserta_register_4 fl)
)
(Blk_7_1
   (asserta_register_8 t28)
)
(Blk_7_Blk_8_0
  (asserta_register_1 tos)
)
(Blk_8_1
  (asserta_register_1 hl)
  (asserta_register_2 fl)
  (asserta_register_3 t30)
)
(Blk_9_1
  (asserta_register_3 state)
  (asserta_register_4 st)
)
(Blk_9_Blk_19_0
  (asserta_register_1 state)
  (asserta_register_2 st)
  (asserta_register_3 hl)
  (asserta_register_4 fl)
)
```

```
(Blk_10_1
  (asserta_register_3 state)
  (asserta_register_4 st)
)
(Blk_10_Blk_19_0
  (asserta_register_1 state)
  (asserta_register_2 st)
  (asserta_register_3 hl)
  (asserta_register_4 fl)
)
(Blk_11_1
  (asserta_register_8 t31)
)
(Blk_11_Blk_12_0
  (asserta_register_1 fl)
  (asserta_register_2 tol)
)
(Blk_11_Blk_15_0
  (asserta_register_2 tos)
  (asserta_register_3 st)
  (asserta_register_4 hl)
  (asserta_register_5 fl)
)
(Blk_12_1
  (asserta_register_1 hl)
  (asserta_register_2 fl)
  (asserta_register_3 t32)
  (asserta_register_4 t33)
)
(Blk_12_2
  (asserta_register_3 t34)
)
(Blk_13_1
  (asserta_register_3 state)
  (asserta_register_4 st)
)
(Blk_13_Blk_19_0
  (asserta_register_1 state)
  (asserta_register_2 st)
  (asserta_register_3 hl)
  (asserta_register_4 fl)
)
(Blk_14_1
  (asserta_register_3 state)
  (asserta_register_4 st)
)
(Blk_14_Blk_19_0
  (asserta_register_1 state)
  (asserta_register_2 st)
  (asserta_register_3 hl)
  (asserta_register_4 fl)
)
(Blk_15_1
  (asserta_register_6 t35)
)
(Blk_15_Blk_16_0
  (asserta_register_1 tos)
)
(Blk_15_Blk_19_0
  (asserta_register_2 st)
  (asserta_register_3 hl)
  (asserta_register_4 fl)
)
(Blk_16_1
  (asserta_register_1 hl)
  (asserta_register_2 fl)
  (asserta_register_3 t36)
)
```

```
(Blk_17_1
  (asserta_register_3 state)
  (asserta_register_4 st)
)
(Blk_17_Blk_19_0
  (asserta_register_1 state)
  (asserta_register_2 st)
  (asserta_register_3 hl)
  (asserta_register_4 fl)
)
(Blk_18_1
  (asserta_register_3 state)
  (asserta_register_4 st)
)
(Blk_18_Blk_19_0
  (asserta_register_1 state)
  (asserta_register_2 st)
  (asserta_register_3 hl)
  (asserta_register_4 fl)
)
(Blk_19_1
  ( )
  ( )
  ( )
)
```

Figure B.4: **Register Binding Information of TLC**

```
Critical Register Binding:

Critical Path 0 :
   ( asserta_register_2 cof 3 )
   ( asserta_register_3 tol 3 )
   ( asserta_register_4 tos 3 )
   ( asserta_register_5 st 3 )
   ( asserta_register_6 hl 3 )
   ( asserta_register_7 fl 3 )
   ( asserta_register_8 t18 5 )
Critical Path 1 :
   ( asserta_register_1 hl 9 )
   ( asserta_register_2 fl 9 )
   ( asserta_register_3 t25 11 )
   ( asserta_register_4 t24 9 )
Critical Path 2 :
   ( asserta_register_1 state 15 )
   ( asserta_register_2 st 15 )
   ( asserta_register_3 hl 15 )
   ( asserta_register_4 fl 15 )
   ( asserta_register_5 cof 1 )
   ( asserta_register_6 tol 1 )
   ( asserta_register_7 tos 1 )
Critical Path 3 :
   ( asserta_register_1 state 19 )
   ( asserta_register_2 st 19 )
   ( asserta_register_3 hl 19 )
   ( asserta_register_4 fl 19 )
   ( asserta_register_5 cof 1 )
   ( asserta_register_6 tol 1 )
   ( asserta_register_7 tos 1 )
Critical Path 4 :
   ( asserta_register_8 t28 21 )
Critical Path 5 :
   ( asserta_register_1 hl 25 )
   ( asserta_register_2 fl 25 )
   ( asserta_register_3 t30 25 )
Critical Path 6 :
   ( asserta_register_1 state 29 )
   ( asserta_register_2 st 29 )
   ( asserta_register_3 hl 29 )
   ( asserta_register_4 fl 29 )
   ( asserta_register_5 cof 1 )
   ( asserta_register_6 tol 1 )
   ( asserta_register_7 tos 1 )
Critical Path 7 :
   ( asserta_register_1 state 33 )
   ( asserta_register_2 st 33 )
   ( asserta_register_3 hl 33 )
   ( asserta_register_4 fl 33 )
   ( asserta_register_5 cof 1 )
   ( asserta_register_6 tol 1 )
   ( asserta_register_7 tos 1 )
Critical Path 8 :
   ( asserta_register_8 t31 35 )
Critical Path 9 :
   ( asserta_register_1 hl 41 )
   ( asserta_register_2 fl 41 )
   ( asserta_register_3 t34 43 )
   ( asserta_register_4 t33 41 )
Critical Path 10 :
   ( asserta_register_1 state 47 )
   ( asserta_register_2 st 47 )
   ( asserta_register_3 hl 47 )
   ( asserta_register_4 fl 47 )
   ( asserta_register_5 cof 1 )
   ( asserta_register_6 tol 1 )
   ( asserta_register_7 tos 1 )
```

```
Critical Path 11 :
   ( asserta_register_1 state 51 )
   ( asserta_register_2 st 51 )
   ( asserta_register_3 hl 51 )
   ( asserta_register_4 fl 51 )
   ( asserta_register_5 cof 1 )
   ( asserta_register_6 tol 1 )
   ( asserta_register_7 tos 1 )
Critical Path 12 :
   ( asserta_register_2 tos 39 )
   ( asserta_register_3 st 39 )
   ( asserta_register_4 hl 39 )
   ( asserta_register_5 fl 39 )
   ( asserta_register_6 t35 53 )
Critical Path 13 :
   ( asserta_register_1 hl 59 )
   ( asserta_register_2 fl 59 )
   ( asserta_register_3 t36 59 )
Critical Path 14 :
   ( asserta_register_1 state 63 )
   ( asserta_register_2 st 63 )
   ( asserta_register_3 hl 63 )
   ( asserta_register_4 fl 63 )
   ( asserta_register_5 cof 1 )
   ( asserta_register_6 tol 1 )
   ( asserta_register_7 tos 1 )
Critical Path 15 :
   ( asserta_register_1 state 67 )
   ( asserta_register_2 st 67 )
   ( asserta_register_3 hl 67 )
   ( asserta_register_4 fl 67 )
   ( asserta_register_5 cof 1 )
   ( asserta_register_6 tol 1 )
   ( asserta_register_7 tos 1 )
Critical Path 16 :
   ( asserta_register_2 st 57 )
   ( asserta_register_3 hl 57 )
   ( asserta_register_4 fl 57 )
   ( asserta_register_5 cof 1 )
   ( asserta_register_6 tol 1 )
   ( asserta_register_7 tos 1 )
Critical Path 17 :
   ( asserta_register_5 cof 1 )
   ( asserta_register_6 tol 1 )
   ( asserta_register_7 tos 1 )
```

Figure B.5: **Critical Register Binding for TLC**

```
% Behavior Axioms

bs1_bs2_ax: AXIOM beh_transition(bt1)
  IMPLIES
    (Val_b(state,bs2) = Val_b(state,bs1) AND
     Val_b(st,bs2) = Val_b(st,bs1) AND
     Val_b(hl,bs2) = Val_b(hl,bs1) AND
     Val_b(fl,bs2) = Val_b(fl,bs1) AND
     Val_b(cof,bs2) = cof_in0 AND
     Val_b(tol,bs2) = Val_b(tol,bs1) AND
     Val_b(tos,bs2) = Val_b(tos,bs1) AND
     Val_b(t18,bs2) = Val_b(t18,bs1) AND
     Val_b(t23,bs2) = Val_b(t23,bs1) AND
     Val_b(t24,bs2) = Val_b(t24,bs1) AND
     Val_b(t25,bs2) = Val_b(t25,bs1) AND
     Val_b(t28,bs2) = Val_b(t28,bs1) AND
     Val_b(t30,bs2) = Val_b(t30,bs1) AND
     Val_b(t31,bs2) = Val_b(t31,bs1) AND
     Val_b(t32,bs2) = Val_b(t32,bs1) AND
     Val_b(t33,bs2) = Val_b(t33,bs1) AND
     Val_b(t34,bs2) = Val_b(t34,bs1) AND
     Val_b(t35,bs2) = Val_b(t35,bs1) AND
     Val_b(t36,bs2) = Val_b(t36,bs1))
bs2_bs3_ax: AXIOM beh_transition(bt2)
  IMPLIES
    (Val_b(state,bs3) = Val_b(state,bs2) AND
     Val_b(st,bs3) = Val_b(st,bs2) AND
     Val_b(hl,bs3) = Val_b(hl,bs2) AND
     Val_b(fl,bs3) = Val_b(fl,bs2) AND
     Val_b(cof,bs3) = Val_b(cof,bs2) AND
     Val_b(tol,bs3) = tol_in1 AND
     Val_b(tos,bs3) = Val_b(tos,bs2) AND
     Val_b(t18,bs3) = Val_b(t18,bs2) AND
     Val_b(t23,bs3) = Val_b(t23,bs2) AND
     Val_b(t24,bs3) = Val_b(t24,bs2) AND
     Val_b(t25,bs3) = Val_b(t25,bs2) AND
     Val_b(t28,bs3) = Val_b(t28,bs2) AND
     Val_b(t30,bs3) = Val_b(t30,bs2) AND
     Val_b(t31,bs3) = Val_b(t31,bs2) AND
     Val_b(t32,bs3) = Val_b(t32,bs2) AND
     Val_b(t33,bs3) = Val_b(t33,bs2) AND
     Val_b(t34,bs3) = Val_b(t34,bs2) AND
     Val_b(t35,bs3) = Val_b(t35,bs2) AND
     Val_b(t36,bs3) = Val_b(t36,bs2))
bs3_bs4_ax: AXIOM beh_transition(bt3)
  IMPLIES
    (Val_b(state,bs4) = Val_b(state,bs3) AND
     Val_b(st,bs4) = Val_b(st,bs3) AND
     Val_b(hl,bs4) = Val_b(hl,bs3) AND
     Val_b(fl,bs4) = Val_b(fl,bs3) AND
     Val_b(cof,bs4) = Val_b(cof,bs3) AND
     Val_b(tol,bs4) = Val_b(tol,bs3) AND
     Val_b(tos,bs4) = tos_in2 AND
     Val_b(t18,bs4) = Val_b(t18,bs3) AND
     Val_b(t23,bs4) = Val_b(t23,bs3) AND
     Val_b(t24,bs4) = Val_b(t24,bs3) AND
     Val_b(t25,bs4) = Val_b(t25,bs3) AND
     Val_b(t28,bs4) = Val_b(t28,bs3) AND
     Val_b(t30,bs4) = Val_b(t30,bs3) AND
     Val_b(t31,bs4) = Val_b(t31,bs3) AND
     Val_b(t32,bs4) = Val_b(t32,bs3) AND
     Val_b(t33,bs4) = Val_b(t33,bs3) AND
     Val_b(t34,bs4) = Val_b(t34,bs3) AND
     Val_b(t35,bs4) = Val_b(t35,bs3) AND
     Val_b(t36,bs4) = Val_b(t36,bs3))
bs4_bs1004_ax: AXIOM beh_transition(bt4)
  IMPLIES
    (Val_b(state,bs1004) = Val_b(state,bs4) AND
     Val_b(st,bs1004) = Val_b(st,bs4) AND
     Val_b(hl,bs1004) = Val_b(hl,bs4) AND
     Val_b(fl,bs1004) = Val_b(fl,bs4) AND
     Val_b(cof,bs1004) = Val_b(cof,bs4) AND
     Val_b(tol,bs1004) = Val_b(tol,bs4) AND
     Val_b(tos,bs1004) = Val_b(tos,bs4) AND
     Val_b(t18,bs1004) = bb_eq(Val_b(state,bs4),CONST_0) AND
     Val_b(t23,bs1004) = Val_b(t23,bs4) AND
     Val_b(t24,bs1004) = Val_b(t24,bs4) AND
     Val_b(t25,bs1004) = Val_b(t25,bs4) AND
     Val_b(t28,bs1004) = Val_b(t28,bs4) AND
     Val_b(t30,bs1004) = Val_b(t30,bs4) AND
     Val_b(t31,bs1004) = Val_b(t31,bs4) AND
     Val_b(t32,bs1004) = Val_b(t32,bs4) AND
     Val_b(t33,bs1004) = Val_b(t33,bs4) AND
     Val_b(t34,bs1004) = Val_b(t34,bs4) AND
     Val_b(t35,bs1004) = Val_b(t35,bs4) AND
     Val_b(t36,bs1004) = Val_b(t36,bs4))
bs1004_bs6_ax: AXIOM beh_transition(bt5)
  IMPLIES
    (Val_b(state,bs6) = Val_b(state,bs1004) AND
     Val_b(st,bs6) = Val_b(st,bs1004) AND
     Val_b(hl,bs6) = Val_b(hl,bs1004) AND
     Val_b(fl,bs6) = Val_b(fl,bs1004) AND
     Val_b(cof,bs6) = Val_b(cof,bs1004) AND
     Val_b(tol,bs6) = Val_b(tol,bs1004) AND
     Val_b(tos,bs6) = Val_b(tos,bs1004) AND
     Val_b(t18,bs6) = Val_b(t18,bs1004) AND
     Val_b(t23,bs6) = Val_b(t23,bs1004) AND
     Val_b(t24,bs6) = Val_b(t24,bs1004) AND
     Val_b(t25,bs6) = Val_b(t25,bs1004) AND
     Val_b(t28,bs6) = Val_b(t28,bs1004) AND
     Val_b(t30,bs6) = Val_b(t30,bs1004) AND
     Val_b(t31,bs6) = Val_b(t31,bs1004) AND
     Val_b(t32,bs6) = Val_b(t32,bs1004) AND
     Val_b(t33,bs6) = Val_b(t33,bs1004) AND
     Val_b(t34,bs6) = Val_b(t34,bs1004) AND
     Val_b(t35,bs6) = Val_b(t35,bs1004) AND
     Val_b(t36,bs6) = Val_b(t36,bs1004))
bs6_bs7_ax: AXIOM beh_transition(bt6)
  IMPLIES
    (Val_b(state,bs7) = Val_b(state,bs6) AND
     Val_b(st,bs7) = Val_b(st,bs6) AND
     Val_b(hl,bs7) = CONST_0 AND
     Val_b(fl,bs7) = Val_b(fl,bs6) AND
     Val_b(cof,bs7) = Val_b(cof,bs6) AND
     Val_b(tol,bs7) = Val_b(tol,bs6) AND
     Val_b(tos,bs7) = Val_b(tos,bs6) AND
     Val_b(t18,bs7) = Val_b(t18,bs6) AND
     Val_b(t23,bs7) = Val_b(t23,bs6) AND
     Val_b(t24,bs7) = Val_b(t24,bs6) AND
     Val_b(t25,bs7) = Val_b(t25,bs6) AND
     Val_b(t28,bs7) = Val_b(t28,bs6) AND
     Val_b(t30,bs7) = Val_b(t30,bs6) AND
     Val_b(t31,bs7) = Val_b(t31,bs6) AND
     Val_b(t32,bs7) = Val_b(t32,bs6) AND
     Val_b(t33,bs7) = Val_b(t33,bs6) AND
     Val_b(t34,bs7) = Val_b(t34,bs6) AND
     Val_b(t35,bs7) = Val_b(t35,bs6) AND
     Val_b(t36,bs7) = Val_b(t36,bs6))
bs7_bs8_ax: AXIOM beh_transition(bt7)
  IMPLIES
    (Val_b(state,bs8) = Val_b(state,bs7) AND
     Val_b(st,bs8) = Val_b(st,bs7) AND
     Val_b(hl,bs8) = Val_b(hl,bs7) AND
     Val_b(fl,bs8) = CONST_0 AND
     Val_b(cof,bs8) = Val_b(cof,bs7) AND
     Val_b(tol,bs8) = Val_b(tol,bs7) AND
     Val_b(tos,bs8) = Val_b(tos,bs7) AND
     Val_b(t18,bs8) = Val_b(t18,bs7) AND
     Val_b(t23,bs8) = Val_b(t23,bs7) AND
     Val_b(t24,bs8) = Val_b(t24,bs7) AND
     Val_b(t25,bs8) = Val_b(t25,bs7) AND
     Val_b(t28,bs8) = Val_b(t28,bs7) AND
     Val_b(t30,bs8) = Val_b(t30,bs7) AND
     Val_b(t31,bs8) = Val_b(t31,bs7) AND
     Val_b(t32,bs8) = Val_b(t32,bs7) AND
     Val_b(t33,bs8) = Val_b(t33,bs7) AND
     Val_b(t34,bs8) = Val_b(t34,bs7) AND
     Val_b(t35,bs8) = Val_b(t35,bs7) AND
     Val_b(t36,bs8) = Val_b(t36,bs7))
bs8_bs9_ax: AXIOM beh_transition(bt8)
  IMPLIES
    (Val_b(state,bs9) = Val_b(state,bs8) AND
     Val_b(st,bs9) = Val_b(st,bs8) AND
     Val_b(hl,bs9) = Val_b(hl,bs8) AND
     Val_b(fl,bs9) = Val_b(fl,bs8) AND
     Val_b(cof,bs9) = Val_b(cof,bs8) AND
     Val_b(tol,bs9) = Val_b(tol,bs8) AND
     Val_b(tos,bs9) = Val_b(tos,bs8) AND
     Val_b(t18,bs9) = Val_b(t18,bs8) AND
     Val_b(t23,bs9) = bb_eq(Val_b(cof,bs8),CONST_0) AND
     Val_b(t24,bs9) = Val_b(t24,bs8) AND
     Val_b(t25,bs9) = Val_b(t25,bs8) AND
     Val_b(t28,bs9) = Val_b(t28,bs8) AND
     Val_b(t30,bs9) = Val_b(t30,bs8) AND
     Val_b(t31,bs9) = Val_b(t31,bs8) AND
     Val_b(t32,bs9) = Val_b(t32,bs8) AND
     Val_b(t33,bs9) = Val_b(t33,bs8) AND
     Val_b(t34,bs9) = Val_b(t34,bs8) AND
     Val_b(t35,bs9) = Val_b(t35,bs8) AND
     Val_b(t36,bs9) = Val_b(t36,bs8))
bs9_bs10_ax: AXIOM beh_transition(bt9)
  IMPLIES
    (Val_b(state,bs10) = Val_b(state,bs9) AND
     Val_b(st,bs10) = Val_b(st,bs9) AND
     Val_b(hl,bs10) = Val_b(hl,bs9) AND
     Val_b(fl,bs10) = Val_b(fl,bs9) AND
     Val_b(cof,bs10) = Val_b(cof,bs9) AND
     Val_b(tol,bs10) = Val_b(tol,bs9) AND
     Val_b(tos,bs10) = Val_b(tos,bs9) AND
     Val_b(t18,bs10) = Val_b(t18,bs9) AND
     Val_b(t23,bs10) = Val_b(t23,bs9) AND
     Val_b(t24,bs10) = bb_eq(Val_b(tol,bs9),CONST_0) AND
     Val_b(t25,bs10) = Val_b(t25,bs9) AND
     Val_b(t28,bs10) = Val_b(t28,bs9) AND
     Val_b(t30,bs10) = Val_b(t30,bs9) AND
     Val_b(t31,bs10) = Val_b(t31,bs9) AND
     Val_b(t32,bs10) = Val_b(t32,bs9) AND
     Val_b(t33,bs10) = Val_b(t33,bs9) AND
     Val_b(t34,bs10) = Val_b(t34,bs9) AND
     Val_b(t35,bs10) = Val_b(t35,bs9) AND
     Val_b(t36,bs10) = Val_b(t36,bs9))
bs10_bs1010_ax: AXIOM beh_transition(bt10)
  IMPLIES
    (Val_b(state,bs1010) = Val_b(state,bs10) AND
     Val_b(st,bs1010) = Val_b(st,bs10) AND
     Val_b(hl,bs1010) = Val_b(hl,bs10) AND
     Val_b(fl,bs1010) = Val_b(fl,bs10) AND
     Val_b(cof,bs1010) = Val_b(cof,bs10) AND
     Val_b(tol,bs1010) = Val_b(tol,bs10) AND
     Val_b(tos,bs1010) = Val_b(tos,bs10) AND
     Val_b(t18,bs1010) = Val_b(t18,bs10) AND
     Val_b(t23,bs1010) = Val_b(t23,bs10) AND
     Val_b(t24,bs1010) = Val_b(t24,bs10) AND
     Val_b(t25,bs1010) = bb_plus(Val_b(t23,bs10),Val_b(t24,bs10)) AND
     Val_b(t28,bs1010) = Val_b(t28,bs10) AND
     Val_b(t30,bs1010) = Val_b(t30,bs10) AND
     Val_b(t31,bs1010) = Val_b(t31,bs10) AND
     Val_b(t32,bs1010) = Val_b(t32,bs10) AND
     Val_b(t33,bs1010) = Val_b(t33,bs10) AND
     Val_b(t34,bs1010) = Val_b(t34,bs10) AND
     Val_b(t35,bs1010) = Val_b(t35,bs10) AND
     Val_b(t36,bs1010) = Val_b(t36,bs10))
bs1010_bs11_ax: AXIOM beh_transition(bt11)
  IMPLIES
    (Val_b(state,bs11) = Val_b(state,bs1010) AND
     Val_b(st,bs11) = Val_b(st,bs1010) AND
     Val_b(hl,bs11) = Val_b(hl,bs1010) AND
     Val_b(fl,bs11) = Val_b(fl,bs1010) AND
     Val_b(cof,bs11) = Val_b(cof,bs1010) AND
     Val_b(tol,bs11) = Val_b(tol,bs1010) AND
     Val_b(tos,bs11) = Val_b(tos,bs1010) AND
     Val_b(t18,bs11) = Val_b(t18,bs1010) AND
     Val_b(t23,bs11) = Val_b(t23,bs1010) AND
     Val_b(t24,bs11) = Val_b(t24,bs1010) AND
     Val_b(t25,bs11) = Val_b(t25,bs1010) AND
     Val_b(t28,bs11) = Val_b(t28,bs1010) AND
     Val_b(t30,bs11) = Val_b(t30,bs1010) AND
     Val_b(t31,bs11) = Val_b(t31,bs1010) AND
     Val_b(t32,bs11) = Val_b(t32,bs1010) AND
     Val_b(t33,bs11) = Val_b(t33,bs1010) AND
     Val_b(t34,bs11) = Val_b(t34,bs1010) AND
     Val_b(t35,bs11) = Val_b(t35,bs1010) AND
     Val_b(t36,bs11) = Val_b(t36,bs1010))
```

```
bs11_bs12_ax: AXIOM beh_transition(bt12)
  IMPLIES
    (Val_b(state,bs12) = CONST_0 AND
     Val_b(st,bs12) = Val_b(st,bs11) AND
     Val_b(hl,bs12) = Val_b(hl,bs11) AND
     Val_b(fl,bs12) = Val_b(fl,bs11) AND
     Val_b(cof,bs12) = Val_b(cof,bs11) AND
     Val_b(tol,bs12) = Val_b(tol,bs11) AND
     Val_b(tos,bs12) = Val_b(tos,bs11) AND
     Val_b(t18,bs12) = Val_b(t18,bs11) AND
     Val_b(t23,bs12) = Val_b(t23,bs11) AND
     Val_b(t24,bs12) = Val_b(t24,bs11) AND
     Val_b(t25,bs12) = Val_b(t25,bs11) AND
     Val_b(t28,bs12) = Val_b(t28,bs11) AND
     Val_b(t30,bs12) = Val_b(t30,bs11) AND
     Val_b(t31,bs12) = Val_b(t31,bs11) AND
     Val_b(t32,bs12) = Val_b(t32,bs11) AND
     Val_b(t33,bs12) = Val_b(t33,bs11) AND
     Val_b(t34,bs12) = Val_b(t34,bs11) AND
     Val_b(t35,bs12) = Val_b(t35,bs11) AND
     Val_b(t36,bs12) = Val_b(t36,bs11))
bs12_bs42_ax: AXIOM beh_transition(bt13)
  IMPLIES
    (Val_b(state,bs42) = Val_b(state,bs12) AND
     Val_b(st,bs42) = CONST_0 AND
     Val_b(hl,bs42) = Val_b(hl,bs12) AND
     Val_b(fl,bs42) = Val_b(fl,bs12) AND
     Val_b(cof,bs42) = Val_b(cof,bs12) AND
     Val_b(tol,bs42) = Val_b(tol,bs12) AND
     Val_b(tos,bs42) = Val_b(tos,bs12) AND
     Val_b(t18,bs42) = Val_b(t18,bs12) AND
     Val_b(t23,bs42) = Val_b(t23,bs12) AND
     Val_b(t24,bs42) = Val_b(t24,bs12) AND
     Val_b(t25,bs42) = Val_b(t25,bs12) AND
     Val_b(t28,bs42) = Val_b(t28,bs12) AND
     Val_b(t30,bs42) = Val_b(t30,bs12) AND
     Val_b(t31,bs42) = Val_b(t31,bs12) AND
     Val_b(t32,bs42) = Val_b(t32,bs12) AND
     Val_b(t33,bs42) = Val_b(t33,bs12) AND
     Val_b(t34,bs42) = Val_b(t34,bs12) AND
     Val_b(t35,bs42) = Val_b(t35,bs12) AND
     Val_b(t36,bs42) = Val_b(t36,bs12))
bs42_bs44_ax: AXIOM beh_transition(bt14)
  IMPLIES
    (Val_b(state,bs44) = Val_b(state,bs42) AND
     Val_b(st,bs44) = Val_b(st,bs42) AND
     Val_b(hl,bs44) = Val_b(hl,bs42) AND
     Val_b(fl,bs44) = Val_b(fl,bs42) AND
     Val_b(cof,bs44) = Val_b(cof,bs42) AND
     Val_b(tol,bs44) = Val_b(tol,bs42) AND
     Val_b(tos,bs44) = Val_b(tos,bs42) AND
     Val_b(t18,bs44) = Val_b(t18,bs42) AND
     Val_b(t23,bs44) = Val_b(t23,bs42) AND
     Val_b(t24,bs44) = Val_b(t24,bs42) AND
     Val_b(t25,bs44) = Val_b(t25,bs42) AND
     Val_b(t28,bs44) = Val_b(t28,bs42) AND
     Val_b(t30,bs44) = Val_b(t30,bs42) AND
     Val_b(t31,bs44) = Val_b(t31,bs42) AND
     Val_b(t32,bs44) = Val_b(t32,bs42) AND
     Val_b(t33,bs44) = Val_b(t33,bs42) AND
     Val_b(t34,bs44) = Val_b(t34,bs42) AND
     Val_b(t35,bs44) = Val_b(t35,bs42) AND
     Val_b(t36,bs44) = Val_b(t36,bs42))
bs44_bs46_ax: AXIOM beh_transition(bt15)
  IMPLIES
    (Val_b(state,bs46) = Val_b(state,bs44) AND
     Val_b(st,bs46) = Val_b(st,bs44) AND
     Val_b(hl,bs46) = Val_b(hl,bs44) AND
     Val_b(fl,bs46) = Val_b(fl,bs44) AND
     Val_b(cof,bs46) = Val_b(cof,bs44) AND
     Val_b(tol,bs46) = Val_b(tol,bs44) AND
     Val_b(tos,bs46) = Val_b(tos,bs44) AND
     Val_b(t18,bs46) = Val_b(t18,bs44) AND
     Val_b(t23,bs46) = Val_b(t23,bs44) AND
     Val_b(t24,bs46) = Val_b(t24,bs44) AND
     Val_b(t25,bs46) = Val_b(t25,bs44) AND
     Val_b(t28,bs46) = Val_b(t28,bs44) AND
     Val_b(t30,bs46) = Val_b(t30,bs44) AND
     Val_b(t31,bs46) = Val_b(t31,bs44) AND
     Val_b(t32,bs46) = Val_b(t32,bs44) AND
     Val_b(t33,bs46) = Val_b(t33,bs44) AND
     Val_b(t34,bs46) = Val_b(t34,bs44) AND
     Val_b(t35,bs46) = Val_b(t35,bs44) AND
     Val_b(t36,bs46) = Val_b(t36,bs44))
bs46_bs1_ax: AXIOM beh_transition(bt16)
  IMPLIES
    (Val_b(state,bs1) = Val_b(state,bs46) AND
     Val_b(st,bs1) = Val_b(st,bs46) AND
     Val_b(hl,bs1) = Val_b(hl,bs46) AND
     Val_b(fl,bs1) = Val_b(fl,bs46) AND
     Val_b(cof,bs1) = Val_b(cof,bs46) AND
     Val_b(tol,bs1) = Val_b(tol,bs46) AND
     Val_b(tos,bs1) = Val_b(tos,bs46) AND
     Val_b(t18,bs1) = Val_b(t18,bs46) AND
     Val_b(t23,bs1) = Val_b(t23,bs46) AND
     Val_b(t24,bs1) = Val_b(t24,bs46) AND
     Val_b(t25,bs1) = Val_b(t25,bs46) AND
     Val_b(t28,bs1) = Val_b(t28,bs46) AND
     Val_b(t30,bs1) = Val_b(t30,bs46) AND
     Val_b(t31,bs1) = Val_b(t31,bs46) AND
     Val_b(t32,bs1) = Val_b(t32,bs46) AND
     Val_b(t33,bs1) = Val_b(t33,bs46) AND
     Val_b(t34,bs1) = Val_b(t34,bs46) AND
     Val_b(t35,bs1) = Val_b(t35,bs46) AND
     Val_b(t36,bs1) = Val_b(t36,bs46))
bs1010_bs13_ax: AXIOM beh_transition(bt17)
  IMPLIES
    (Val_b(state,bs13) = Val_b(state,bs1010) AND
     Val_b(st,bs13) = Val_b(st,bs1010) AND
     Val_b(hl,bs13) = Val_b(hl,bs1010) AND
     Val_b(fl,bs13) = Val_b(fl,bs1010) AND
     Val_b(cof,bs13) = Val_b(cof,bs1010) AND
     Val_b(tol,bs13) = Val_b(tol,bs1010) AND
     Val_b(tos,bs13) = Val_b(tos,bs1010) AND
     Val_b(t18,bs13) = Val_b(t18,bs1010) AND
     Val_b(t23,bs13) = Val_b(t23,bs1010) AND
     Val_b(t24,bs13) = Val_b(t24,bs1010) AND
     Val_b(t25,bs13) = Val_b(t25,bs1010) AND
     Val_b(t28,bs13) = Val_b(t28,bs1010) AND
     Val_b(t30,bs13) = Val_b(t30,bs1010) AND
     Val_b(t31,bs13) = Val_b(t31,bs1010) AND
     Val_b(t32,bs13) = Val_b(t32,bs1010) AND
     Val_b(t33,bs13) = Val_b(t33,bs1010) AND
     Val_b(t34,bs13) = Val_b(t34,bs1010) AND
     Val_b(t35,bs13) = Val_b(t35,bs1010) AND
     Val_b(t36,bs13) = Val_b(t36,bs1010))
bs13_bs14_ax: AXIOM beh_transition(bt18)
  IMPLIES
    (Val_b(state,bs14) = CONST_0 AND
     Val_b(st,bs14) = Val_b(st,bs13) AND
     Val_b(hl,bs14) = Val_b(hl,bs13) AND
     Val_b(fl,bs14) = Val_b(fl,bs13) AND
     Val_b(cof,bs14) = Val_b(cof,bs13) AND
     Val_b(tol,bs14) = Val_b(tol,bs13) AND
     Val_b(tos,bs14) = Val_b(tos,bs13) AND
     Val_b(t18,bs14) = Val_b(t18,bs13) AND
     Val_b(t23,bs14) = Val_b(t23,bs13) AND
     Val_b(t24,bs14) = Val_b(t24,bs13) AND
     Val_b(t25,bs14) = Val_b(t25,bs13) AND
     Val_b(t28,bs14) = Val_b(t28,bs13) AND
     Val_b(t30,bs14) = Val_b(t30,bs13) AND
     Val_b(t31,bs14) = Val_b(t31,bs13) AND
     Val_b(t32,bs14) = Val_b(t32,bs13) AND
     Val_b(t33,bs14) = Val_b(t33,bs13) AND
     Val_b(t34,bs14) = Val_b(t34,bs13) AND
     Val_b(t35,bs14) = Val_b(t35,bs13) AND
     Val_b(t36,bs14) = Val_b(t36,bs13))
bs14_bs42_ax: AXIOM beh_transition(bt19)
  IMPLIES
    (Val_b(state,bs42) = Val_b(state,bs14) AND
     Val_b(st,bs42) = CONST_0 AND
     Val_b(hl,bs42) = Val_b(hl,bs14) AND
     Val_b(fl,bs42) = Val_b(fl,bs14) AND
     Val_b(cof,bs42) = Val_b(cof,bs14) AND
     Val_b(tol,bs42) = Val_b(tol,bs14) AND
     Val_b(tos,bs42) = Val_b(tos,bs14) AND
     Val_b(t18,bs42) = Val_b(t18,bs14) AND
     Val_b(t23,bs42) = Val_b(t23,bs14) AND
     Val_b(t24,bs42) = Val_b(t24,bs14) AND
     Val_b(t25,bs42) = Val_b(t25,bs14) AND
     Val_b(t28,bs42) = Val_b(t28,bs14) AND
     Val_b(t30,bs42) = Val_b(t30,bs14) AND
     Val_b(t31,bs42) = Val_b(t31,bs14) AND
     Val_b(t32,bs42) = Val_b(t32,bs14) AND
     Val_b(t33,bs42) = Val_b(t33,bs14) AND
     Val_b(t34,bs42) = Val_b(t34,bs14) AND
     Val_b(t35,bs42) = Val_b(t35,bs14) AND
     Val_b(t36,bs42) = Val_b(t36,bs14))
bs1004_bs15_ax: AXIOM beh_transition(bt20)
  IMPLIES
    (Val_b(state,bs15) = Val_b(state,bs1004) AND
     Val_b(st,bs15) = Val_b(st,bs1004) AND
     Val_b(hl,bs15) = Val_b(hl,bs1004) AND
     Val_b(fl,bs15) = Val_b(fl,bs1004) AND
     Val_b(cof,bs15) = Val_b(cof,bs1004) AND
     Val_b(tol,bs15) = Val_b(tol,bs1004) AND
     Val_b(tos,bs15) = Val_b(tos,bs1004) AND
     Val_b(t18,bs15) = Val_b(t18,bs1004) AND
     Val_b(t23,bs15) = Val_b(t23,bs1004) AND
     Val_b(t24,bs15) = Val_b(t24,bs1004) AND
     Val_b(t25,bs15) = Val_b(t25,bs1004) AND
     Val_b(t28,bs15) = Val_b(t28,bs1004) AND
     Val_b(t30,bs15) = Val_b(t30,bs1004) AND
     Val_b(t31,bs15) = Val_b(t31,bs1004) AND
     Val_b(t32,bs15) = Val_b(t32,bs1004) AND
     Val_b(t33,bs15) = Val_b(t33,bs1004) AND
     Val_b(t34,bs15) = Val_b(t34,bs1004) AND
     Val_b(t35,bs15) = Val_b(t35,bs1004) AND
     Val_b(t36,bs15) = Val_b(t36,bs1004))
bs15_bs1015_ax: AXIOM beh_transition(bt21)
  IMPLIES
    (Val_b(state,bs1015) = Val_b(state,bs15) AND
     Val_b(st,bs1015) = Val_b(st,bs15) AND
     Val_b(hl,bs1015) = Val_b(hl,bs15) AND
     Val_b(fl,bs1015) = Val_b(fl,bs15) AND
     Val_b(cof,bs1015) = Val_b(cof,bs15) AND
     Val_b(tol,bs1015) = Val_b(tol,bs15) AND
     Val_b(tos,bs1015) = Val_b(tos,bs15) AND
     Val_b(t18,bs1015) = Val_b(t18,bs15) AND
     Val_b(t23,bs1015) = Val_b(t23,bs15) AND
     Val_b(t24,bs1015) = Val_b(t24,bs15) AND
     Val_b(t25,bs1015) = Val_b(t25,bs15) AND
     Val_b(t28,bs1015) = bb_eq(Val_b(state,bs15),CONST_0) AND
     Val_b(t30,bs1015) = Val_b(t30,bs15) AND
     Val_b(t31,bs1015) = Val_b(t31,bs15) AND
     Val_b(t32,bs1015) = Val_b(t32,bs15) AND
     Val_b(t33,bs1015) = Val_b(t33,bs15) AND
     Val_b(t34,bs1015) = Val_b(t34,bs15) AND
     Val_b(t35,bs1015) = Val_b(t35,bs15) AND
     Val_b(t36,bs1015) = Val_b(t36,bs15))
bs1015_bs16_ax: AXIOM beh_transition(bt22)
  IMPLIES
    (Val_b(state,bs16) = Val_b(state,bs1015) AND
     Val_b(st,bs16) = Val_b(st,bs1015) AND
     Val_b(hl,bs16) = Val_b(hl,bs1015) AND
     Val_b(fl,bs16) = Val_b(fl,bs1015) AND
     Val_b(cof,bs16) = Val_b(cof,bs1015) AND
     Val_b(tol,bs16) = Val_b(tol,bs1015) AND
     Val_b(tos,bs16) = Val_b(tos,bs1015) AND
     Val_b(t18,bs16) = Val_b(t18,bs1015) AND
     Val_b(t23,bs16) = Val_b(t23,bs1015) AND
     Val_b(t24,bs16) = Val_b(t24,bs1015) AND
     Val_b(t25,bs16) = Val_b(t25,bs1015) AND
     Val_b(t28,bs16) = Val_b(t28,bs1015) AND
     Val_b(t30,bs16) = Val_b(t30,bs1015) AND
     Val_b(t31,bs16) = Val_b(t31,bs1015) AND
     Val_b(t32,bs16) = Val_b(t32,bs1015) AND
     Val_b(t33,bs16) = Val_b(t33,bs1015) AND
     Val_b(t34,bs16) = Val_b(t34,bs1015) AND
     Val_b(t35,bs16) = Val_b(t35,bs1015) AND
     Val_b(t36,bs16) = Val_b(t36,bs1015))
```

```
bs16_bs17_ax: AXIOM beh_transition(bt23)
  IMPLIES
    (Val_b(state,bs17) = Val_b(state,bs16) AND
     Val_b(st,bs17) = Val_b(st,bs16) AND
     Val_b(hl,bs17) = CONST_0 AND
     Val_b(fl,bs17) = Val_b(fl,bs16) AND
     Val_b(cof,bs17) = Val_b(cof,bs16) AND
     Val_b(tol,bs17) = Val_b(tol,bs16) AND
     Val_b(tos,bs17) = Val_b(tos,bs16) AND
     Val_b(t18,bs17) = Val_b(t18,bs16) AND
     Val_b(t23,bs17) = Val_b(t23,bs16) AND
     Val_b(t24,bs17) = Val_b(t24,bs16) AND
     Val_b(t25,bs17) = Val_b(t25,bs16) AND
     Val_b(t28,bs17) = Val_b(t28,bs16) AND
     Val_b(t30,bs17) = Val_b(t30,bs16) AND
     Val_b(t31,bs17) = Val_b(t31,bs16) AND
     Val_b(t32,bs17) = Val_b(t32,bs16) AND
     Val_b(t33,bs17) = Val_b(t33,bs16) AND
     Val_b(t34,bs17) = Val_b(t34,bs16) AND
     Val_b(t35,bs17) = Val_b(t35,bs16) AND
     Val_b(t36,bs17) = Val_b(t36,bs16))
bs17_bs18_ax: AXIOM beh_transition(bt24)
  IMPLIES
    (Val_b(state,bs18) = Val_b(state,bs17) AND
     Val_b(st,bs18) = Val_b(st,bs17) AND
     Val_b(hl,bs18) = Val_b(hl,bs17) AND
     Val_b(fl,bs18) = CONST_0 AND
     Val_b(cof,bs18) = Val_b(cof,bs17) AND
     Val_b(tol,bs18) = Val_b(tol,bs17) AND
     Val_b(tos,bs18) = Val_b(tos,bs17) AND
     Val_b(t18,bs18) = Val_b(t18,bs17) AND
     Val_b(t23,bs18) = Val_b(t23,bs17) AND
     Val_b(t24,bs18) = Val_b(t24,bs17) AND
     Val_b(t25,bs18) = Val_b(t25,bs17) AND
     Val_b(t28,bs18) = Val_b(t28,bs17) AND
     Val_b(t30,bs18) = Val_b(t30,bs17) AND
     Val_b(t31,bs18) = Val_b(t31,bs17) AND
     Val_b(t32,bs18) = Val_b(t32,bs17) AND
     Val_b(t33,bs18) = Val_b(t33,bs17) AND
     Val_b(t34,bs18) = Val_b(t34,bs17) AND
     Val_b(t35,bs18) = Val_b(t35,bs17) AND
     Val_b(t36,bs18) = Val_b(t36,bs17))
bs18_bs1018_ax: AXIOM beh_transition(bt25)
  IMPLIES
    (Val_b(state,bs1018) = Val_b(state,bs18) AND
     Val_b(st,bs1018) = Val_b(st,bs18) AND
     Val_b(hl,bs1018) = Val_b(hl,bs18) AND
     Val_b(fl,bs1018) = Val_b(fl,bs18) AND
     Val_b(cof,bs1018) = Val_b(cof,bs18) AND
     Val_b(tol,bs1018) = Val_b(tol,bs18) AND
     Val_b(tos,bs1018) = Val_b(tos,bs18) AND
     Val_b(t18,bs1018) = Val_b(t18,bs18) AND
     Val_b(t23,bs1018) = Val_b(t23,bs18) AND
     Val_b(t24,bs1018) = Val_b(t24,bs18) AND
     Val_b(t25,bs1018) = Val_b(t25,bs18) AND
     Val_b(t28,bs1018) = Val_b(t28,bs18) AND
     Val_b(t30,bs1018) = bb_eq(Val_b(tos,bs18),CONST_0) AND
     Val_b(t31,bs1018) = Val_b(t31,bs18) AND
     Val_b(t32,bs1018) = Val_b(t32,bs18) AND
     Val_b(t33,bs1018) = Val_b(t33,bs18) AND
     Val_b(t34,bs1018) = Val_b(t34,bs18) AND
     Val_b(t35,bs1018) = Val_b(t35,bs18) AND
     Val_b(t36,bs1018) = Val_b(t36,bs18))
bs1018_bs19_ax: AXIOM beh_transition(bt26)
  IMPLIES
    (Val_b(state,bs19) = Val_b(state,bs1018) AND
     Val_b(st,bs19) = Val_b(st,bs1018) AND
     Val_b(hl,bs19) = Val_b(hl,bs1018) AND
     Val_b(fl,bs19) = Val_b(fl,bs1018) AND
     Val_b(cof,bs19) = Val_b(cof,bs1018) AND
     Val_b(tol,bs19) = Val_b(tol,bs1018) AND
     Val_b(tos,bs19) = Val_b(tos,bs1018) AND
     Val_b(t18,bs19) = Val_b(t18,bs1018) AND
     Val_b(t23,bs19) = Val_b(t23,bs1018) AND
     Val_b(t24,bs19) = Val_b(t24,bs1018) AND
     Val_b(t25,bs19) = Val_b(t25,bs1018) AND
     Val_b(t28,bs19) = Val_b(t28,bs1018) AND
     Val_b(t30,bs19) = Val_b(t30,bs1018) AND
     Val_b(t31,bs19) = Val_b(t31,bs1018) AND
     Val_b(t32,bs19) = Val_b(t32,bs1018) AND
     Val_b(t33,bs19) = Val_b(t33,bs1018) AND
     Val_b(t34,bs19) = Val_b(t34,bs1018) AND
     Val_b(t35,bs19) = Val_b(t35,bs1018) AND
     Val_b(t36,bs19) = Val_b(t36,bs1018))
bs19_bs20_ax: AXIOM beh_transition(bt27)
  IMPLIES
    (Val_b(state,bs20) = CONST_0 AND
     Val_b(st,bs20) = Val_b(st,bs19) AND
     Val_b(hl,bs20) = Val_b(hl,bs19) AND
     Val_b(fl,bs20) = Val_b(fl,bs19) AND
     Val_b(cof,bs20) = Val_b(cof,bs19) AND
     Val_b(tol,bs20) = Val_b(tol,bs19) AND
     Val_b(tos,bs20) = Val_b(tos,bs19) AND
     Val_b(t18,bs20) = Val_b(t18,bs19) AND
     Val_b(t23,bs20) = Val_b(t23,bs19) AND
     Val_b(t24,bs20) = Val_b(t24,bs19) AND
     Val_b(t25,bs20) = Val_b(t25,bs19) AND
     Val_b(t28,bs20) = Val_b(t28,bs19) AND
     Val_b(t30,bs20) = Val_b(t30,bs19) AND
     Val_b(t31,bs20) = Val_b(t31,bs19) AND
     Val_b(t32,bs20) = Val_b(t32,bs19) AND
     Val_b(t33,bs20) = Val_b(t33,bs19) AND
     Val_b(t34,bs20) = Val_b(t34,bs19) AND
     Val_b(t35,bs20) = Val_b(t35,bs19) AND
     Val_b(t36,bs20) = Val_b(t36,bs19))
bs20_bs42_ax: AXIOM beh_transition(bt28)
  IMPLIES
    (Val_b(state,bs42) = Val_b(state,bs20) AND
     Val_b(st,bs42) = CONST_0 AND
     Val_b(hl,bs42) = Val_b(hl,bs20) AND
     Val_b(fl,bs42) = Val_b(fl,bs20) AND
     Val_b(cof,bs42) = Val_b(cof,bs20) AND
     Val_b(tol,bs42) = Val_b(tol,bs20) AND
     Val_b(tos,bs42) = Val_b(tos,bs20) AND
     Val_b(t18,bs42) = Val_b(t18,bs20) AND
     Val_b(t23,bs42) = Val_b(t23,bs20) AND
     Val_b(t24,bs42) = Val_b(t24,bs20) AND
     Val_b(t25,bs42) = Val_b(t25,bs20) AND
     Val_b(t28,bs42) = Val_b(t28,bs20) AND
     Val_b(t30,bs42) = Val_b(t30,bs20) AND
     Val_b(t31,bs42) = Val_b(t31,bs20) AND
     Val_b(t32,bs42) = Val_b(t32,bs20) AND
     Val_b(t33,bs42) = Val_b(t33,bs20) AND
     Val_b(t34,bs42) = Val_b(t34,bs20) AND
     Val_b(t35,bs42) = Val_b(t35,bs20) AND
     Val_b(t36,bs42) = Val_b(t36,bs20))
bs1018_bs21_ax: AXIOM beh_transition(bt29)
  IMPLIES
    (Val_b(state,bs21) = Val_b(state,bs1018) AND
     Val_b(st,bs21) = Val_b(st,bs1018) AND
     Val_b(hl,bs21) = Val_b(hl,bs1018) AND
     Val_b(fl,bs21) = Val_b(fl,bs1018) AND
     Val_b(cof,bs21) = Val_b(cof,bs1018) AND
     Val_b(tol,bs21) = Val_b(tol,bs1018) AND
     Val_b(tos,bs21) = Val_b(tos,bs1018) AND
     Val_b(t18,bs21) = Val_b(t18,bs1018) AND
     Val_b(t23,bs21) = Val_b(t23,bs1018) AND
     Val_b(t24,bs21) = Val_b(t24,bs1018) AND
     Val_b(t25,bs21) = Val_b(t25,bs1018) AND
     Val_b(t28,bs21) = Val_b(t28,bs1018) AND
     Val_b(t30,bs21) = Val_b(t30,bs1018) AND
     Val_b(t31,bs21) = Val_b(t31,bs1018) AND
     Val_b(t32,bs21) = Val_b(t32,bs1018) AND
     Val_b(t33,bs21) = Val_b(t33,bs1018) AND
     Val_b(t34,bs21) = Val_b(t34,bs1018) AND
     Val_b(t35,bs21) = Val_b(t35,bs1018) AND
     Val_b(t36,bs21) = Val_b(t36,bs1018))
bs21_bs22_ax: AXIOM beh_transition(bt30)
  IMPLIES
    (Val_b(state,bs22) = CONST_0 AND
     Val_b(st,bs22) = Val_b(st,bs21) AND
     Val_b(hl,bs22) = Val_b(hl,bs21) AND
     Val_b(fl,bs22) = Val_b(fl,bs21) AND
     Val_b(cof,bs22) = Val_b(cof,bs21) AND
     Val_b(tol,bs22) = Val_b(tol,bs21) AND
     Val_b(tos,bs22) = Val_b(tos,bs21) AND
     Val_b(t18,bs22) = Val_b(t18,bs21) AND
     Val_b(t23,bs22) = Val_b(t23,bs21) AND
     Val_b(t24,bs22) = Val_b(t24,bs21) AND
     Val_b(t25,bs22) = Val_b(t25,bs21) AND
     Val_b(t28,bs22) = Val_b(t28,bs21) AND
     Val_b(t30,bs22) = Val_b(t30,bs21) AND
     Val_b(t31,bs22) = Val_b(t31,bs21) AND
     Val_b(t32,bs22) = Val_b(t32,bs21) AND
     Val_b(t33,bs22) = Val_b(t33,bs21) AND
     Val_b(t34,bs22) = Val_b(t34,bs21) AND
     Val_b(t35,bs22) = Val_b(t35,bs21) AND
     Val_b(t36,bs22) = Val_b(t36,bs21))
 bs22_bs42_ax: AXIOM beh_transition(bt31)
   IMPLIES
    (Val_b(state,bs42) = Val_b(state,bs22) AND
     Val_b(st,bs42) = CONST_0 AND
     Val_b(hl,bs42) = Val_b(hl,bs22) AND
     Val_b(fl,bs42) = Val_b(fl,bs22) AND
     Val_b(cof,bs42) = Val_b(cof,bs22) AND
     Val_b(tol,bs42) = Val_b(tol,bs22) AND
     Val_b(tos,bs42) = Val_b(tos,bs22) AND
     Val_b(t18,bs42) = Val_b(t18,bs22) AND
     Val_b(t23,bs42) = Val_b(t23,bs22) AND
     Val_b(t24,bs42) = Val_b(t24,bs22) AND
     Val_b(t25,bs42) = Val_b(t25,bs22) AND
     Val_b(t28,bs42) = Val_b(t28,bs22) AND
     Val_b(t30,bs42) = Val_b(t30,bs22) AND
     Val_b(t31,bs42) = Val_b(t31,bs22) AND
     Val_b(t32,bs42) = Val_b(t32,bs22) AND
     Val_b(t33,bs42) = Val_b(t33,bs22) AND
     Val_b(t34,bs42) = Val_b(t34,bs22) AND
     Val_b(t35,bs42) = Val_b(t35,bs22) AND
     Val_b(t36,bs42) = Val_b(t36,bs22))
 bs1015_bs23_ax: AXIOM beh_transition(bt32)
   IMPLIES
    (Val_b(state,bs23) = Val_b(state,bs1015) AND
     Val_b(st,bs23) = Val_b(st,bs1015) AND
     Val_b(hl,bs23) = Val_b(hl,bs1015) AND
     Val_b(fl,bs23) = Val_b(fl,bs1015) AND
     Val_b(cof,bs23) = Val_b(cof,bs1015) AND
     Val_b(tol,bs23) = Val_b(tol,bs1015) AND
     Val_b(tos,bs23) = Val_b(tos,bs1015) AND
     Val_b(t18,bs23) = Val_b(t18,bs1015) AND
     Val_b(t23,bs23) = Val_b(t23,bs1015) AND
     Val_b(t24,bs23) = Val_b(t24,bs1015) AND
     Val_b(t25,bs23) = Val_b(t25,bs1015) AND
     Val_b(t28,bs23) = Val_b(t28,bs1015) AND
     Val_b(t30,bs23) = Val_b(t30,bs1015) AND
     Val_b(t31,bs23) = Val_b(t31,bs1015) AND
     Val_b(t32,bs23) = Val_b(t32,bs1015) AND
     Val_b(t33,bs23) = Val_b(t33,bs1015) AND
     Val_b(t34,bs23) = Val_b(t34,bs1015) AND
     Val_b(t35,bs23) = Val_b(t35,bs1015) AND
     Val_b(t36,bs23) = Val_b(t36,bs1015))
 bs23_bs1023_ax: AXIOM beh_transition(bt33)
   IMPLIES
    (Val_b(state,bs1023) = Val_b(state,bs23) AND
     Val_b(st,bs1023) = Val_b(st,bs23) AND
     Val_b(hl,bs1023) = Val_b(hl,bs23) AND
     Val_b(fl,bs1023) = Val_b(fl,bs23) AND
     Val_b(cof,bs1023) = Val_b(cof,bs23) AND
     Val_b(tol,bs1023) = Val_b(tol,bs23) AND
     Val_b(tos,bs1023) = Val_b(tos,bs23) AND
     Val_b(t18,bs1023) = Val_b(t18,bs23) AND
     Val_b(t23,bs1023) = Val_b(t23,bs23) AND
     Val_b(t24,bs1023) = Val_b(t24,bs23) AND
     Val_b(t25,bs1023) = Val_b(t25,bs23) AND
     Val_b(t28,bs1023) = Val_b(t28,bs23) AND
     Val_b(t30,bs1023) = Val_b(t30,bs23) AND
     Val_b(t31,bs1023) = bb_eq(Val_b(state,bs23),CONST_0) AND
     Val_b(t32,bs1023) = Val_b(t32,bs23) AND
     Val_b(t33,bs1023) = Val_b(t33,bs23) AND
     Val_b(t34,bs1023) = Val_b(t34,bs23) AND
     Val_b(t35,bs1023) = Val_b(t35,bs23) AND
     Val_b(t36,bs1023) = Val_b(t36,bs23))
```

```
bs24_bs25_ax: AXIOM beh_transition(bt35)
  IMPLIES
    (Val_b(state,bs25) = Val_b(state,bs24) AND
     Val_b(st,bs25) = Val_b(st,bs24) AND
     Val_b(hl,bs25) = CONST_0 AND
     Val_b(fl,bs25) = Val_b(fl,bs24) AND
     Val_b(cof,bs25) = Val_b(cof,bs24) AND
     Val_b(tol,bs25) = Val_b(tol,bs24) AND
     Val_b(tos,bs25) = Val_b(tos,bs24) AND
     Val_b(t18,bs25) = Val_b(t18,bs24) AND
     Val_b(t23,bs25) = Val_b(t23,bs24) AND
     Val_b(t24,bs25) = Val_b(t24,bs24) AND
     Val_b(t25,bs25) = Val_b(t25,bs24) AND
     Val_b(t28,bs25) = Val_b(t28,bs24) AND
     Val_b(t30,bs25) = Val_b(t30,bs24) AND
     Val_b(t31,bs25) = Val_b(t31,bs24) AND
     Val_b(t32,bs25) = Val_b(t32,bs24) AND
     Val_b(t33,bs25) = Val_b(t33,bs24) AND
     Val_b(t34,bs25) = Val_b(t34,bs24) AND
     Val_b(t35,bs25) = Val_b(t35,bs24) AND
     Val_b(t36,bs25) = Val_b(t36,bs24))
bs25_bs26_ax: AXIOM beh_transition(bt36)
  IMPLIES
    (Val_b(state,bs26) = Val_b(state,bs25) AND
     Val_b(st,bs26) = Val_b(st,bs25) AND
     Val_b(hl,bs26) = Val_b(hl,bs25) AND
     Val_b(fl,bs26) = CONST_0 AND
     Val_b(cof,bs26) = Val_b(cof,bs25) AND
     Val_b(tol,bs26) = Val_b(tol,bs25) AND
     Val_b(tos,bs26) = Val_b(tos,bs25) AND
     Val_b(t18,bs26) = Val_b(t18,bs25) AND
     Val_b(t23,bs26) = Val_b(t23,bs25) AND
     Val_b(t24,bs26) = Val_b(t24,bs25) AND
     Val_b(t25,bs26) = Val_b(t25,bs25) AND
     Val_b(t28,bs26) = Val_b(t28,bs25) AND
     Val_b(t30,bs26) = Val_b(t30,bs25) AND
     Val_b(t31,bs26) = Val_b(t31,bs25) AND
     Val_b(t32,bs26) = Val_b(t32,bs25) AND
     Val_b(t33,bs26) = Val_b(t33,bs25) AND
     Val_b(t34,bs26) = Val_b(t34,bs25) AND
     Val_b(t35,bs26) = Val_b(t35,bs25) AND
     Val_b(t36,bs26) = Val_b(t36,bs25))
bs26_bs27_ax: AXIOM beh_transition(bt37)
  IMPLIES
    (Val_b(state,bs27) = Val_b(state,bs26) AND
     Val_b(st,bs27) = Val_b(st,bs26) AND
     Val_b(hl,bs27) = Val_b(hl,bs26) AND
     Val_b(fl,bs27) = Val_b(fl,bs26) AND
     Val_b(cof,bs27) = Val_b(cof,bs26) AND
     Val_b(tol,bs27) = Val_b(tol,bs26) AND
     Val_b(tos,bs27) = Val_b(tos,bs26) AND
     Val_b(t18,bs27) = Val_b(t18,bs26) AND
     Val_b(t23,bs27) = Val_b(t23,bs26) AND
     Val_b(t24,bs27) = Val_b(t24,bs26) AND
     Val_b(t25,bs27) = Val_b(t25,bs26) AND
     Val_b(t28,bs27) = Val_b(t28,bs26) AND
     Val_b(t30,bs27) = Val_b(t30,bs26) AND
     Val_b(t31,bs27) = Val_b(t31,bs26) AND
     Val_b(t32,bs27) = bb_eq(Val_b(cof,bs26),CONST_0) AND
     Val_b(t33,bs27) = Val_b(t33,bs26) AND
     Val_b(t34,bs27) = Val_b(t34,bs26) AND
     Val_b(t35,bs27) = Val_b(t35,bs26) AND
     Val_b(t36,bs27) = Val_b(t36,bs26))
bs27_bs28_ax: AXIOM beh_transition(bt38)
  IMPLIES
    (Val_b(state,bs28) = Val_b(state,bs27) AND
     Val_b(st,bs28) = Val_b(st,bs27) AND
     Val_b(hl,bs28) = Val_b(hl,bs27) AND
     Val_b(fl,bs28) = Val_b(fl,bs27) AND
     Val_b(cof,bs28) = Val_b(cof,bs27) AND
     Val_b(tol,bs28) = Val_b(tol,bs27) AND
     Val_b(tos,bs28) = Val_b(tos,bs27) AND
     Val_b(t18,bs28) = Val_b(t18,bs27) AND
     Val_b(t23,bs28) = Val_b(t23,bs27) AND
     Val_b(t24,bs28) = Val_b(t24,bs27) AND
     Val_b(t25,bs28) = Val_b(t25,bs27) AND
     Val_b(t28,bs28) = Val_b(t28,bs27) AND
     Val_b(t30,bs28) = Val_b(t30,bs27) AND
     Val_b(t31,bs28) = Val_b(t31,bs27) AND
     Val_b(t32,bs28) = Val_b(t32,bs27) AND
     Val_b(t33,bs28) = bb_eq(Val_b(tol,bs27),CONST_0) AND
     Val_b(t34,bs28) = Val_b(t34,bs27) AND
     Val_b(t35,bs28) = Val_b(t35,bs27) AND
     Val_b(t36,bs28) = Val_b(t36,bs27))
bs28_bs1028_ax: AXIOM beh_transition(bt39)
  IMPLIES
    (Val_b(state,bs1028) = Val_b(state,bs28) AND
     Val_b(st,bs1028) = Val_b(st,bs28) AND
     Val_b(hl,bs1028) = Val_b(hl,bs28) AND
     Val_b(fl,bs1028) = Val_b(fl,bs28) AND
     Val_b(cof,bs1028) = Val_b(cof,bs28) AND
     Val_b(tol,bs1028) = Val_b(tol,bs28) AND
     Val_b(tos,bs1028) = Val_b(tos,bs28) AND
     Val_b(t18,bs1028) = Val_b(t18,bs28) AND
     Val_b(t23,bs1028) = Val_b(t23,bs28) AND
     Val_b(t24,bs1028) = Val_b(t24,bs28) AND
     Val_b(t25,bs1028) = Val_b(t25,bs28) AND
     Val_b(t28,bs1028) = Val_b(t28,bs28) AND
     Val_b(t30,bs1028) = Val_b(t30,bs28) AND
     Val_b(t31,bs1028) = Val_b(t31,bs28) AND
     Val_b(t32,bs1028) = Val_b(t32,bs28) AND
     Val_b(t33,bs1028) = Val_b(t33,bs28) AND
     Val_b(t34,bs1028) = bb_plus(Val_b(t32,bs28),Val_b(t33,bs28)) AND
     Val_b(t35,bs1028) = Val_b(t35,bs28) AND
     Val_b(t36,bs1028) = Val_b(t36,bs28))
bs1028_bs29_ax: AXIOM beh_transition(bt40)
  IMPLIES
    (Val_b(state,bs29) = Val_b(state,bs1028) AND
     Val_b(st,bs29) = Val_b(st,bs1028) AND
     Val_b(hl,bs29) = Val_b(hl,bs1028) AND
     Val_b(fl,bs29) = Val_b(fl,bs1028) AND
     Val_b(cof,bs29) = Val_b(cof,bs1028) AND
     Val_b(tol,bs29) = Val_b(tol,bs1028) AND
     Val_b(tos,bs29) = Val_b(tos,bs1028) AND
     Val_b(t18,bs29) = Val_b(t18,bs1028) AND
     Val_b(t23,bs29) = Val_b(t23,bs1028) AND
     Val_b(t24,bs29) = Val_b(t24,bs1028) AND
     Val_b(t25,bs29) = Val_b(t25,bs1028) AND
     Val_b(t28,bs29) = Val_b(t28,bs1028) AND
     Val_b(t30,bs29) = Val_b(t30,bs1028) AND
     Val_b(t31,bs29) = Val_b(t31,bs1028) AND
     Val_b(t32,bs29) = Val_b(t32,bs1028) AND
     Val_b(t33,bs29) = Val_b(t33,bs1028) AND
     Val_b(t34,bs29) = Val_b(t34,bs1028) AND
     Val_b(t35,bs29) = Val_b(t35,bs1028) AND
     Val_b(t36,bs29) = Val_b(t36,bs1028))
bs29_bs30_ax: AXIOM beh_transition(bt41)
  IMPLIES
    (Val_b(state,bs30) = CONST_0 AND
     Val_b(st,bs30) = Val_b(st,bs29) AND
     Val_b(hl,bs30) = Val_b(hl,bs29) AND
     Val_b(fl,bs30) = Val_b(fl,bs29) AND
     Val_b(cof,bs30) = Val_b(cof,bs29) AND
     Val_b(tol,bs30) = Val_b(tol,bs29) AND
     Val_b(tos,bs30) = Val_b(tos,bs29) AND
     Val_b(t18,bs30) = Val_b(t18,bs29) AND
     Val_b(t23,bs30) = Val_b(t23,bs29) AND
     Val_b(t24,bs30) = Val_b(t24,bs29) AND
     Val_b(t25,bs30) = Val_b(t25,bs29) AND
     Val_b(t28,bs30) = Val_b(t28,bs29) AND
     Val_b(t30,bs30) = Val_b(t30,bs29) AND
     Val_b(t31,bs30) = Val_b(t31,bs29) AND
     Val_b(t32,bs30) = Val_b(t32,bs29) AND
     Val_b(t33,bs30) = Val_b(t33,bs29) AND
     Val_b(t34,bs30) = Val_b(t34,bs29) AND
     Val_b(t35,bs30) = Val_b(t35,bs29) AND
     Val_b(t36,bs30) = Val_b(t36,bs29))
bs30_bs42_ax: AXIOM beh_transition(bt42)
  IMPLIES
    (Val_b(state,bs42) = Val_b(state,bs30) AND
     Val_b(st,bs42) = CONST_0 AND
     Val_b(hl,bs42) = Val_b(hl,bs30) AND
     Val_b(fl,bs42) = Val_b(fl,bs30) AND
     Val_b(cof,bs42) = Val_b(cof,bs30) AND
     Val_b(tol,bs42) = Val_b(tol,bs30) AND
     Val_b(tos,bs42) = Val_b(tos,bs30) AND
     Val_b(t18,bs42) = Val_b(t18,bs30) AND
     Val_b(t23,bs42) = Val_b(t23,bs30) AND
     Val_b(t24,bs42) = Val_b(t24,bs30) AND
     Val_b(t25,bs42) = Val_b(t25,bs30) AND
     Val_b(t28,bs42) = Val_b(t28,bs30) AND
     Val_b(t30,bs42) = Val_b(t30,bs30) AND
     Val_b(t31,bs42) = Val_b(t31,bs30) AND
     Val_b(t32,bs42) = Val_b(t32,bs30) AND
     Val_b(t33,bs42) = Val_b(t33,bs30) AND
     Val_b(t34,bs42) = Val_b(t34,bs30) AND
     Val_b(t35,bs42) = Val_b(t35,bs30) AND
     Val_b(t36,bs42) = Val_b(t36,bs30))
bs1028_bs31_ax: AXIOM beh_transition(bt43)
  IMPLIES
    (Val_b(state,bs31) = Val_b(state,bs1028) AND
     Val_b(st,bs31) = Val_b(st,bs1028) AND
     Val_b(hl,bs31) = Val_b(hl,bs1028) AND
     Val_b(fl,bs31) = Val_b(fl,bs1028) AND
     Val_b(cof,bs31) = Val_b(cof,bs1028) AND
     Val_b(tol,bs31) = Val_b(tol,bs1028) AND
     Val_b(tos,bs31) = Val_b(tos,bs1028) AND
     Val_b(t18,bs31) = Val_b(t18,bs1028) AND
     Val_b(t23,bs31) = Val_b(t23,bs1028) AND
     Val_b(t24,bs31) = Val_b(t24,bs1028) AND
     Val_b(t25,bs31) = Val_b(t25,bs1028) AND
     Val_b(t28,bs31) = Val_b(t28,bs1028) AND
     Val_b(t30,bs31) = Val_b(t30,bs1028) AND
     Val_b(t31,bs31) = Val_b(t31,bs1028) AND
     Val_b(t32,bs31) = Val_b(t32,bs1028) AND
     Val_b(t33,bs31) = Val_b(t33,bs1028) AND
     Val_b(t34,bs31) = Val_b(t34,bs1028) AND
     Val_b(t35,bs31) = Val_b(t35,bs1028) AND
     Val_b(t36,bs31) = Val_b(t36,bs1028))
bs31_bs32_ax: AXIOM beh_transition(bt44)
  IMPLIES
    (Val_b(state,bs32) = CONST_0 AND
     Val_b(st,bs32) = Val_b(st,bs31) AND
     Val_b(hl,bs32) = Val_b(hl,bs31) AND
     Val_b(fl,bs32) = Val_b(fl,bs31) AND
     Val_b(cof,bs32) = Val_b(cof,bs31) AND
     Val_b(tol,bs32) = Val_b(tol,bs31) AND
     Val_b(tos,bs32) = Val_b(tos,bs31) AND
     Val_b(t18,bs32) = Val_b(t18,bs31) AND
     Val_b(t23,bs32) = Val_b(t23,bs31) AND
     Val_b(t24,bs32) = Val_b(t24,bs31) AND
     Val_b(t25,bs32) = Val_b(t25,bs31) AND
     Val_b(t28,bs32) = Val_b(t28,bs31) AND
     Val_b(t30,bs32) = Val_b(t30,bs31) AND
     Val_b(t31,bs32) = Val_b(t31,bs31) AND
     Val_b(t32,bs32) = Val_b(t32,bs31) AND
     Val_b(t33,bs32) = Val_b(t33,bs31) AND
     Val_b(t34,bs32) = Val_b(t34,bs31) AND
     Val_b(t35,bs32) = Val_b(t35,bs31) AND
     Val_b(t36,bs32) = Val_b(t36,bs31))
bs32_bs42_ax: AXIOM beh_transition(bt45)
  IMPLIES
    (Val_b(state,bs42) = Val_b(state,bs32) AND
     Val_b(st,bs42) = CONST_0 AND
     Val_b(hl,bs42) = Val_b(hl,bs32) AND
     Val_b(fl,bs42) = Val_b(fl,bs32) AND
     Val_b(cof,bs42) = Val_b(cof,bs32) AND
     Val_b(tol,bs42) = Val_b(tol,bs32) AND
     Val_b(tos,bs42) = Val_b(tos,bs32) AND
     Val_b(t18,bs42) = Val_b(t18,bs32) AND
     Val_b(t23,bs42) = Val_b(t23,bs32) AND
     Val_b(t24,bs42) = Val_b(t24,bs32) AND
     Val_b(t25,bs42) = Val_b(t25,bs32) AND
     Val_b(t28,bs42) = Val_b(t28,bs32) AND
     Val_b(t30,bs42) = Val_b(t30,bs32) AND
     Val_b(t31,bs42) = Val_b(t31,bs32) AND
     Val_b(t32,bs42) = Val_b(t32,bs32) AND
     Val_b(t33,bs42) = Val_b(t33,bs32) AND
     Val_b(t34,bs42) = Val_b(t34,bs32) AND
     Val_b(t35,bs42) = Val_b(t35,bs32) AND
     Val_b(t36,bs42) = Val_b(t36,bs32))
```

```
bs1023_bs33_ax: AXIOM beh_transition(bt46)
  IMPLIES
    (Val_b(state,bs33) = Val_b(state,bs1023) AND
     Val_b(st,bs33) = Val_b(st,bs1023) AND
     Val_b(hl,bs33) = Val_b(hl,bs1023) AND
     Val_b(fl,bs33) = Val_b(fl,bs1023) AND
     Val_b(cof,bs33) = Val_b(cof,bs1023) AND
     Val_b(tol,bs33) = Val_b(tol,bs1023) AND
     Val_b(tos,bs33) = Val_b(tos,bs1023) AND
     Val_b(t18,bs33) = Val_b(t18,bs1023) AND
     Val_b(t23,bs33) = Val_b(t23,bs1023) AND
     Val_b(t24,bs33) = Val_b(t24,bs1023) AND
     Val_b(t25,bs33) = Val_b(t25,bs1023) AND
     Val_b(t28,bs33) = Val_b(t28,bs1023) AND
     Val_b(t30,bs33) = Val_b(t30,bs1023) AND
     Val_b(t31,bs33) = Val_b(t31,bs1023) AND
     Val_b(t32,bs33) = Val_b(t32,bs1023) AND
     Val_b(t33,bs33) = Val_b(t33,bs1023) AND
     Val_b(t34,bs33) = Val_b(t34,bs1023) AND
     Val_b(t35,bs33) = Val_b(t35,bs1023) AND
     Val_b(t36,bs33) = Val_b(t36,bs1023))
bs33_bs1033_ax: AXIOM beh_transition(bt47)
  IMPLIES
    (Val_b(state,bs1033) = Val_b(state,bs33) AND
     Val_b(st,bs1033) = Val_b(st,bs33) AND
     Val_b(hl,bs1033) = Val_b(hl,bs33) AND
     Val_b(fl,bs1033) = Val_b(fl,bs33) AND
     Val_b(cof,bs1033) = Val_b(cof,bs33) AND
     Val_b(tol,bs1033) = Val_b(tol,bs33) AND
     Val_b(tos,bs1033) = Val_b(tos,bs33) AND
     Val_b(t18,bs1033) = Val_b(t18,bs33) AND
     Val_b(t23,bs1033) = Val_b(t23,bs33) AND
     Val_b(t24,bs1033) = Val_b(t24,bs33) AND
     Val_b(t25,bs1033) = Val_b(t25,bs33) AND
     Val_b(t28,bs1033) = Val_b(t28,bs33) AND
     Val_b(t30,bs1033) = Val_b(t30,bs33) AND
     Val_b(t31,bs1033) = Val_b(t31,bs33) AND
     Val_b(t32,bs1033) = Val_b(t32,bs33) AND
     Val_b(t33,bs1033) = Val_b(t33,bs33) AND
     Val_b(t34,bs1033) = Val_b(t34,bs33) AND
     Val_b(t35,bs1033) = bb_eq(Val_b(state,bs33),CONST_0) AND
     Val_b(t36,bs1033) = Val_b(t36,bs33))
bs1033_bs34_ax: AXIOM beh_transition(bt48)
  IMPLIES
    (Val_b(state,bs34) = Val_b(state,bs1033) AND
     Val_b(st,bs34) = Val_b(st,bs1033) AND
     Val_b(hl,bs34) = Val_b(hl,bs1033) AND
     Val_b(fl,bs34) = Val_b(fl,bs1033) AND
     Val_b(cof,bs34) = Val_b(cof,bs1033) AND
     Val_b(tol,bs34) = Val_b(tol,bs1033) AND
     Val_b(tos,bs34) = Val_b(tos,bs1033) AND
     Val_b(t18,bs34) = Val_b(t18,bs1033) AND
     Val_b(t23,bs34) = Val_b(t23,bs1033) AND
     Val_b(t24,bs34) = Val_b(t24,bs1033) AND
     Val_b(t25,bs34) = Val_b(t25,bs1033) AND
     Val_b(t28,bs34) = Val_b(t28,bs1033) AND
     Val_b(t30,bs34) = Val_b(t30,bs1033) AND
     Val_b(t31,bs34) = Val_b(t31,bs1033) AND
     Val_b(t32,bs34) = Val_b(t32,bs1033) AND
     Val_b(t33,bs34) = Val_b(t33,bs1033) AND
     Val_b(t34,bs34) = Val_b(t34,bs1033) AND
     Val_b(t35,bs34) = Val_b(t35,bs1033) AND
     Val_b(t36,bs34) = Val_b(t36,bs1033))
bs34_bs35_ax: AXIOM beh_transition(bt49)
  IMPLIES
    (Val_b(state,bs35) = Val_b(state,bs34) AND
     Val_b(st,bs35) = Val_b(st,bs34) AND
     Val_b(hl,bs35) = CONST_0 AND
     Val_b(fl,bs35) = Val_b(fl,bs34) AND
     Val_b(cof,bs35) = Val_b(cof,bs34) AND
     Val_b(tol,bs35) = Val_b(tol,bs34) AND
     Val_b(tos,bs35) = Val_b(tos,bs34) AND
     Val_b(t18,bs35) = Val_b(t18,bs34) AND
     Val_b(t23,bs35) = Val_b(t23,bs34) AND
     Val_b(t24,bs35) = Val_b(t24,bs34) AND
     Val_b(t25,bs35) = Val_b(t25,bs34) AND
     Val_b(t28,bs35) = Val_b(t28,bs34) AND
     Val_b(t30,bs35) = Val_b(t30,bs34) AND
     Val_b(t31,bs35) = Val_b(t31,bs34) AND
     Val_b(t32,bs35) = Val_b(t32,bs34) AND
     Val_b(t33,bs35) = Val_b(t33,bs34) AND
     Val_b(t34,bs35) = Val_b(t34,bs34) AND
     Val_b(t35,bs35) = Val_b(t35,bs34) AND
     Val_b(t36,bs35) = Val_b(t36,bs34))
bs35_bs36_ax: AXIOM beh_transition(bt50)
  IMPLIES
    (Val_b(state,bs36) = Val_b(state,bs35) AND
     Val_b(st,bs36) = Val_b(st,bs35) AND
     Val_b(hl,bs36) = Val_b(hl,bs35) AND
     Val_b(fl,bs36) = CONST_0 AND
     Val_b(cof,bs36) = Val_b(cof,bs35) AND
     Val_b(tol,bs36) = Val_b(tol,bs35) AND
     Val_b(tos,bs36) = Val_b(tos,bs35) AND
     Val_b(t18,bs36) = Val_b(t18,bs35) AND
     Val_b(t23,bs36) = Val_b(t23,bs35) AND
     Val_b(t24,bs36) = Val_b(t24,bs35) AND
     Val_b(t25,bs36) = Val_b(t25,bs35) AND
     Val_b(t28,bs36) = Val_b(t28,bs35) AND
     Val_b(t30,bs36) = Val_b(t30,bs35) AND
     Val_b(t31,bs36) = Val_b(t31,bs35) AND
     Val_b(t32,bs36) = Val_b(t32,bs35) AND
     Val_b(t33,bs36) = Val_b(t33,bs35) AND
     Val_b(t34,bs36) = Val_b(t34,bs35) AND
     Val_b(t35,bs36) = Val_b(t35,bs35) AND
     Val_b(t36,bs36) = Val_b(t36,bs35))
bs36_bs1036_ax: AXIOM beh_transition(bt51)
  IMPLIES
    (Val_b(state,bs1036) = Val_b(state,bs36) AND
     Val_b(st,bs1036) = Val_b(st,bs36) AND
     Val_b(hl,bs1036) = Val_b(hl,bs36) AND
     Val_b(fl,bs1036) = Val_b(fl,bs36) AND
     Val_b(cof,bs1036) = Val_b(cof,bs36) AND
     Val_b(tol,bs1036) = Val_b(tol,bs36) AND
     Val_b(tos,bs1036) = Val_b(tos,bs36) AND
     Val_b(t18,bs1036) = Val_b(t18,bs36) AND
     Val_b(t23,bs1036) = Val_b(t23,bs36) AND
     Val_b(t24,bs1036) = Val_b(t24,bs36) AND
     Val_b(t25,bs1036) = Val_b(t25,bs36) AND
     Val_b(t28,bs1036) = Val_b(t28,bs36) AND
     Val_b(t30,bs1036) = Val_b(t30,bs36) AND
     Val_b(t31,bs1036) = Val_b(t31,bs36) AND
     Val_b(t32,bs1036) = Val_b(t32,bs36) AND
     Val_b(t33,bs1036) = Val_b(t33,bs36) AND
     Val_b(t34,bs1036) = Val_b(t34,bs36) AND
     Val_b(t35,bs1036) = Val_b(t35,bs36) AND
     Val_b(t36,bs1036) = bb_eq(Val_b(tos,bs36),CONST_0))
bs1036_bs37_ax: AXIOM beh_transition(bt52)
  IMPLIES
    (Val_b(state,bs37) = Val_b(state,bs1036) AND
     Val_b(st,bs37) = Val_b(st,bs1036) AND
     Val_b(hl,bs37) = Val_b(hl,bs1036) AND
     Val_b(fl,bs37) = Val_b(fl,bs1036) AND
     Val_b(cof,bs37) = Val_b(cof,bs1036) AND
     Val_b(tol,bs37) = Val_b(tol,bs1036) AND
     Val_b(tos,bs37) = Val_b(tos,bs1036) AND
     Val_b(t18,bs37) = Val_b(t18,bs1036) AND
     Val_b(t23,bs37) = Val_b(t23,bs1036) AND
     Val_b(t24,bs37) = Val_b(t24,bs1036) AND
     Val_b(t25,bs37) = Val_b(t25,bs1036) AND
     Val_b(t28,bs37) = Val_b(t28,bs1036) AND
     Val_b(t30,bs37) = Val_b(t30,bs1036) AND
     Val_b(t31,bs37) = Val_b(t31,bs1036) AND
     Val_b(t32,bs37) = Val_b(t32,bs1036) AND
     Val_b(t33,bs37) = Val_b(t33,bs1036) AND
     Val_b(t34,bs37) = Val_b(t34,bs1036) AND
     Val_b(t35,bs37) = Val_b(t35,bs1036) AND
     Val_b(t36,bs37) = Val_b(t36,bs1036))
bs37_bs38_ax: AXIOM beh_transition(bt53)
  IMPLIES
    (Val_b(state,bs38) = CONST_0 AND
     Val_b(st,bs38) = Val_b(st,bs37) AND
     Val_b(hl,bs38) = Val_b(hl,bs37) AND
     Val_b(fl,bs38) = Val_b(fl,bs37) AND
     Val_b(cof,bs38) = Val_b(cof,bs37) AND
     Val_b(tol,bs38) = Val_b(tol,bs37) AND
     Val_b(tos,bs38) = Val_b(tos,bs37) AND
     Val_b(t18,bs38) = Val_b(t18,bs37) AND
     Val_b(t23,bs38) = Val_b(t23,bs37) AND
     Val_b(t24,bs38) = Val_b(t24,bs37) AND
     Val_b(t25,bs38) = Val_b(t25,bs37) AND
     Val_b(t28,bs38) = Val_b(t28,bs37) AND
     Val_b(t30,bs38) = Val_b(t30,bs37) AND
     Val_b(t31,bs38) = Val_b(t31,bs37) AND
     Val_b(t32,bs38) = Val_b(t32,bs37) AND
     Val_b(t33,bs38) = Val_b(t33,bs37) AND
     Val_b(t34,bs38) = Val_b(t34,bs37) AND
     Val_b(t35,bs38) = Val_b(t35,bs37) AND
     Val_b(t36,bs38) = Val_b(t36,bs37))
bs38_bs42_ax: AXIOM beh_transition(bt54)
  IMPLIES
    (Val_b(state,bs42) = Val_b(state,bs38) AND
     Val_b(st,bs42) = CONST_0 AND
     Val_b(hl,bs42) = Val_b(hl,bs38) AND
     Val_b(fl,bs42) = Val_b(fl,bs38) AND
     Val_b(cof,bs42) = Val_b(cof,bs38) AND
     Val_b(tol,bs42) = Val_b(tol,bs38) AND
     Val_b(tos,bs42) = Val_b(tos,bs38) AND
     Val_b(t18,bs42) = Val_b(t18,bs38) AND
     Val_b(t23,bs42) = Val_b(t23,bs38) AND
     Val_b(t24,bs42) = Val_b(t24,bs38) AND
     Val_b(t25,bs42) = Val_b(t25,bs38) AND
     Val_b(t28,bs42) = Val_b(t28,bs38) AND
     Val_b(t30,bs42) = Val_b(t30,bs38) AND
     Val_b(t31,bs42) = Val_b(t31,bs38) AND
     Val_b(t32,bs42) = Val_b(t32,bs38) AND
     Val_b(t33,bs42) = Val_b(t33,bs38) AND
     Val_b(t34,bs42) = Val_b(t34,bs38) AND
     Val_b(t35,bs42) = Val_b(t35,bs38) AND
     Val_b(t36,bs42) = Val_b(t36,bs38))
bs1036_bs39_ax: AXIOM beh_transition(bt55)
  IMPLIES
    (Val_b(state,bs39) = Val_b(state,bs1036) AND
     Val_b(st,bs39) = Val_b(st,bs1036) AND
     Val_b(hl,bs39) = Val_b(hl,bs1036) AND
     Val_b(fl,bs39) = Val_b(fl,bs1036) AND
     Val_b(cof,bs39) = Val_b(cof,bs1036) AND
     Val_b(tol,bs39) = Val_b(tol,bs1036) AND
     Val_b(tos,bs39) = Val_b(tos,bs1036) AND
     Val_b(t18,bs39) = Val_b(t18,bs1036) AND
     Val_b(t23,bs39) = Val_b(t23,bs1036) AND
     Val_b(t24,bs39) = Val_b(t24,bs1036) AND
     Val_b(t25,bs39) = Val_b(t25,bs1036) AND
     Val_b(t28,bs39) = Val_b(t28,bs1036) AND
     Val_b(t30,bs39) = Val_b(t30,bs1036) AND
     Val_b(t31,bs39) = Val_b(t31,bs1036) AND
     Val_b(t32,bs39) = Val_b(t32,bs1036) AND
     Val_b(t33,bs39) = Val_b(t33,bs1036) AND
     Val_b(t34,bs39) = Val_b(t34,bs1036) AND
     Val_b(t35,bs39) = Val_b(t35,bs1036) AND
     Val_b(t36,bs39) = Val_b(t36,bs1036))
bs39_bs40_ax: AXIOM beh_transition(bt56)
  IMPLIES
    (Val_b(state,bs40) = CONST_0 AND
     Val_b(st,bs40) = Val_b(st,bs39) AND
     Val_b(hl,bs40) = Val_b(hl,bs39) AND
     Val_b(fl,bs40) = Val_b(fl,bs39) AND
     Val_b(cof,bs40) = Val_b(cof,bs39) AND
     Val_b(tol,bs40) = Val_b(tol,bs39) AND
     Val_b(tos,bs40) = Val_b(tos,bs39) AND
     Val_b(t18,bs40) = Val_b(t18,bs39) AND
     Val_b(t23,bs40) = Val_b(t23,bs39) AND
     Val_b(t24,bs40) = Val_b(t24,bs39) AND
     Val_b(t25,bs40) = Val_b(t25,bs39) AND
     Val_b(t28,bs40) = Val_b(t28,bs39) AND
     Val_b(t30,bs40) = Val_b(t30,bs39) AND
     Val_b(t31,bs40) = Val_b(t31,bs39) AND
     Val_b(t32,bs40) = Val_b(t32,bs39) AND
     Val_b(t33,bs40) = Val_b(t33,bs39) AND
     Val_b(t34,bs40) = Val_b(t34,bs39) AND
     Val_b(t35,bs40) = Val_b(t35,bs39) AND
     Val_b(t36,bs40) = Val_b(t36,bs39))
```

```
bs40_bs42_ax: AXIOM beh_transition(bt57)
  IMPLIES
    (Val_b(state,bs42) = Val_b(state,bs40) AND
     Val_b(st,bs42) = CONST_0 AND
     Val_b(hl,bs42) = Val_b(hl,bs40) AND
     Val_b(fl,bs42) = Val_b(fl,bs40) AND
     Val_b(cof,bs42) = Val_b(cof,bs40) AND
     Val_b(tol,bs42) = Val_b(tol,bs40) AND
     Val_b(tos,bs42) = Val_b(tos,bs40) AND
     Val_b(t18,bs42) = Val_b(t18,bs40) AND
     Val_b(t23,bs42) = Val_b(t23,bs40) AND
     Val_b(t24,bs42) = Val_b(t24,bs40) AND
     Val_b(t25,bs42) = Val_b(t25,bs40) AND
     Val_b(t28,bs42) = Val_b(t28,bs40) AND
     Val_b(t30,bs42) = Val_b(t30,bs40) AND
     Val_b(t31,bs42) = Val_b(t31,bs40) AND
     Val_b(t32,bs42) = Val_b(t32,bs40) AND
     Val_b(t33,bs42) = Val_b(t33,bs40) AND
     Val_b(t34,bs42) = Val_b(t34,bs40) AND
     Val_b(t35,bs42) = Val_b(t35,bs40) AND
     Val_b(t36,bs42) = Val_b(t36,bs40))
bs1033_bs42_ax: AXIOM beh_transition(bt58)
  IMPLIES
    (Val_b(state,bs42) = Val_b(state,bs1033) AND
     Val_b(st,bs42) = Val_b(st,bs1033) AND
     Val_b(hl,bs42) = Val_b(hl,bs1033) AND
     Val_b(fl,bs42) = Val_b(fl,bs1033) AND
     Val_b(cof,bs42) = Val_b(cof,bs1033) AND
     Val_b(tol,bs42) = Val_b(tol,bs1033) AND
     Val_b(tos,bs42) = Val_b(tos,bs1033) AND
     Val_b(t18,bs42) = Val_b(t18,bs1033) AND
     Val_b(t23,bs42) = Val_b(t23,bs1033) AND
     Val_b(t24,bs42) = Val_b(t24,bs1033) AND
     Val_b(t25,bs42) = Val_b(t25,bs1033) AND
     Val_b(t28,bs42) = Val_b(t28,bs1033) AND
     Val_b(t30,bs42) = Val_b(t30,bs1033) AND
     Val_b(t31,bs42) = Val_b(t31,bs1033) AND
     Val_b(t32,bs42) = Val_b(t32,bs1033) AND
     Val_b(t33,bs42) = Val_b(t33,bs1033) AND
     Val_b(t34,bs42) = Val_b(t34,bs1033) AND
     Val_b(t35,bs42) = Val_b(t35,bs1033) AND
     Val_b(t36,bs42) = Val_b(t36,bs1033))
```

Figure B.6: **TLC Behavior Axioms**

```
% Data-Path Axioms

asserta_register_1_ld : PRED[rtl_state]
asserta_register_2_ld : PRED[rtl_state]
asserta_register_3_ld : PRED[rtl_state]
asserta_register_4_ld : PRED[rtl_state]
asserta_register_5_ld : PRED[rtl_state]
asserta_register_6_ld : PRED[rtl_state]
asserta_register_7_ld : PRED[rtl_state]
asserta_register_8_ld : PRED[rtl_state]
asserta_mux_1_sel_0 : PRED[rtl_state]
asserta_mux_1_sel_1 : PRED[rtl_state]
asserta_mux_1_sel_2 : PRED[rtl_state]
asserta_mux_2_sel_0 : PRED[rtl_state]
asserta_mux_2_sel_1 : PRED[rtl_state]
asserta_mux_2_sel_2 : PRED[rtl_state]
asserta_mux_3_sel_0 : PRED[rtl_state]
asserta_mux_3_sel_1 : PRED[rtl_state]
asserta_mux_3_sel_2 : PRED[rtl_state]
asserta_mux_3_sel_3 : PRED[rtl_state]
asserta_mux_4_sel_0 : PRED[rtl_state]
asserta_mux_4_sel_1 : PRED[rtl_state]
asserta_mux_4_sel_2 : PRED[rtl_state]
asserta_mux_5_sel_0 : PRED[rtl_state]
asserta_mux_5_sel_1 : PRED[rtl_state]
asserta_mux_6_sel_0 : PRED[rtl_state]
asserta_mux_6_sel_1 : PRED[rtl_state]
asserta_mux_7_sel_0 : PRED[rtl_state]
asserta_mux_8_sel_0 : PRED[rtl_state]
asserta_mux_8_sel_1 : PRED[rtl_state]
asserta_mux_9_sel_0 : PRED[rtl_state]

asserta_register_1_ld_ax : AXIOM (FORALL (s : rtl_state) :
        asserta_register_1_ld(s) = Control_Signal(s,1))
asserta_register_2_ld_ax : AXIOM (FORALL (s : rtl_state) :
        asserta_register_2_ld(s) = Control_Signal(s,3))
asserta_register_3_ld_ax : AXIOM (FORALL (s : rtl_state) :
        asserta_register_3_ld(s) = Control_Signal(s,5))
asserta_register_4_ld_ax : AXIOM (FORALL (s : rtl_state) :
        asserta_register_4_ld(s) = Control_Signal(s,7))
asserta_register_5_ld_ax : AXIOM (FORALL (s : rtl_state) :
        asserta_register_5_ld(s) = Control_Signal(s,9))
asserta_register_6_ld_ax : AXIOM (FORALL (s : rtl_state) :
        asserta_register_6_ld(s) = Control_Signal(s,11))
asserta_register_7_ld_ax : AXIOM (FORALL (s : rtl_state) :
        asserta_register_7_ld(s) = Control_Signal(s,13))
asserta_register_8_ld_ax : AXIOM (FORALL (s : rtl_state) :
        asserta_register_8_ld(s) = Control_Signal(s,15))
asserta_mux_1_sel_0_ax : AXIOM (FORALL (s : rtl_state) :
        asserta_mux_1_sel_0(s) = Control_Signal(s,18))
asserta_mux_1_sel_1_ax : AXIOM (FORALL (s : rtl_state) :
        asserta_mux_1_sel_1(s) = Control_Signal(s,17))
asserta_mux_1_sel_2_ax : AXIOM (FORALL (s : rtl_state) :
        asserta_mux_1_sel_2(s) = Control_Signal(s,16))
asserta_mux_2_sel_0_ax : AXIOM (FORALL (s : rtl_state) :
        asserta_mux_2_sel_0(s) = Control_Signal(s,21))
asserta_mux_2_sel_1_ax : AXIOM (FORALL (s : rtl_state) :
        asserta_mux_2_sel_1(s) = Control_Signal(s,20))
asserta_mux_2_sel_2_ax : AXIOM (FORALL (s : rtl_state) :
        asserta_mux_2_sel_2(s) = Control_Signal(s,19))
asserta_mux_3_sel_0_ax : AXIOM (FORALL (s : rtl_state) :
        asserta_mux_3_sel_0(s) = Control_Signal(s,25))
asserta_mux_3_sel_1_ax : AXIOM (FORALL (s : rtl_state) :
        asserta_mux_3_sel_1(s) = Control_Signal(s,24))
asserta_mux_3_sel_2_ax : AXIOM (FORALL (s : rtl_state) :
        asserta_mux_3_sel_2(s) = Control_Signal(s,23))
asserta_mux_3_sel_3_ax : AXIOM (FORALL (s : rtl_state) :
        asserta_mux_3_sel_3(s) = Control_Signal(s,22))
asserta_mux_4_sel_0_ax : AXIOM (FORALL (s : rtl_state) :
        asserta_mux_4_sel_0(s) = Control_Signal(s,28))
asserta_mux_4_sel_1_ax : AXIOM (FORALL (s : rtl_state) :
        asserta_mux_4_sel_1(s) = Control_Signal(s,27))
```

```
asserta_mux_4_sel_2_ax : AXIOM (FORALL (s : rtl_state) :
        asserta_mux_4_sel_2(s) = Control_Signal(s,26))
asserta_mux_5_sel_0_ax : AXIOM (FORALL (s : rtl_state) :
        asserta_mux_5_sel_0(s) = Control_Signal(s,30))
asserta_mux_5_sel_1_ax : AXIOM (FORALL (s : rtl_state) :
        asserta_mux_5_sel_1(s) = Control_Signal(s,29))
asserta_mux_6_sel_0_ax : AXIOM (FORALL (s : rtl_state) :
        asserta_mux_6_sel_0(s) = Control_Signal(s,32))
asserta_mux_6_sel_1_ax : AXIOM (FORALL (s : rtl_state) :
        asserta_mux_6_sel_1(s) = Control_Signal(s,31))
asserta_mux_7_sel_0_ax : AXIOM (FORALL (s : rtl_state) :
        asserta_mux_7_sel_0(s) = Control_Signal(s,33))
asserta_mux_8_sel_0_ax : AXIOM (FORALL (s : rtl_state) :
        asserta_mux_8_sel_0(s) = Control_Signal(s,35))
asserta_mux_8_sel_1_ax : AXIOM (FORALL (s : rtl_state) :
        asserta_mux_8_sel_1(s) = Control_Signal(s,34))
asserta_mux_9_sel_0_ax : AXIOM (FORALL (s : rtl_state) :
        asserta_mux_9_sel_0(s) = Control_Signal(s,36))
asserta_eq_0_ax :
  AXIOM asserta_eq(asserta_register_1_out1,asserta_mux_8_out1,asserta_eq_0_out1)
asserta_eq_1_ax :
  AXIOM asserta_eq(asserta_register_1_out1,asserta_mux_9_out1,asserta_eq_1_out1)
asserta_eq_2_ax :
  AXIOM asserta_eq(asserta_register_2_out1,asserta_constreg_0_out1,asserta_eq_2_out1)
asserta_adder_3_ax :
  AXIOM asserta_adder(asserta_register_3_out1,asserta_register_4_out1,asserta_adder_3_out1)
asserta_adder_4_ax :
  AXIOM asserta_adder(asserta_register_3_out1,asserta_register_4_out1,asserta_adder_4_out1)
asserta_register_1_ax :
  AXIOM asserta_register(asserta_mux_1_out1,asserta_register_1_ld,asserta_register_1_out1)
asserta_register_2_ax :
  AXIOM asserta_register(asserta_mux_2_out1,asserta_register_2_ld,asserta_register_2_out1)
asserta_register_3_ax :
  AXIOM asserta_register(asserta_mux_3_out1,asserta_register_3_ld,asserta_register_3_out1)
asserta_register_4_ax :
  AXIOM asserta_register(asserta_mux_4_out1,asserta_register_4_ld,asserta_register_4_out1)
asserta_register_5_ax :
  AXIOM asserta_register(asserta_mux_5_out1,asserta_register_5_ld,asserta_register_5_out1)
asserta_register_6_ax :
  AXIOM asserta_register(asserta_mux_6_out1,asserta_register_6_ld,asserta_register_6_out1)
asserta_register_7_ax :
  AXIOM asserta_register(asserta_mux_7_out1,asserta_register_7_ld,asserta_register_7_out1)
asserta_register_8_ax :
  AXIOM asserta_register(asserta_eq_0_out1,asserta_register_8_ld,asserta_register_8_out1)
asserta_mux_1_ax :
  AXIOM  asserta_mux_8_1(asserta_register_4_out1,asserta_constreg_0_out1,UNCONNECTED,
                         asserta_register_2_out1,asserta_constreg_0_out1,asserta_constreg_0_out1,
                         asserta_register_3_out1,UNCONNECTED,asserta_mux_1_sel_0,
                         asserta_mux_1_sel_1,asserta_mux_1_sel_2,asserta_mux_1_out1)
asserta_mux_2_ax :
  AXIOM  asserta_mux_8_1(asserta_register_3_out1,asserta_constreg_0_out1,UNCONNECTED,
                         asserta_register_5_out1,asserta_constreg_0_out1,asserta_register_4_out1,
                         asserta_constreg_0_out1,UNCONNECTED,asserta_mux_2_sel_0,
                         asserta_mux_2_sel_1,asserta_mux_2_sel_2,asserta_mux_2_out1)
asserta_mux_3_ax :
  AXIOM  asserta_mux_16_1(asserta_adder_3_out1,asserta_eq_1_out1,asserta_constreg_0_out1,
                          asserta_register_6_out1,asserta_constreg_0_out1,asserta_constreg_0_out1,
                          asserta_adder_4_out1,asserta_constreg_0_out1,asserta_register_5_out1,
                          asserta_register_1_out1,asserta_register_4_out1,UNCONNECTED,UNCONNECTED,
                          UNCONNECTED,UNCONNECTED,UNCONNECTED,asserta_mux_3_sel_0,asserta_mux_3_sel_1,
                          asserta_mux_3_sel_2,asserta_mux_3_sel_3,asserta_mux_3_out1)
asserta_mux_4_ax :
  AXIOM  asserta_mux_8_1(UNCONNECTED,asserta_register_7_out1,asserta_eq_2_out1,asserta_constreg_0_out1,
                         asserta_constreg_0_out1,asserta_register_6_out1,asserta_register_2_out1,
                         asserta_register_5_out1,asserta_mux_4_sel_0,asserta_mux_4_sel_1,
                         asserta_mux_4_sel_2,asserta_mux_4_out1)
asserta_mux_5_ax :
  AXIOM  asserta_mux_4_1(asserta_register_7_out1,asserta_constreg_cof_in0_out1,asserta_register_2_out1,
                         UNCONNECTED,asserta_mux_5_sel_0,asserta_mux_5_sel_1,asserta_mux_5_out1)
```

```
asserta_mux_6_ax :
  AXIOM  asserta_mux_4_1(asserta_eq_0_out1,asserta_constreg_tol_in1_out1,asserta_register_3_out1,
                         UNCONNECTED,asserta_mux_6_sel_0,asserta_mux_6_sel_1,asserta_mux_6_out1)
asserta_mux_7_ax :
  AXIOM  asserta_mux_2_1(asserta_constreg_tos_in2_out1,asserta_register_4_out1,asserta_mux_7_sel_0,
                         asserta_mux_7_out1)
asserta_mux_8_ax :
  AXIOM  asserta_mux_4_1(asserta_constreg_0_out1,asserta_constreg_0_out1,asserta_constreg_0_out1,
                         asserta_constreg_0_out1,asserta_mux_8_sel_0,asserta_mux_8_sel_1,
                         asserta_mux_8_out1)
asserta_mux_9_ax :
  AXIOM  asserta_mux_2_1(asserta_constreg_0_out1,asserta_constreg_0_out1,asserta_mux_9_sel_0,
                         asserta_mux_9_out1)
asserta_constreg_cof_in0_ax :
  AXIOM asserta_constreg(cof_in0,asserta_constreg_cof_in0_out1)
asserta_constreg_tol_in1_ax :
  AXIOM asserta_constreg(tol_in1,asserta_constreg_tol_in1_out1)
asserta_constreg_tos_in2_ax :
  AXIOM asserta_constreg(tos_in2,asserta_constreg_tos_in2_out1)
asserta_constreg_0_ax :
  AXIOM asserta_constreg(CONST_0,asserta_constreg_0_out1)
```

Figure B.7: **TLC Data-Path Axioms**

```
tlc_control     : THEORY
  BEGIN
    IMPORTING tlc_state, tlc_dcls
    Control_Signal(s: rtl_state, id: index) : bool =
      IF id=0 THEN FALSE
      ELSIF (id = 1) THEN
        CASES s OF
          ds7  : TRUE,
          ds9  : TRUE,
          ds15  : TRUE,
          ds19  : TRUE,
          ds23  : TRUE,
          ds25  : TRUE,
          ds29  : TRUE,
          ds33  : TRUE,
          ds37  : TRUE,
          ds41  : TRUE,
          ds47  : TRUE,
          ds51  : TRUE,
          ds55  : TRUE,
          ds59  : TRUE,
          ds63  : TRUE,
          ds67  : TRUE
          ELSE    FALSE
        ENDCASES
      ELSIF (id = 2) THEN FALSE
      ELSIF (id = 3) THEN
        CASES s OF
          ds3  : TRUE,
          ds7  : TRUE,
          ds9  : TRUE,
          ds15  : TRUE,
          ds19  : TRUE,
          ds25  : TRUE,
          ds29  : TRUE,
          ds33  : TRUE,
          ds37  : TRUE,
          ds39  : TRUE,
          ds41  : TRUE,
          ds47  : TRUE,
          ds51  : TRUE,
          ds57  : TRUE,
          ds59  : TRUE,
          ds63  : TRUE,
          ds67  : TRUE
          ELSE    FALSE
        ENDCASES
      ELSIF (id = 4) THEN FALSE
      ELSIF (id = 5) THEN
        CASES s OF
          ds3  : TRUE,
          ds9  : TRUE,
          ds11  : TRUE,
          ds13  : TRUE,
          ds15  : TRUE,
          ds17  : TRUE,
          ds19  : TRUE,
          ds25  : TRUE,
          ds27  : TRUE,
          ds29  : TRUE,
          ds31  : TRUE,
          ds33  : TRUE,
          ds39  : TRUE,
          ds41  : TRUE,
          ds43  : TRUE,
          ds45  : TRUE,
          ds47  : TRUE,
          ds49  : TRUE,
          ds51  : TRUE,
          ds57  : TRUE,
          ds59  : TRUE,
          ds61  : TRUE,
          ds63  : TRUE,
          ds65  : TRUE,
          ds67  : TRUE
          ELSE    FALSE
        ENDCASES
      ELSIF (id = 6) THEN FALSE
      ELSIF (id = 7) THEN
        CASES s OF
          ds3  : TRUE,
      ds9  : TRUE,
      ds13  : TRUE,
      ds15  : TRUE,
      ds17  : TRUE,
      ds19  : TRUE,
      ds27  : TRUE,
      ds29  : TRUE,
      ds31  : TRUE,
      ds33  : TRUE,
      ds39  : TRUE,
      ds41  : TRUE,
      ds45  : TRUE,
      ds47  : TRUE,
      ds49  : TRUE,
      ds51  : TRUE,
      ds57  : TRUE,
      ds61  : TRUE,
      ds63  : TRUE,
      ds65  : TRUE,
      ds67  : TRUE
      ELSE    FALSE
    ENDCASES
  ELSIF (id = 8) THEN FALSE
  ELSIF (id = 9) THEN
    CASES s OF
      ds1  : TRUE,
      ds3  : TRUE,
      ds39  : TRUE
      ELSE    FALSE
    ENDCASES
  ELSIF (id = 10) THEN FALSE
  ELSIF (id = 11) THEN
    CASES s OF
      ds1  : TRUE,
      ds3  : TRUE,
      ds53  : TRUE
      ELSE    FALSE
    ENDCASES
  ELSIF (id = 12) THEN FALSE
  ELSIF (id = 13) THEN
    CASES s OF
      ds1  : TRUE,
      ds3  : TRUE
      ELSE    FALSE
    ENDCASES
  ELSIF (id = 14) THEN FALSE
  ELSIF (id = 15) THEN
    CASES s OF
      ds5  : TRUE,
      ds21  : TRUE,
      ds35  : TRUE
      ELSE    FALSE
    ENDCASES
  ELSIF (id = 16) THEN
    CASES s OF
      ds9  : TRUE,
      ds15  : TRUE,
      ds19  : TRUE,
      ds29  : TRUE,
      ds33  : TRUE,
      ds41  : TRUE,
      ds47  : TRUE,
      ds51  : TRUE,
      ds59  : TRUE,
      ds63  : TRUE,
      ds67  : TRUE
      ELSE    FALSE
    ENDCASES
  ELSIF (id = 17) THEN
    CASES s OF
      ds7  : TRUE,
      ds15  : TRUE,
      ds19  : TRUE,
      ds29  : TRUE,
      ds33  : TRUE,
      ds37  : TRUE,
      ds47  : TRUE,
      ds51  : TRUE,
      ds55  : TRUE,
      ds63  : TRUE,
      ds67  : TRUE
      ELSE    FALSE
    ENDCASES
  ELSIF (id = 18) THEN
    CASES s OF
      ds7  : TRUE,
      ds25  : TRUE,
      ds37  : TRUE,
      ds41  : TRUE,
      ds55  : TRUE,
      ds59  : TRUE
      ELSE    FALSE
    ENDCASES
  ELSIF (id = 19) THEN
    CASES s OF
      ds15  : TRUE,
      ds19  : TRUE,
      ds29  : TRUE,
      ds33  : TRUE,
      ds39  : TRUE,
      ds41  : TRUE,
      ds47  : TRUE,
      ds51  : TRUE,
      ds59  : TRUE,
      ds63  : TRUE,
      ds67  : TRUE
      ELSE    FALSE
    ENDCASES
  ELSIF (id = 20) THEN
    CASES s OF
      ds3  : TRUE,
      ds59  : TRUE
      ELSE    FALSE
    ENDCASES
  ELSIF (id = 21) THEN
    CASES s OF
      ds3  : TRUE,
      ds9  : TRUE,
      ds15  : TRUE,
      ds19  : TRUE,
      ds25  : TRUE,
      ds29  : TRUE,
      ds33  : TRUE,
      ds39  : TRUE,
      ds47  : TRUE,
      ds51  : TRUE,
      ds63  : TRUE,
      ds67  : TRUE
      ELSE    FALSE
    ENDCASES
  ELSIF (id = 22) THEN
    CASES s OF
      ds15  : TRUE,
      ds19  : TRUE,
      ds29  : TRUE,
      ds33  : TRUE,
      ds39  : TRUE,
      ds47  : TRUE,
      ds51  : TRUE,
      ds57  : TRUE,
      ds63  : TRUE,
      ds67  : TRUE
      ELSE    FALSE
    ENDCASES
  ELSIF (id = 23) THEN
    CASES s OF
      ds13  : TRUE,
      ds27  : TRUE,
      ds31  : TRUE,
      ds43  : TRUE,
      ds45  : TRUE,
      ds49  : TRUE,
      ds65  : TRUE
      ELSE    FALSE
    ENDCASES
  ELSIF (id = 24) THEN
    CASES s OF
      ds3  : TRUE,
      ds17  : TRUE,
      ds43  : TRUE,
      ds45  : TRUE,
      ds57  : TRUE,
      ds61  : TRUE,
      ds65  : TRUE
      ELSE    FALSE
    ENDCASES
```

```
ELSIF (id = 25) THEN
  CASES s OF
    ds3  : TRUE,
    ds9  : TRUE,
    ds15  : TRUE,
    ds19  : TRUE,
    ds25  : TRUE,
    ds27  : TRUE,
    ds29  : TRUE,
    ds33  : TRUE,
    ds41  : TRUE,
    ds45  : TRUE,
    ds47  : TRUE,
    ds49  : TRUE,
    ds51  : TRUE,
    ds59  : TRUE,
    ds63  : TRUE,
    ds65  : TRUE,
    ds67  : TRUE
    ELSE    FALSE
  ENDCASES
ELSIF (id = 26) THEN
  CASES s OF
    ds15  : TRUE,
    ds17  : TRUE,
    ds19  : TRUE,
    ds29  : TRUE,
    ds31  : TRUE,
    ds33  : TRUE,
    ds39  : TRUE,
    ds47  : TRUE,
    ds49  : TRUE,
    ds51  : TRUE,
    ds57  : TRUE,
    ds63  : TRUE,
    ds65  : TRUE,
    ds67  : TRUE
    ELSE    FALSE
  ENDCASES
ELSIF (id = 27) THEN
  CASES s OF
    ds9  : TRUE,
    ds13  : TRUE,
    ds15  : TRUE,
    ds19  : TRUE,
    ds27  : TRUE,
    ds29  : TRUE,
    ds33  : TRUE,
    ds41  : TRUE,
    ds45  : TRUE,
    ds47  : TRUE,
    ds51  : TRUE,
    ds57  : TRUE,
    ds61  : TRUE,
    ds63  : TRUE,
    ds67  : TRUE
    ELSE    FALSE
  ENDCASES
ELSIF (id = 28) THEN
  CASES s OF
    ds3  : TRUE,
    ds13  : TRUE,
    ds27  : TRUE,
    ds39  : TRUE,
    ds45  : TRUE,
    ds57  : TRUE,
    ds61  : TRUE
    ELSE    FALSE
  ENDCASES
ELSIF (id = 29) THEN
  CASES s OF
    ds3  : TRUE
    ELSE    FALSE
  ENDCASES
ELSIF (id = 30) THEN
  CASES s OF
    ds1  : TRUE
    ELSE    FALSE
  ENDCASES
ELSIF (id = 31) THEN
  CASES s OF
    ds3  : TRUE
    ELSE    FALSE
  ENDCASES
```

```
      ELSIF (id = 32) THEN
        CASES s OF
          ds1  : TRUE
          ELSE    FALSE
        ENDCASES
      ELSIF (id = 33) THEN
        CASES s OF
          ds3  : TRUE
          ELSE    FALSE
        ENDCASES
       ELSIF (id = 34) THEN
        CASES s OF
          ds35  : TRUE,
          ds53  : TRUE
          ELSE    FALSE
        ENDCASES
      ELSIF (id = 35) THEN
        CASES s OF
          ds21  : TRUE,
          ds53  : TRUE
          ELSE    FALSE
        ENDCASES
      ELSIF (id = 36) THEN
        CASES s OF
          ds41  : TRUE
          ELSE    FALSE
        ENDCASES
       ELSE
         FALSE
       ENDIF
  END tlc_control
```

Figure B.8: **Description of the Controller of the RTL Design**

```
% TLC RTL Component Axioms

 asserta_register_ax :
   AXIOM (FORALL  (input        : signal,
                   ld           : PRED[rtl_state],
                   output       : signal,
                   s1           : rtl_state,
                   s2           : rtl_state) :
   (asserta_register(input, ld, output) AND rtl_transition((# SOURCE := s1 , TARGET := s2 #))
         IMPLIES
   ((ld(s1) IMPLIES Val_d(output,s2) = Val_d(input,s1)) AND
   (NOT ld(s1) IMPLIES Val_d(output,s2) = Val_d(output,s1)))))

 asserta_constreg_ax :
   AXIOM (FORALL  (val          : value,
                   output       : signal,
                   s            : rtl_state) :
   (asserta_constreg(val,output)
         IMPLIES
   (Val_d(output,s) = val)))

 asserta_adder_ax :
   AXIOM (FORALL  (input1        : signal,
                   input2        : signal,
                   output        : signal,
                   s             : rtl_state) :
   asserta_adder(input1, input2, output)
        IMPLIES
   Val_d(output,s) = bb_plus(Val_d(input1,s), Val_d(input2,s)))

 asserta_subtractor_ax :
   AXIOM (FORALL  (input1        : signal,
                   input2        : signal,
                   output        : signal,
                   s             : rtl_state) :
   asserta_subtractor(input1, input2, output)
        IMPLIES
   Val_d(output,s) = bb_minus(Val_d(input1,s), Val_d(input2,s)))

 asserta_multiplier_ax :
   AXIOM (FORALL  (input1        : signal,
                   input2        : signal,
                   output        : signal,
                   s             : rtl_state) :
   asserta_multiplier(input1, input2, output)
        IMPLIES
   Val_d(output,s) = bb_mult(Val_d(input1,s), Val_d(input2,s)))

 asserta_divider_ax :
   AXIOM (FORALL  (input1        : signal,
                   input2        : signal,
                   output        : signal,
                   s             : rtl_state) :
   asserta_divider(input1, input2, output)
        IMPLIES
   Val_d(output,s) = bb_div(Val_d(input1,s), Val_d(input2,s)))

 asserta_eq_ax :
   AXIOM (FORALL  (input1        : signal,
                   input2        : signal,
                   output        : signal,
                   s             : rtl_state) :
   asserta_eq(input1, input2, output)
        IMPLIES
   Val_d(output,s) = bb_eq(Val_d(input1,s), Val_d(input2,s)))

 c_and_ax :
   AXIOM (FORALL  (input1        : signal,
                   input2        : signal,
                   output        : signal,
                   s             : rtl_state) :
   c_and(input1, input2, output)
        IMPLIES
   Val_d(output,s) = bb_and(Val_d(input1,s), Val_d(input2,s)))
```

```
c_or_ax :
  AXIOM (FORALL  (input1        : signal,
                  input2        : signal,
                  output        : signal,
                  s             : rtl_state) :
  c_or(input1, input2, output)
       IMPLIES
  Val_d(output,s) = bb_or(Val_d(input1,s), Val_d(input2,s)))

asserta_greatthan_ax :
  AXIOM (FORALL  (input1        : signal,
                  input2        : signal,
                  output        : signal,
                  s             : rtl_state) :
  asserta_greatthan(input1, input2, output)
       IMPLIES
  Val_d(output,s) = bb_gt(Val_d(input1,s), Val_d(input2,s)))

asserta_mux_2_1_ax :
  AXIOM (FORALL  (input1       : signal,
                  input2       : signal,
                  sel0         : PRED[rtl_state],
                  output       : signal,
                  s            : rtl_state) :
  asserta_mux_2_1(input1, input2, sel0, output)
        IMPLIES
  ((sel0(s) IMPLIES Val_d(output,s) = Val_d(input2,s)) AND
  (NOT sel0(s) IMPLIES Val_d(output,s) = Val_d(input1,s))))

asserta_mux_4_1_ax :
  AXIOM (FORALL  (input1       : signal,
                  input2       : signal,
                  input3       : signal,
                  input4       : signal,
                  sel0         : PRED[rtl_state],
                  sel1         : PRED[rtl_state],
                  output       : signal,
                  s            : rtl_state) :
  asserta_mux_4_1(input1, input2, input3, input4, sel0, sel1, output)
       IMPLIES
  (((sel0(s) AND sel1(s)) IMPLIES Val_d(output,s) = Val_d(input4,s)) AND
  ((NOT sel0(s) AND sel1(s)) IMPLIES Val_d(output,s) = Val_d(input3,s)) AND
  ((sel0(s) AND NOT sel1(s)) IMPLIES Val_d(output,s) = Val_d(input2,s)) AND
  ((NOT sel0(s) AND NOT sel1(s)) IMPLIES Val_d(output,s) = Val_d(input1,s))))

asserta_mux_8_1_ax :
  AXIOM (FORALL  (input1       : signal,
                  input2       : signal,
                  input3       : signal,
                  input4       : signal,
                  input5       : signal,
                  input6       : signal,
                  input7       : signal,
                  input8       : signal,
                  sel0         : PRED[rtl_state],
                  sel1         : PRED[rtl_state],
                  sel2         : PRED[rtl_state],
                  output       : signal,
                  s            : rtl_state) :
  asserta_mux_8_1(input1, input2, input3, input4, input5, input6, input7, input8, sel0, sel1, sel2,
                  output)
       IMPLIES
  (((sel0(s) AND sel1(s) AND sel2(s)) IMPLIES Val_d(output,s) = Val_d(input8,s)) AND
  ((NOT sel0(s) AND sel1(s) AND sel2(s)) IMPLIES Val_d(output,s) = Val_d(input7,s)) AND
  ((sel0(s) AND NOT sel1(s) AND sel2(s)) IMPLIES Val_d(output,s) = Val_d(input6,s)) AND
  ((NOT sel0(s) AND NOT sel1(s) AND sel2(s)) IMPLIES Val_d(output,s) = Val_d(input5,s)) AND
  ((sel0(s) AND sel1(s) AND NOT sel2(s)) IMPLIES Val_d(output,s) = Val_d(input4,s)) AND
  ((NOT sel0(s) AND sel1(s) AND NOT sel2(s)) IMPLIES Val_d(output,s) = Val_d(input3,s)) AND
  ((sel0(s) AND NOT sel1(s) AND NOT sel2(s)) IMPLIES Val_d(output,s) = Val_d(input2,s)) AND
  ((NOT sel0(s) AND NOT sel1(s) AND NOT sel2(s)) IMPLIES Val_d(output,s) = Val_d(input1,s))))

asserta_mux_16_1_ax :
  AXIOM (FORALL  (input1       : signal,
                  input2       : signal,
                  input3       : signal,
                  input4       : signal,
                  input5       : signal,
                 input6       : signal,
                 input7       : signal,
                 input8       : signal,
                 input9       : signal,
                 input10      : signal,
                 input11      : signal,
                 input12      : signal,
                 input13      : signal,
                 input14      : signal,
                 input15      : signal,
                 input16      : signal,
                 sel0         : PRED[rtl_state],
                 sel1         : PRED[rtl_state],
                 sel2         : PRED[rtl_state],
                 sel3         : PRED[rtl_state],
                 output       : signal,
                 s            : rtl_state) :
  asserta_mux_16_1(input1, input2, input3, input4, input5, input6, input7, input8, input9, input10,
                  input11, input12, input13, input14, input15, input16, sel0, sel1, sel2, sel3,
                  output)
       IMPLIES
  (((sel0(s) AND sel1(s) AND sel2(s) AND sel3(s)) IMPLIES
       Val_d(output,s) = Val_d(input16,s)) AND
  ((NOT sel0(s) AND sel1(s) AND sel2(s) AND sel3(s)) IMPLIES
       Val_d(output,s) = Val_d(input15,s)) AND
  ((sel0(s) AND NOT sel1(s) AND sel2(s) AND sel3(s)) IMPLIES
       Val_d(output,s) = Val_d(input14,s)) AND
  ((NOT sel0(s) AND NOT sel1(s) AND sel2(s) AND sel3(s)) IMPLIES
       Val_d(output,s) = Val_d(input13,s)) AND
  ((sel0(s) AND sel1(s) AND NOT sel2(s) AND sel3(s)) IMPLIES
       Val_d(output,s) = Val_d(input12,s)) AND
  ((NOT sel0(s) AND sel1(s) AND NOT sel2(s) AND sel3(s)) IMPLIES
       Val_d(output,s) = Val_d(input11,s)) AND
  ((sel0(s) AND NOT sel1(s) AND NOT sel2(s) AND sel3(s)) IMPLIES
       Val_d(output,s) = Val_d(input10,s)) AND
  ((NOT sel0(s) AND NOT sel1(s) AND NOT sel2(s) AND sel3(s)) IMPLIES
       Val_d(output,s) = Val_d(input9,s)) AND
  ((sel0(s) AND sel1(s) AND sel2(s) AND NOT sel3(s)) IMPLIES
       Val_d(output,s) = Val_d(input8,s)) AND
  ((NOT sel0(s) AND sel1(s) AND sel2(s) AND NOT sel3(s)) IMPLIES
       Val_d(output,s) = Val_d(input7,s)) AND
  ((sel0(s) AND NOT sel1(s) AND sel2(s) AND NOT sel3(s)) IMPLIES
       Val_d(output,s) = Val_d(input6,s)) AND
  ((NOT sel0(s) AND NOT sel1(s) AND sel2(s) AND NOT sel3(s)) IMPLIES
       Val_d(output,s) = Val_d(input5,s)) AND
  ((sel0(s) AND sel1(s) AND NOT sel2(s) AND NOT sel3(s)) IMPLIES
       Val_d(output,s) = Val_d(input4,s)) AND
  ((NOT sel0(s) AND sel1(s) AND NOT sel2(s) AND NOT sel3(s)) IMPLIES
       Val_d(output,s) = Val_d(input3,s)) AND
  ((sel0(s) AND NOT sel1(s) AND NOT sel2(s) AND NOT sel3(s)) IMPLIES
       Val_d(output,s) = Val_d(input2,s)) AND
  ((NOT sel0(s) AND NOT sel1(s) AND NOT sel2(s) AND NOT sel3(s)) IMPLIES
       Val_d(output,s) = Val_d(input1,s))))

asserta_mux_32_1_ax :
  AXIOM (FORALL  (input1       : signal,
                 input2       : signal,
                 input3       : signal,
                 input4       : signal,
                 input5       : signal,
                 input6       : signal,
                 input7       : signal,
                 input8       : signal,
                 input9       : signal,
                 input10      : signal,
                 input11      : signal,
                 input12      : signal,
                 input13      : signal,
                 input14      : signal,
                 input15      : signal,
                 input16      : signal,
                 input17      : signal,
                 input18      : signal,
                 input19      : signal,
                 input20      : signal,
                 input21      : signal,
                 input22      : signal,
                 input22       : signal,
                 input23       : signal,
                 input24       : signal,
                 input25       : signal,
                 input26       : signal,
                 input27       : signal,
                 input28       : signal,
                 input29       : signal,
                 input30       : signal,
                 input31       : signal,
                 input32       : signal,
                 sel0          : PRED[rtl_state],
                 sel1          : PRED[rtl_state],
                 sel2          : PRED[rtl_state],
                 sel3          : PRED[rtl_state],
                 sel4          : PRED[rtl_state],
                 output        : signal,
                 s             : rtl_state) :
asserta_mux_32_1(input1, input2, input3, input4, input5, input6, input7, input8, input9, input10,
                 input11, input12, input13, input14, input15, input16, input17, input18, input19,
                 input20, input21, input22, input23, input24, input25, input26, input27, input28,
                 input29, input30, input31, input32, sel0, sel1, sel2, sel3, sel4, output)
    IMPLIES
(((sel0(s) AND sel1(s) AND sel2(s) AND sel3(s) AND sel4(s)) IMPLIES
    Val_d(output,s) = Val_d(input16,s)) AND
((NOT sel0(s) AND sel1(s) AND sel2(s) AND sel3(s) AND sel4(s)) IMPLIES
    Val_d(output,s) = Val_d(input15,s)) AND
((sel0(s) AND NOT sel1(s) AND sel2(s) AND sel3(s) AND sel4(s)) IMPLIES
    Val_d(output,s) = Val_d(input14,s)) AND
((NOT sel0(s) AND NOT sel1(s) AND sel2(s) AND sel3(s) AND sel4(s)) IMPLIES
    Val_d(output,s) = Val_d(input13,s)) AND
((sel0(s) AND sel1(s) AND NOT sel2(s) AND sel3(s) AND sel4(s)) IMPLIES
    Val_d(output,s) = Val_d(input12,s)) AND
((NOT sel0(s) AND sel1(s) AND NOT sel2(s) AND sel3(s) AND sel4(s)) IMPLIES
    Val_d(output,s) = Val_d(input11,s)) AND
((sel0(s) AND NOT sel1(s) AND NOT sel2(s) AND sel3(s) AND sel4(s)) IMPLIES
    Val_d(output,s) = Val_d(input10,s)) AND
((NOT sel0(s) AND NOT sel1(s) AND NOT sel2(s) AND sel3(s) AND sel4(s)) IMPLIES
    Val_d(output,s) = Val_d(input9,s)) AND
((sel0(s) AND sel1(s) AND sel2(s) AND NOT sel3(s) AND sel4(s))
    IMPLIES Val_d(output,s) = Val_d(input8,s)) AND
((NOT sel0(s) AND sel1(s) AND sel2(s) AND NOT sel3(s) AND sel4(s))
    IMPLIES Val_d(output,s) = Val_d(input7,s)) AND
((sel0(s) AND NOT sel1(s) AND sel2(s) AND NOT sel3(s) AND sel4(s))
    IMPLIES Val_d(output,s) = Val_d(input6,s)) AND
((NOT sel0(s) AND NOT sel1(s) AND sel2(s) AND NOT sel3(s) AND sel4(s))
    IMPLIES Val_d(output,s) = Val_d(input5,s)) AND
((sel0(s) AND sel1(s) AND NOT sel2(s) AND NOT sel3(s) AND sel4(s))
    IMPLIES Val_d(output,s) = Val_d(input4,s)) AND
((NOT sel0(s) AND sel1(s) AND NOT sel2(s) AND NOT sel3(s) AND sel4(s))
    IMPLIES Val_d(output,s) = Val_d(input3,s)) AND
((sel0(s) AND NOT sel1(s) AND NOT sel2(s) AND NOT sel3(s) AND sel4(s))
    IMPLIES Val_d(output,s) = Val_d(input2,s)) AND
((NOT sel0(s) AND NOT sel1(s) AND NOT sel2(s) AND NOT sel3(s) AND sel4(s))
    IMPLIES Val_d(output,s) = Val_d(input1,s)) AND
((sel0(s) AND sel1(s) AND sel2(s) AND sel3(s) AND NOT sel4(s))
    IMPLIES Val_d(output,s) = Val_d(input16,s)) AND
((NOT sel0(s) AND sel1(s) AND sel2(s) AND sel3(s) AND NOT sel4(s))
    IMPLIES Val_d(output,s) = Val_d(input15,s)) AND
((sel0(s) AND NOT sel1(s) AND sel2(s) AND sel3(s) AND NOT sel4(s))
    IMPLIES Val_d(output,s) = Val_d(input14,s)) AND
((NOT sel0(s) AND NOT sel1(s) AND sel2(s) AND sel3(s) AND NOT sel4(s))
    IMPLIES Val_d(output,s) = Val_d(input13,s)) AND
((sel0(s) AND sel1(s) AND NOT sel2(s) AND sel3(s) AND NOT sel4(s))
    IMPLIES Val_d(output,s) = Val_d(input12,s)) AND
((NOT sel0(s) AND sel1(s) AND NOT sel2(s) AND sel3(s) AND NOT sel4(s))
    IMPLIES Val_d(output,s) = Val_d(input11,s)) AND
((sel0(s) AND NOT sel1(s) AND NOT sel2(s) AND sel3(s) AND NOT sel4(s))
    IMPLIES Val_d(output,s) = Val_d(input10,s)) AND
((NOT sel0(s) AND NOT sel1(s) AND NOT sel2(s) AND sel3(s) AND NOT sel4(s))
    IMPLIES Val_d(output,s) = Val_d(input9,s)) AND
((sel0(s) AND sel1(s) AND sel2(s) AND NOT sel3(s) AND NOT sel4(s))
    IMPLIES Val_d(output,s) = Val_d(input8,s)) AND
((NOT sel0(s) AND sel1(s) AND sel2(s) AND NOT sel3(s) AND NOT sel4(s))
    IMPLIES Val_d(output,s) = Val_d(input7,s)) AND
((sel0(s) AND NOT sel1(s) AND sel2(s) AND NOT sel3(s) AND NOT sel4(s))
    IMPLIES Val_d(output,s) = Val_d(input6,s)) AND
```

```
((NOT sel0(s) AND NOT sel1(s) AND sel2(s) AND NOT sel3(s) AND NOT sel4(s))
     IMPLIES Val_d(output,s) = Val_d(input5,s)) AND
((sel0(s) AND sel1(s) AND NOT sel2(s) AND NOT sel3(s) AND NOT sel4(s))
     IMPLIES Val_d(output,s) = Val_d(input4,s)) AND
((NOT sel0(s) AND sel1(s) AND NOT sel2(s) AND NOT sel3(s) AND NOT sel4(s))
     IMPLIES Val_d(output,s) = Val_d(input3,s)) AND
((sel0(s) AND NOT sel1(s) AND NOT sel2(s) AND NOT sel3(s) AND NOT sel4(s))
     IMPLIES Val_d(output,s) = Val_d(input2,s)) AND
((NOT sel0(s) AND NOT sel1(s) AND NOT sel2(s) AND NOT sel3(s) AND NOT sel4(s))
     IMPLIES Val_d(output,s) = Val_d(input1,s))))
```

Figure B.9: **Behavior Description of the Components used in Synthesis of TLC**

```
% TLC Lemmas
eq_cp0_st_lemma : LEMMA (equivalent_states(First_b(bcp0),First_d(B_p(bcp0))) AND
                          beh_transition_condition(bcp0) AND
                          rtl_transition_condition(B_p(bcp0)))
                                 IMPLIES
                          Val_b(st,Last_b(bcp0)) = Val_d(asserta_register_5_out1,Last_d(B_p(bcp0)))

eq_cp0_hl_lemma : LEMMA (equivalent_states(First_b(bcp0),First_d(B_p(bcp0))) AND
                          beh_transition_condition(bcp0) AND
                          rtl_transition_condition(B_p(bcp0)))
                                 IMPLIES
                          Val_b(hl,Last_b(bcp0)) = Val_d(asserta_register_6_out1,Last_d(B_p(bcp0)))

eq_cp0_fl_lemma : LEMMA (equivalent_states(First_b(bcp0),First_d(B_p(bcp0))) AND
                          beh_transition_condition(bcp0) AND
                          rtl_transition_condition(B_p(bcp0)))
                                 IMPLIES
                          Val_b(fl,Last_b(bcp0)) = Val_d(asserta_register_7_out1,Last_d(B_p(bcp0)))

eq_cp0_cof_lemma : LEMMA (equivalent_states(First_b(bcp0),First_d(B_p(bcp0))) AND
                          beh_transition_condition(bcp0) AND
                          rtl_transition_condition(B_p(bcp0)))
                                 IMPLIES
                          Val_b(cof,Last_b(bcp0)) = Val_d(asserta_register_2_out1,Last_d(B_p(bcp0)))

eq_cp0_tol_lemma : LEMMA (equivalent_states(First_b(bcp0),First_d(B_p(bcp0))) AND
                          beh_transition_condition(bcp0) AND
                          rtl_transition_condition(B_p(bcp0)))
                                 IMPLIES
                          Val_b(tol,Last_b(bcp0)) = Val_d(asserta_register_3_out1,Last_d(B_p(bcp0)))

eq_cp0_tos_lemma : LEMMA (equivalent_states(First_b(bcp0),First_d(B_p(bcp0))) AND
                          beh_transition_condition(bcp0) AND
                          rtl_transition_condition(B_p(bcp0)))
                                 IMPLIES
                          Val_b(tos,Last_b(bcp0)) = Val_d(asserta_register_4_out1,Last_d(B_p(bcp0)))

eq_cp0_t18_lemma : LEMMA (equivalent_states(First_b(bcp0),First_d(B_p(bcp0))) AND
                          beh_transition_condition(bcp0) AND
                          rtl_transition_condition(B_p(bcp0)))
                                 IMPLIES
                          Val_b(t18,Last_b(bcp0)) = Val_d(asserta_register_8_out1,Last_d(B_p(bcp0)))

eq_cp1_hl_lemma : LEMMA (equivalent_states(First_b(bcp1),First_d(B_p(bcp1))) AND
                          beh_transition_condition(bcp1) AND
                          rtl_transition_condition(B_p(bcp1)))
                                 IMPLIES
                          Val_b(hl,Last_b(bcp1)) = Val_d(asserta_register_1_out1,Last_d(B_p(bcp1)))

eq_cp1_fl_lemma : LEMMA (equivalent_states(First_b(bcp1),First_d(B_p(bcp1))) AND
                          beh_transition_condition(bcp1) AND
                          rtl_transition_condition(B_p(bcp1)))
                                 IMPLIES
                          Val_b(fl,Last_b(bcp1)) = Val_d(asserta_register_2_out1,Last_d(B_p(bcp1)))

eq_cp1_t24_lemma : LEMMA (equivalent_states(First_b(bcp1),First_d(B_p(bcp1))) AND
                          beh_transition_condition(bcp1) AND
                          rtl_transition_condition(B_p(bcp1)))
                                 IMPLIES
                          Val_b(t24,Last_b(bcp1)) = Val_d(asserta_register_4_out1,Last_d(B_p(bcp1)))

eq_cp1_t25_lemma : LEMMA (equivalent_states(First_b(bcp1),First_d(B_p(bcp1))) AND
                          beh_transition_condition(bcp1) AND
                          rtl_transition_condition(B_p(bcp1)))
                                 IMPLIES
                          Val_b(t25,Last_b(bcp1)) = Val_d(asserta_register_3_out1,Last_d(B_p(bcp1)))

eq_cp2_state_lemma : LEMMA (equivalent_states(First_b(bcp2),First_d(B_p(bcp2))) AND
                          beh_transition_condition(bcp2) AND
                          rtl_transition_condition(B_p(bcp2)))
                                 IMPLIES
                          Val_b(state,Last_b(bcp2)) = Val_d(asserta_register_1_out1,Last_d(B_p(bcp2)))

eq_cp2_st_lemma : LEMMA (equivalent_states(First_b(bcp2),First_d(B_p(bcp2))) AND
                          beh_transition_condition(bcp2) AND
                          rtl_transition_condition(B_p(bcp2)))
                                 IMPLIES
                          Val_b(st,Last_b(bcp2)) = Val_d(asserta_register_2_out1,Last_d(B_p(bcp2)))
```

```
eq_cp2_hl_lemma : LEMMA (equivalent_states(First_b(bcp2),First_d(B_p(bcp2))) AND
                         beh_transition_condition(bcp2) AND
                         rtl_transition_condition(B_p(bcp2)))
                                IMPLIES
                         Val_b(hl,Last_b(bcp2)) = Val_d(asserta_register_3_out1,Last_d(B_p(bcp2)))

eq_cp2_fl_lemma : LEMMA (equivalent_states(First_b(bcp2),First_d(B_p(bcp2))) AND
                         beh_transition_condition(bcp2) AND
                         rtl_transition_condition(B_p(bcp2)))
                                IMPLIES
                         Val_b(fl,Last_b(bcp2)) = Val_d(asserta_register_4_out1,Last_d(B_p(bcp2)))

eq_cp2_cof_lemma : LEMMA (equivalent_states(First_b(bcp2),First_d(B_p(bcp2))) AND
                         beh_transition_condition(bcp2) AND
                         rtl_transition_condition(B_p(bcp2)))
                                IMPLIES
                         Val_b(cof,Last_b(bcp2)) = Val_d(asserta_register_5_out1,Last_d(B_p(bcp2)))

eq_cp2_tol_lemma : LEMMA (equivalent_states(First_b(bcp2),First_d(B_p(bcp2))) AND
                         beh_transition_condition(bcp2) AND
                         rtl_transition_condition(B_p(bcp2)))
                                IMPLIES
                         Val_b(tol,Last_b(bcp2)) = Val_d(asserta_register_6_out1,Last_d(B_p(bcp2)))

eq_cp2_tos_lemma : LEMMA (equivalent_states(First_b(bcp2),First_d(B_p(bcp2))) AND
                         beh_transition_condition(bcp2) AND
                         rtl_transition_condition(B_p(bcp2)))
                                IMPLIES
                         Val_b(tos,Last_b(bcp2)) = Val_d(asserta_register_7_out1,Last_d(B_p(bcp2)))

eq_cp3_st_lemma : LEMMA (equivalent_states(First_b(bcp3),First_d(B_p(bcp3))) AND
                         beh_transition_condition(bcp3) AND
                         rtl_transition_condition(B_p(bcp3)))
                                IMPLIES
                         Val_b(st,Last_b(bcp3)) = Val_d(asserta_register_2_out1,Last_d(B_p(bcp3)))

eq_cp3_hl_lemma : LEMMA (equivalent_states(First_b(bcp3),First_d(B_p(bcp3))) AND
                         beh_transition_condition(bcp3) AND
                         rtl_transition_condition(B_p(bcp3)))
                                IMPLIES
                         Val_b(hl,Last_b(bcp3)) = Val_d(asserta_register_3_out1,Last_d(B_p(bcp3)))

eq_cp3_fl_lemma : LEMMA (equivalent_states(First_b(bcp3),First_d(B_p(bcp3))) AND
                         beh_transition_condition(bcp3) AND
                         rtl_transition_condition(B_p(bcp3)))
                                IMPLIES
                         Val_b(fl,Last_b(bcp3)) = Val_d(asserta_register_4_out1,Last_d(B_p(bcp3)))

eq_cp3_cof_lemma : LEMMA (equivalent_states(First_b(bcp3),First_d(B_p(bcp3))) AND
                         beh_transition_condition(bcp3) AND
                         rtl_transition_condition(B_p(bcp3)))
                                IMPLIES
                         Val_b(cof,Last_b(bcp3)) = Val_d(asserta_register_5_out1,Last_d(B_p(bcp3)))

eq_cp3_tol_lemma : LEMMA (equivalent_states(First_b(bcp3),First_d(B_p(bcp3))) AND
                         beh_transition_condition(bcp3) AND
                         rtl_transition_condition(B_p(bcp3)))
                                IMPLIES
                         Val_b(tol,Last_b(bcp3)) = Val_d(asserta_register_6_out1,Last_d(B_p(bcp3)))

eq_cp3_tos_lemma : LEMMA (equivalent_states(First_b(bcp3),First_d(B_p(bcp3))) AND
                         beh_transition_condition(bcp3) AND
                         rtl_transition_condition(B_p(bcp3)))
                                IMPLIES
                         Val_b(tos,Last_b(bcp3)) = Val_d(asserta_register_7_out1,Last_d(B_p(bcp3)))

eq_cp4_t28_lemma : LEMMA (equivalent_states(First_b(bcp4),First_d(B_p(bcp4))) AND
                         beh_transition_condition(bcp4) AND
                         rtl_transition_condition(B_p(bcp4)))
                                IMPLIES
                         Val_b(t28,Last_b(bcp4)) = Val_d(asserta_register_8_out1,Last_d(B_p(bcp4)))

eq_cp5_hl_lemma : LEMMA (equivalent_states(First_b(bcp5),First_d(B_p(bcp5))) AND
                         beh_transition_condition(bcp5) AND
                         rtl_transition_condition(B_p(bcp5)))
                                IMPLIES
                         Val_b(hl,Last_b(bcp5)) = Val_d(asserta_register_1_out1,Last_d(B_p(bcp5)))
```

```
eq_cp5_fl_lemma : LEMMA (equivalent_states(First_b(bcp5),First_d(B_p(bcp5))) AND
                         beh_transition_condition(bcp5) AND
                         rtl_transition_condition(B_p(bcp5)))
                                IMPLIES
                         Val_b(fl,Last_b(bcp5)) = Val_d(asserta_register_2_out1,Last_d(B_p(bcp5)))

eq_cp5_t30_lemma : LEMMA (equivalent_states(First_b(bcp5),First_d(B_p(bcp5))) AND
                         beh_transition_condition(bcp5) AND
                         rtl_transition_condition(B_p(bcp5)))
                                IMPLIES
                         Val_b(t30,Last_b(bcp5)) = Val_d(asserta_register_3_out1,Last_d(B_p(bcp5)))

eq_cp6_state_lemma : LEMMA (equivalent_states(First_b(bcp6),First_d(B_p(bcp6))) AND
                         beh_transition_condition(bcp6) AND
                         rtl_transition_condition(B_p(bcp6)))
                                IMPLIES
                         Val_b(state,Last_b(bcp6)) = Val_d(asserta_register_1_out1,Last_d(B_p(bcp6)))

eq_cp6_st_lemma : LEMMA (equivalent_states(First_b(bcp6),First_d(B_p(bcp6))) AND
                         beh_transition_condition(bcp6) AND
                         rtl_transition_condition(B_p(bcp6)))
                                IMPLIES
                         Val_b(st,Last_b(bcp6)) = Val_d(asserta_register_2_out1,Last_d(B_p(bcp6)))

eq_cp6_hl_lemma : LEMMA (equivalent_states(First_b(bcp6),First_d(B_p(bcp6))) AND
                         beh_transition_condition(bcp6) AND
                         rtl_transition_condition(B_p(bcp6)))
                                IMPLIES
                         Val_b(hl,Last_b(bcp6)) = Val_d(asserta_register_3_out1,Last_d(B_p(bcp6)))

eq_cp6_fl_lemma : LEMMA (equivalent_states(First_b(bcp6),First_d(B_p(bcp6))) AND
                         beh_transition_condition(bcp6) AND
                         rtl_transition_condition(B_p(bcp6)))
                                IMPLIES
                         Val_b(fl,Last_b(bcp6)) = Val_d(asserta_register_4_out1,Last_d(B_p(bcp6)))

eq_cp6_cof_lemma : LEMMA (equivalent_states(First_b(bcp6),First_d(B_p(bcp6))) AND
                         beh_transition_condition(bcp6) AND
                         rtl_transition_condition(B_p(bcp6)))
                                IMPLIES
                         Val_b(cof,Last_b(bcp6)) = Val_d(asserta_register_5_out1,Last_d(B_p(bcp6)))

eq_cp6_tol_lemma : LEMMA (equivalent_states(First_b(bcp6),First_d(B_p(bcp6))) AND
                         beh_transition_condition(bcp6) AND
                         rtl_transition_condition(B_p(bcp6)))
                                IMPLIES
                         Val_b(tol,Last_b(bcp6)) = Val_d(asserta_register_6_out1,Last_d(B_p(bcp6)))

eq_cp6_tos_lemma : LEMMA (equivalent_states(First_b(bcp6),First_d(B_p(bcp6))) AND
                         beh_transition_condition(bcp6) AND
                         rtl_transition_condition(B_p(bcp6)))
                                IMPLIES
                         Val_b(tos,Last_b(bcp6)) = Val_d(asserta_register_7_out1,Last_d(B_p(bcp6)))

eq_cp7_state_lemma : LEMMA (equivalent_states(First_b(bcp7),First_d(B_p(bcp7))) AND
                         beh_transition_condition(bcp7) AND
                         rtl_transition_condition(B_p(bcp7)))
                                IMPLIES
                         Val_b(state,Last_b(bcp7)) = Val_d(asserta_register_1_out1,Last_d(B_p(bcp7)))

eq_cp7_st_lemma : LEMMA (equivalent_states(First_b(bcp7),First_d(B_p(bcp7))) AND
                         beh_transition_condition(bcp7) AND
                         rtl_transition_condition(B_p(bcp7)))
                                IMPLIES
                         Val_b(st,Last_b(bcp7)) = Val_d(asserta_register_2_out1,Last_d(B_p(bcp7)))

eq_cp7_hl_lemma : LEMMA (equivalent_states(First_b(bcp7),First_d(B_p(bcp7))) AND
                         beh_transition_condition(bcp7) AND
                         rtl_transition_condition(B_p(bcp7)))
                                IMPLIES
                         Val_b(hl,Last_b(bcp7)) = Val_d(asserta_register_3_out1,Last_d(B_p(bcp7)))

eq_cp7_fl_lemma : LEMMA (equivalent_states(First_b(bcp7),First_d(B_p(bcp7))) AND
                         beh_transition_condition(bcp7) AND
                         rtl_transition_condition(B_p(bcp7)))
                                IMPLIES
                         Val_b(fl,Last_b(bcp7)) = Val_d(asserta_register_4_out1,Last_d(B_p(bcp7)))
```

```
eq_cp7_cof_lemma : LEMMA (equivalent_states(First_b(bcp7),First_d(B_p(bcp7))) AND
                          beh_transition_condition(bcp7) AND
                          rtl_transition_condition(B_p(bcp7)))
                                IMPLIES
                          Val_b(cof,Last_b(bcp7)) = Val_d(asserta_register_5_out1,Last_d(B_p(bcp7)))

eq_cp7_tol_lemma : LEMMA (equivalent_states(First_b(bcp7),First_d(B_p(bcp7))) AND
                          beh_transition_condition(bcp7) AND
                          rtl_transition_condition(B_p(bcp7)))
                                IMPLIES
                          Val_b(tol,Last_b(bcp7)) = Val_d(asserta_register_6_out1,Last_d(B_p(bcp7)))

eq_cp7_tos_lemma : LEMMA (equivalent_states(First_b(bcp7),First_d(B_p(bcp7))) AND
                          beh_transition_condition(bcp7) AND
                          rtl_transition_condition(B_p(bcp7)))
                                 IMPLIES
                          Val_b(tos,Last_b(bcp7)) = Val_d(asserta_register_7_out1,Last_d(B_p(bcp7)))

 eq_cp8_t31_lemma : LEMMA (equivalent_states(First_b(bcp8),First_d(B_p(bcp8))) AND
                           beh_transition_condition(bcp8) AND
                           rtl_transition_condition(B_p(bcp8)))
                                 IMPLIES
                           Val_b(t31,Last_b(bcp8)) = Val_d(asserta_register_8_out1,Last_d(B_p(bcp8)))

 eq_cp9_hl_lemma : LEMMA (equivalent_states(First_b(bcp9),First_d(B_p(bcp9))) AND
                           beh_transition_condition(bcp9) AND
                           rtl_transition_condition(B_p(bcp9)))
                                 IMPLIES
                           Val_b(hl,Last_b(bcp9)) = Val_d(asserta_register_1_out1,Last_d(B_p(bcp9)))

 eq_cp9_fl_lemma : LEMMA (equivalent_states(First_b(bcp9),First_d(B_p(bcp9))) AND
                           beh_transition_condition(bcp9) AND
                           rtl_transition_condition(B_p(bcp9)))
                                 IMPLIES
                           Val_b(fl,Last_b(bcp9)) = Val_d(asserta_register_2_out1,Last_d(B_p(bcp9)))

 eq_cp9_t33_lemma : LEMMA (equivalent_states(First_b(bcp9),First_d(B_p(bcp9))) AND
                           beh_transition_condition(bcp9) AND
                           rtl_transition_condition(B_p(bcp9)))
                                 IMPLIES
                           Val_b(t33,Last_b(bcp9)) = Val_d(asserta_register_4_out1,Last_d(B_p(bcp9)))

 eq_cp9_t34_lemma : LEMMA (equivalent_states(First_b(bcp9),First_d(B_p(bcp9))) AND
                           beh_transition_condition(bcp9) AND
                           rtl_transition_condition(B_p(bcp9)))
                                 IMPLIES
                           Val_b(t34,Last_b(bcp9)) = Val_d(asserta_register_3_out1,Last_d(B_p(bcp9)))

 eq_cp10_state_lemma : LEMMA (equivalent_states(First_b(bcp10),First_d(B_p(bcp10))) AND
                           beh_transition_condition(bcp10) AND
                           rtl_transition_condition(B_p(bcp10)))
                                 IMPLIES
                           Val_b(state,Last_b(bcp10)) =Val_d(asserta_register_1_out1,Last_d(B_p(bcp10)))

 eq_cp10_st_lemma : LEMMA (equivalent_states(First_b(bcp10),First_d(B_p(bcp10))) AND
                           beh_transition_condition(bcp10) AND
                           rtl_transition_condition(B_p(bcp10)))
                                 IMPLIES
                           Val_b(st,Last_b(bcp10)) = Val_d(asserta_register_2_out1,Last_d(B_p(bcp10)))

 eq_cp10_hl_lemma : LEMMA (equivalent_states(First_b(bcp10),First_d(B_p(bcp10))) AND
                           beh_transition_condition(bcp10) AND
                           rtl_transition_condition(B_p(bcp10)))
                                 IMPLIES
                           Val_b(hl,Last_b(bcp10)) = Val_d(asserta_register_3_out1,Last_d(B_p(bcp10)))

 eq_cp10_fl_lemma : LEMMA (equivalent_states(First_b(bcp10),First_d(B_p(bcp10))) AND
                           beh_transition_condition(bcp10) AND
                           rtl_transition_condition(B_p(bcp10)))
                                 IMPLIES
                           Val_b(fl,Last_b(bcp10)) = Val_d(asserta_register_4_out1,Last_d(B_p(bcp10)))

 eq_cp10_cof_lemma : LEMMA (equivalent_states(First_b(bcp10),First_d(B_p(bcp10))) AND
                           beh_transition_condition(bcp10) AND
                           rtl_transition_condition(B_p(bcp10)))
                                 IMPLIES
                           Val_b(cof,Last_b(bcp10)) = Val_d(asserta_register_5_out1,Last_d(B_p(bcp10)))
```

```
  eq_cp10_tol_lemma : LEMMA (equivalent_states(First_b(bcp10),First_d(B_p(bcp10))) AND
                             beh_transition_condition(bcp10) AND
                             rtl_transition_condition(B_p(bcp10)))
                                    IMPLIES
                             Val_b(tol,Last_b(bcp10)) = Val_d(asserta_register_6_out1,Last_d(B_p(bcp10)))

  eq_cp10_tos_lemma : LEMMA (equivalent_states(First_b(bcp10),First_d(B_p(bcp10))) AND
                             beh_transition_condition(bcp10) AND
                             rtl_transition_condition(B_p(bcp10)))
                                    IMPLIES
                             Val_b(tos,Last_b(bcp10)) = Val_d(asserta_register_7_out1,Last_d(B_p(bcp10)))

  eq_cp11_state_lemma : LEMMA (equivalent_states(First_b(bcp11),First_d(B_p(bcp11))) AND
                             beh_transition_condition(bcp11) AND
                             rtl_transition_condition(B_p(bcp11)))
                                    IMPLIES
                             Val_b(state,Last_b(bcp11)) =Val_d(asserta_register_1_out1,Last_d(B_p(bcp11)))

  eq_cp11_st_lemma : LEMMA (equivalent_states(First_b(bcp11),First_d(B_p(bcp11))) AND
                             beh_transition_condition(bcp11) AND
                             rtl_transition_condition(B_p(bcp11)))
                                    IMPLIES
                             Val_b(st,Last_b(bcp11)) = Val_d(asserta_register_2_out1,Last_d(B_p(bcp11)))

  eq_cp11_hl_lemma : LEMMA (equivalent_states(First_b(bcp11),First_d(B_p(bcp11))) AND
                             beh_transition_condition(bcp11) AND
                             rtl_transition_condition(B_p(bcp11)))
                                    IMPLIES
                             Val_b(hl,Last_b(bcp11)) = Val_d(asserta_register_3_out1,Last_d(B_p(bcp11)))

 eq_cp11_fl_lemma : LEMMA (equivalent_states(First_b(bcp11),First_d(B_p(bcp11))) AND
                             beh_transition_condition(bcp11) AND
                             rtl_transition_condition(B_p(bcp11)))
                                    IMPLIES
                             Val_b(fl,Last_b(bcp11)) = Val_d(asserta_register_4_out1,Last_d(B_p(bcp11)))

  eq_cp11_cof_lemma : LEMMA (equivalent_states(First_b(bcp11),First_d(B_p(bcp11))) AND
                             beh_transition_condition(bcp11) AND
                             rtl_transition_condition(B_p(bcp11)))
                                    IMPLIES
                             Val_b(cof,Last_b(bcp11)) = Val_d(asserta_register_5_out1,Last_d(B_p(bcp11)))

  eq_cp11_tol_lemma : LEMMA (equivalent_states(First_b(bcp11),First_d(B_p(bcp11))) AND
                             beh_transition_condition(bcp11) AND
                             rtl_transition_condition(B_p(bcp11)))
                                    IMPLIES
                             Val_b(tol,Last_b(bcp11)) = Val_d(asserta_register_6_out1,Last_d(B_p(bcp11)))

  eq_cp11_tos_lemma : LEMMA (equivalent_states(First_b(bcp11),First_d(B_p(bcp11))) AND
                             beh_transition_condition(bcp11) AND
                             rtl_transition_condition(B_p(bcp11)))
                                    IMPLIES
                             Val_b(tos,Last_b(bcp11)) = Val_d(asserta_register_7_out1,Last_d(B_p(bcp11)))

  eq_cp12_st_lemma : LEMMA (equivalent_states(First_b(bcp12),First_d(B_p(bcp12))) AND
                             beh_transition_condition(bcp12) AND
                             rtl_transition_condition(B_p(bcp12)))
                                    IMPLIES
                             Val_b(st,Last_b(bcp12)) = Val_d(asserta_register_3_out1,Last_d(B_p(bcp12)))

  eq_cp12_hl_lemma : LEMMA (equivalent_states(First_b(bcp12),First_d(B_p(bcp12))) AND
                             beh_transition_condition(bcp12) AND
                             rtl_transition_condition(B_p(bcp12)))
                                    IMPLIES
                             Val_b(hl,Last_b(bcp12)) = Val_d(asserta_register_4_out1,Last_d(B_p(bcp12)))

  eq_cp12_fl_lemma : LEMMA (equivalent_states(First_b(bcp12),First_d(B_p(bcp12))) AND
                             beh_transition_condition(bcp12) AND
                             rtl_transition_condition(B_p(bcp12)))
                                    IMPLIES
                             Val_b(fl,Last_b(bcp12)) = Val_d(asserta_register_5_out1,Last_d(B_p(bcp12)))

  eq_cp12_tos_lemma : LEMMA (equivalent_states(First_b(bcp12),First_d(B_p(bcp12))) AND
                             beh_transition_condition(bcp12) AND
                             rtl_transition_condition(B_p(bcp12)))
                                    IMPLIES
                             Val_b(tos,Last_b(bcp12)) = Val_d(asserta_register_2_out1,Last_d(B_p(bcp12)))
```

```
eq_cp12_t35_lemma : LEMMA (equivalent_states(First_b(bcp12),First_d(B_p(bcp12))) AND
                          beh_transition_condition(bcp12) AND
                          rtl_transition_condition(B_p(bcp12)))
                                 IMPLIES
                          Val_b(t35,Last_b(bcp12)) = Val_d(asserta_register_6_out1,Last_d(B_p(bcp12)))

eq_cp13_hl_lemma : LEMMA (equivalent_states(First_b(bcp13),First_d(B_p(bcp13))) AND
                          beh_transition_condition(bcp13) AND
                          rtl_transition_condition(B_p(bcp13)))
                                 IMPLIES
                          Val_b(hl,Last_b(bcp13)) = Val_d(asserta_register_1_out1,Last_d(B_p(bcp13)))

eq_cp13_fl_lemma : LEMMA (equivalent_states(First_b(bcp13),First_d(B_p(bcp13))) AND
                          beh_transition_condition(bcp13) AND
                          rtl_transition_condition(B_p(bcp13)))
                                 IMPLIES
                          Val_b(fl,Last_b(bcp13)) = Val_d(asserta_register_2_out1,Last_d(B_p(bcp13)))

eq_cp13_t36_lemma : LEMMA (equivalent_states(First_b(bcp13),First_d(B_p(bcp13))) AND
                          beh_transition_condition(bcp13) AND
                          rtl_transition_condition(B_p(bcp13)))
                                 IMPLIES
                          Val_b(t36,Last_b(bcp13)) = Val_d(asserta_register_3_out1,Last_d(B_p(bcp13)))

eq_cp14_state_lemma : LEMMA (equivalent_states(First_b(bcp14),First_d(B_p(bcp14))) AND
                          beh_transition_condition(bcp14) AND
                          rtl_transition_condition(B_p(bcp14)))
                                 IMPLIES
                          Val_b(state,Last_b(bcp14)) =Val_d(asserta_register_1_out1,Last_d(B_p(bcp14)))

eq_cp14_st_lemma : LEMMA (equivalent_states(First_b(bcp14),First_d(B_p(bcp14))) AND
                          beh_transition_condition(bcp14) AND
                          rtl_transition_condition(B_p(bcp14)))
                                 IMPLIES
                          Val_b(st,Last_b(bcp14)) = Val_d(asserta_register_2_out1,Last_d(B_p(bcp14)))

eq_cp14_hl_lemma : LEMMA (equivalent_states(First_b(bcp14),First_d(B_p(bcp14))) AND
                          beh_transition_condition(bcp14) AND
                          rtl_transition_condition(B_p(bcp14)))
                                 IMPLIES
                          Val_b(hl,Last_b(bcp14)) = Val_d(asserta_register_3_out1,Last_d(B_p(bcp14)))

eq_cp14_fl_lemma : LEMMA (equivalent_states(First_b(bcp14),First_d(B_p(bcp14))) AND
                          beh_transition_condition(bcp14) AND
                          rtl_transition_condition(B_p(bcp14)))
                                 IMPLIES
                          Val_b(fl,Last_b(bcp14)) = Val_d(asserta_register_4_out1,Last_d(B_p(bcp14)))

eq_cp14_cof_lemma : LEMMA (equivalent_states(First_b(bcp14),First_d(B_p(bcp14))) AND
                          beh_transition_condition(bcp14) AND
                          rtl_transition_condition(B_p(bcp14)))
                                 IMPLIES
                          Val_b(cof,Last_b(bcp14)) = Val_d(asserta_register_5_out1,Last_d(B_p(bcp14)))

eq_cp14_tol_lemma : LEMMA (equivalent_states(First_b(bcp14),First_d(B_p(bcp14))) AND
                          beh_transition_condition(bcp14) AND
                          rtl_transition_condition(B_p(bcp14)))
                                 IMPLIES
                          Val_b(tol,Last_b(bcp14)) = Val_d(asserta_register_6_out1,Last_d(B_p(bcp14)))

eq_cp14_tos_lemma : LEMMA (equivalent_states(First_b(bcp14),First_d(B_p(bcp14))) AND
                          beh_transition_condition(bcp14) AND
                          rtl_transition_condition(B_p(bcp14)))
                                 IMPLIES
                          Val_b(tos,Last_b(bcp14)) = Val_d(asserta_register_7_out1,Last_d(B_p(bcp14)))

eq_cp15_state_lemma : LEMMA (equivalent_states(First_b(bcp15),First_d(B_p(bcp15))) AND
                          beh_transition_condition(bcp15) AND
                          rtl_transition_condition(B_p(bcp15)))
                                 IMPLIES
                          Val_b(state,Last_b(bcp15)) =Val_d(asserta_register_1_out1,Last_d(B_p(bcp15)))

eq_cp15_st_lemma : LEMMA (equivalent_states(First_b(bcp15),First_d(B_p(bcp15))) AND
                          beh_transition_condition(bcp15) AND
                          rtl_transition_condition(B_p(bcp15)))
                                 IMPLIES
                          Val_b(st,Last_b(bcp15)) = Val_d(asserta_register_2_out1,Last_d(B_p(bcp15)))
```

```
eq_cp15_hl_lemma : LEMMA (equivalent_states(First_b(bcp15),First_d(B_p(bcp15))) AND
                          beh_transition_condition(bcp15) AND
                          rtl_transition_condition(B_p(bcp15)))
                                IMPLIES
                          Val_b(hl,Last_b(bcp15)) = Val_d(asserta_register_3_out1,Last_d(B_p(bcp15)))

eq_cp15_fl_lemma : LEMMA (equivalent_states(First_b(bcp15),First_d(B_p(bcp15))) AND
                          beh_transition_condition(bcp15) AND
                          rtl_transition_condition(B_p(bcp15)))
                                IMPLIES
                          Val_b(fl,Last_b(bcp15)) = Val_d(asserta_register_4_out1,Last_d(B_p(bcp15)))

eq_cp15_cof_lemma : LEMMA (equivalent_states(First_b(bcp15),First_d(B_p(bcp15))) AND
                          beh_transition_condition(bcp15) AND
                          rtl_transition_condition(B_p(bcp15)))
                                IMPLIES
                          Val_b(cof,Last_b(bcp15)) = Val_d(asserta_register_5_out1,Last_d(B_p(bcp15)))

eq_cp15_tol_lemma : LEMMA (equivalent_states(First_b(bcp15),First_d(B_p(bcp15))) AND
                          beh_transition_condition(bcp15) AND
                          rtl_transition_condition(B_p(bcp15)))
                                IMPLIES
                          Val_b(tol,Last_b(bcp15)) = Val_d(asserta_register_6_out1,Last_d(B_p(bcp15)))

eq_cp15_tos_lemma : LEMMA (equivalent_states(First_b(bcp15),First_d(B_p(bcp15))) AND
                          beh_transition_condition(bcp15) AND
                          rtl_transition_condition(B_p(bcp15)))
                                IMPLIES
                          Val_b(tos,Last_b(bcp15)) = Val_d(asserta_register_7_out1,Last_d(B_p(bcp15)))

eq_cp16_st_lemma : LEMMA (equivalent_states(First_b(bcp16),First_d(B_p(bcp16))) AND
                          beh_transition_condition(bcp16) AND
                          rtl_transition_condition(B_p(bcp16)))
                                IMPLIES
                          Val_b(st,Last_b(bcp16)) = Val_d(asserta_register_2_out1,Last_d(B_p(bcp16)))

eq_cp16_hl_lemma : LEMMA (equivalent_states(First_b(bcp16),First_d(B_p(bcp16))) AND
                          beh_transition_condition(bcp16) AND
                          rtl_transition_condition(B_p(bcp16)))
                                IMPLIES
                          Val_b(hl,Last_b(bcp16)) = Val_d(asserta_register_3_out1,Last_d(B_p(bcp16)))

eq_cp16_fl_lemma : LEMMA (equivalent_states(First_b(bcp16),First_d(B_p(bcp16))) AND
                          beh_transition_condition(bcp16) AND
                          rtl_transition_condition(B_p(bcp16)))
                                IMPLIES
                          Val_b(fl,Last_b(bcp16)) = Val_d(asserta_register_4_out1,Last_d(B_p(bcp16)))

eq_cp16_cof_lemma : LEMMA (equivalent_states(First_b(bcp16),First_d(B_p(bcp16))) AND
                          beh_transition_condition(bcp16) AND
                          rtl_transition_condition(B_p(bcp16)))
                                IMPLIES
                          Val_b(cof,Last_b(bcp16)) = Val_d(asserta_register_5_out1,Last_d(B_p(bcp16)))

eq_cp16_tol_lemma : LEMMA (equivalent_states(First_b(bcp16),First_d(B_p(bcp16))) AND
                          beh_transition_condition(bcp16) AND
                          rtl_transition_condition(B_p(bcp16)))
                                IMPLIES
                          Val_b(tol,Last_b(bcp16)) = Val_d(asserta_register_6_out1,Last_d(B_p(bcp16)))

eq_cp16_tos_lemma : LEMMA (equivalent_states(First_b(bcp16),First_d(B_p(bcp16))) AND
                          beh_transition_condition(bcp16) AND
                          rtl_transition_condition(B_p(bcp16)))
                                IMPLIES
                          Val_b(tos,Last_b(bcp16)) = Val_d(asserta_register_7_out1,Last_d(B_p(bcp16)))

eq_cp17_cof_lemma : LEMMA (equivalent_states(First_b(bcp17),First_d(B_p(bcp17))) AND
                          beh_transition_condition(bcp17) AND
                          rtl_transition_condition(B_p(bcp17)))
                                IMPLIES
                          Val_b(cof,Last_b(bcp17)) = Val_d(asserta_register_5_out1,Last_d(B_p(bcp17)))

eq_cp17_tol_lemma : LEMMA (equivalent_states(First_b(bcp17),First_d(B_p(bcp17))) AND
                          beh_transition_condition(bcp17) AND
                          rtl_transition_condition(B_p(bcp17)))
                                IMPLIES
                          Val_b(tol,Last_b(bcp17)) = Val_d(asserta_register_6_out1,Last_d(B_p(bcp17)))
```

```
eq_cp17_tos_lemma : LEMMA (equivalent_states(First_b(bcp17),First_d(B_p(bcp17))) AND
                           beh_transition_condition(bcp17) AND
                           rtl_transition_condition(B_p(bcp17)))
                                IMPLIES
                           Val_b(tos,Last_b(bcp17)) = Val_d(asserta_register_7_out1,Last_d(B_p(bcp17)))
```

Figure B.10: **TLC Critical Path Equivalence Lemmas**

```
(AUTO-REWRITE-THEORY "tlc" :ALWAYS? T)
(AUTO-REWRITE-THEORY "tlc_control" :ALWAYS? T)
(FLATTEN)
(LEMMA "bs1_bs2_ax")
(LEMMA "bs2_bs3_ax")
(LEMMA "bs3_bs4_ax")
(LEMMA "bs4_bs1004_ax")
(LEMMA "btc_bcp0_ax")
(BDDSIMP)
(LEMMA "Bp_bcp0_ax")
(REPLACE -1)
(EXPAND "beh_transition_condition")
(EXPAND "rtl_transition_condition")
(EXPAND "First_b")
(EXPAND "First_d")
(EXPAND "Last_b")
(EXPAND "Last_d")
(ASSERT)
(REPEAT (EXPAND "reverse"))
(REPEAT (EXPAND "append"))
(LEMMA "equivalent_states_bs1_ds1_ax)
(LEMMA "equivalent_states_bs1004_ds6_ax)
(BDDSIMP)
(LEMMA "dtc_dcp0_ax")
(ASSERT)
(BDDSIMP)
(LEMMA "asserta_register_5_ax")
(LEMMA "asserta_register_ax" ("input" "asserta_mux_5_out1" "output" "asserta_register_5_out1" "ld"
       "asserta_register_5_ld" "s1" "ds5" "s2" "ds6"))
(EXPAND "Control_Signal")
(ASSERT)
(LEMMA "asserta_register_ax" ("input" "asserta_mux_5_out1" "output" "asserta_register_5_out1" "ld"
       "asserta_register_5_ld" "s1" "ds4" "s2" "ds5"))
(EXPAND "Control_Signal")
(ASSERT)
(LEMMA "asserta_register_ax" ("input" "asserta_mux_5_out1" "output" "asserta_register_5_out1" "ld"
       "asserta_register_5_ld" "s1" "ds3" "s2" "ds4"))
(EXPAND "Control_Signal")
(ASSERT)
(LEMMA "asserta_mux_5_ax")
(LEMMA "asserta_mux_4_1_ax" ("input1" "asserta_register_7_out1" "input2" "asserta_constreg_cof_in0_out1"
       "input3"  "asserta_register_2_out1" "input4"  "UNCONNECTED" "sel0" "asserta_mux_5_sel_0""sel1"
       "asserta_mux_5_sel_1" "output" "asserta_mux_5_out1" "s" "ds3"))
(EXPAND "Control_Signal")
(ASSERT)
(LEMMA "asserta_register_2_ax")
(LEMMA "asserta_register_ax" ("input" "asserta_mux_2_out1" "output" "asserta_register_2_out1" "ld"
       "asserta_register_2_ld" "s1" "ds2" "s2" "ds3"))
(EXPAND "Control_Signal")
(ASSERT)
(LEMMA "asserta_register_ax" ("input" "asserta_mux_2_out1" "output" "asserta_register_2_out1" "ld"
       "asserta_register_2_ld" "s1" "ds1" "s2" "ds2"))
(EXPAND "Control_Signal")
(ASSERT)
(ASSERT)
(PROP)
(ASSERT)
```

Figure B.11: **Proof Script for Lemma eq_cp0_st_lemma**

```
(AUTO-REWRITE-THEORY "tlc" :ALWAYS? T)
(AUTO-REWRITE-THEORY "tlc_control" :ALWAYS? T)
(FLATTEN)
(LEMMA "bs1004_bs6_ax")
(LEMMA "bs6_bs7_ax")
(LEMMA "bs7_bs8_ax")
(LEMMA "bs8_bs9_ax")
(LEMMA "bs9_bs10_ax")
(LEMMA "bs10_bs1010_ax")
(LEMMA "btc_bcp1_ax")
(BDDSIMP)
(LEMMA "Bp_bcp1_ax")
(REPLACE -1)
(EXPAND "beh_transition_condition")
(EXPAND "rtl_transition_condition")
(EXPAND "First_b")
(EXPAND "First_d")
(EXPAND "Last_b")
(EXPAND "Last_d")
(ASSERT)
(REPEAT (EXPAND "reverse"))
(REPEAT (EXPAND "append"))
(LEMMA "equivalent_states_bs1004_ds6_ax)
(LEMMA "equivalent_states_bs1010_ds12_ax)
(BDDSIMP)
(LEMMA "dtc_dcp1_ax")
(ASSERT)
(BDDSIMP)
(LEMMA "asserta_register_1_ax")
(LEMMA "asserta_register_ax" ("input" "asserta_mux_1_out1" "output" "asserta_register_1_out1" "ld"
       "asserta_register_1_ld" "s1" "ds11" "s2" "ds12"))
(EXPAND "Control_Signal")
(ASSERT)
(LEMMA "asserta_register_ax" ("input" "asserta_mux_1_out1" "output" "asserta_register_1_out1" "ld"
       "asserta_register_1_ld" "s1" "ds10" "s2" "ds11"))
(EXPAND "Control_Signal")
(ASSERT)
(LEMMA "asserta_register_ax" ("input" "asserta_mux_1_out1" "output" "asserta_register_1_out1" "ld"
       "asserta_register_1_ld" "s1" "ds9" "s2" "ds10"))
(EXPAND "Control_Signal")
(ASSERT)
(LEMMA "asserta_mux_1_ax")
(LEMMA "asserta_mux_8_1_ax" ("input1"  "asserta_register_4_out1" "input2"  "asserta_constreg_3_out1"
       "input3"  "UNCONNECTED" "input4"  "asserta_register_2_out1" "input5"  "asserta_constreg_1_out1"
       "input6"  "asserta_constreg_3_out1" "input7"  "asserta_register_3_out1" "input8"  "UNCONNECTED"
       "sel0" "asserta_mux_1_sel_0""sel1" "asserta_mux_1_sel_1""sel2" "asserta_mux_1_sel_2" "output"
       "asserta_mux_1_out1" "s" "ds9"))
(EXPAND "Control_Signal")
(ASSERT)
(LEMMA "asserta_constreg_1_ax")
(LEMMA "asserta_constreg_ax" ("val" "CONST_10" "output" "asserta_constreg_1_out1" "s" "ds9"))
(PROP)
(ASSERT)
(PROP)
(ASSERT)
```

Figure B.12: **Proof Script for Lemma eq_cp1_hl_lemma**

Appendix C

Symbolic Analysis of Specifications and Implementations

In the previous chapters, the equivalence between a behavior specification and its register transfer level implementation was formally defined, and a verification approach based on this definition of equivalence was presented. In this approach, certain states of the behavior automaton are selected as check points or points of comparison of the two designs (specification and implementation). These states are referred to as critical behavior states. Controller states equivalent to the behavior critical states (which are identified by the synthesis tool), are the critical states of the implementation design. Certain (or sometimes all) specification variables are critical. These variables are critical, since they hold the results of the behavior operations, and the correct function of the design is defined in terms of the values they hold at critical states. The RTL registers, that are the physical representation of the critical specification variables, are the critical registers of the implementation design. The register binding information provided by the high-level synthesis system is used to identify the critical RTL registers. The critical paths were defined as state transition paths between the critical states (for the exact definition refer to section 6.1). Informally, the verification process consists of establishing that if at the initial states of corresponding pairs of behavior-controller critical paths, critical variables and critical registers have equal values, then at the final states of those paths they have equal values, too. Then, to prove the equivalence, a set of lemmas - one lemma for each critical variable along each critical path - should be proven.

In Chapter 7, we mentioned that these proofs are constructed hierarchically. As part of proof construction at the lowest level, behavior operations and register transfer operations are analyzed and symbolic values for the behavior variables and design registers are calculated. We use symbolic analysis for performing these calculation and providing the proofs. Symbolic analysis is used as the behavior operations and RTL functional units in our models are uninterpreted. We will see in the following sections that when assuming uninterpreted domain of values for behavior variables and design registers, symbolic evaluation of the variables and symbolic analysis of the effects of register transfer operations are suitable and sufficient for constructing the proof of equivalence. This is not the case if we consider the real values of the specification variables or the implementation registers. When using symbolic methods, the lemmas are proved independent from the specific values of the variables or registers at the initial states of the critical paths. The proofs hold for all possible values of the critical variables and registers at the initial or final states of the paths. However, if we want to prove the same lemmas by interpreting the operations and observing their effects on the real values of the variables and registers, then all possible values of the variables and registers at the initial states of the critical paths, as well as all possible combinations of these values should be considered. In such a case complexity of the verification algorithm increases dramatically. In this situation, model checking or simulation (that each has its obvious draw backs) should be used in place of symbolic methods, in order to establish equivalence exhaustively.

The descriptions of the behavior(specification) and structure(implementation) of a design define the relation of the output values to the input values. In other words, they specify how the outputs of each component respond to the values arriving at its inputs. We noticed that even though these descriptions define relations between the inputs and outputs, and do not depend on specific input or output values, they can be used for predicting and comparing the behavior of the specification and implementation designs. For example, it may not be possible to calculate the exact value of a specification variable, upon the exit from a loop; but, we can still derive a relation between the value of that variable at the beginning and its value at the end of each iteration of the loop. Similarly, the relation between the content of a register at the beginning and its content at the end of each iteration of loop constructs of the controller may be derived. This style of analysis is often sufficient for proving interesting properties of the implementation design. For example, with an analysis such as above, we can prove that a particular input-output pair of the implementation description have the exact same relation as their corresponding input-out pair in the behavior description. These proofs can be conducted through symbolic analysis of the values of the variables in the specification description and the contents of the registers of the implementation design, when they are both formally specified by models like those presented in the previous chapters. Symbolic analysis is a means for verifying that the 'satisfaction' or 'correctness' between the specification and implementation holds.

Symbolic evaluation is simply calculating the values of a set of variables or the contents of a set of registers after a sequence of state transitions in terms of their values or their contents prior to those state transitions. In the following sections the symbolic analysis of the specification expressions, and symbolic analysis of the data-path operations will be discussed. Examples are provided to clarify the concepts.

C.1 Specification Variables

The value of each specification variable after a sequence of behavior state transitions can be calculated in terms of the value of that as well as other specification variables prior to that transition sequence. If many conditional transitions are present in the sequence then the symbolic expression representing the values may be very complex. In such cases, the same states may be revisited multiple times, or some states may be visited conditionally. Therefore, the same variable can assume multiple values at the same state, and the values of the variables cannot be uniquely defined. In symbolic analysis of the specification variables in this application, the longest transition sequence is a critical path. No cycles are present in a critical path, therefore, at most one conditional transition is present in the sequence and the symbolic values of the variables at each state can be defined unambiguously.

In the following sections, we will explain the symbolic evaluation of the mathematical and logical expressions. Symbolic evaluation is a process in which the symbolic value of the expression is calculated in terms of the symbolic values of its subexpressions. Symbolic evaluation is therefore a recursive process, where the evaluation of an expression is reduced to evaluation of its subexpressions, and this continues recursively until all the subexpressions are simple terms or expressions, and their symbolic values do not require further calculation.

C.1.1 Mathematical and Logical Expressions

Given the set of the specification variables $\langle r_{b1}, r_{b2}, \cdots, r_{bn}\rangle$, and $\langle val_1, val_2, \cdots, val_n\rangle$, their symbolic values at a state s_{b_i}, the values of mathematical and logical expressions of the behavioral specification at state s_{b_i}, can be evaluated in terms of these symbolic values.

Suppose f is a mathematical (+,-,$\cdots$), logical ($\wedge$, $\vee$, $\cdots$) or a relational ($<$, $\equiv$, $>$, $\cdots$) function. A behavior expression can have three different forms. The symbolic evaluation of these expressions is as follow:

1. $exp = \mathbf{c}$:

$$sym\_eval(exp, s_{b_i})|_{(\langle r_{b1}, r_{b2}, \cdots, r_{bn}\rangle \,:=\, \langle val_1, val_2, \cdots, val_n\rangle)} = c$$

2. $exp = \mathbf{r_{bk}}$:

$$sym\_eval(exp, s_{b_i})|_{(\langle r_{b1}, r_{b2}, \cdots, r_{bn}\rangle \,:=\, \langle val_1, val_2, \cdots, val_n\rangle)} = val_k$$

3. $exp = f(r_{b1}, r_{b2}, \cdots, r_{bn})$:

$$\begin{array}{l} sym\_eval(exp, s_{b_i})|_{(\langle r_{b1},\cdots,r_{bn}\rangle \,:=\, \langle val_1,\cdots,val_n\rangle)} = \\ sym\_eval(f(r_{b1}, r_{b2}, \cdots, r_{bn}), s_{b_i})|_{(\langle r_{b1},\cdots,r_{bn}\rangle \,:=\, \langle val_1,\cdots,val_n\rangle)} = \\ f(sym\_eval(r_{b1}|_{(\langle r_{b1},\cdots,r_{bn}\rangle \,:=\, \langle val_1,\cdots,val_n\rangle)}, s_{b_i}), \\ \quad sym\_eval(r_{b2}|_{(\langle r_{b1},\cdots,r_{bn}\rangle \,:=\, \langle val_1,\cdots,val_n\rangle)}, s_{b_i}), \\ \quad \cdots, \\ \quad sym\_eval(r_{bn}|_{(\langle r_{b1},\cdots,r_{bn}\rangle \,:=\, \langle val_1,\cdots val_n\rangle)}, s_{b_i})) = \\ f(val_1, val_2, \cdots, val_n) \end{array}$$

C.1.2 Statements in the Specifications

In the formal model of the specification, each behavior state corresponds to a statement of the behavior description. Behavior statements define design operations. Each operation may affect the value of a specification variable. By symbolic analysis of these statements we can calculate new symbolic values for the specification variables in a transition from one state to the other. As mentioned before each state is either conditional or non-conditional. The conditional states correspond to conditional statements such as **if** and **while** statements. Non-conditional states correspond to assignment statements. Suppose $\langle var_1, var_2, \cdots, var_n\rangle$ are the specification variables, $\langle val_1, val_2, \cdots, val_n\rangle$ is the value set of these variables at a state s_b and $\langle s_b, s'_b\rangle$ is a state transition. The symbolic values of the specification variables at s'_b, the target of state transition s'_b, can be calculated in terms of $\langle val_1, val_2, \cdots, val_n\rangle$, the symbolic values of the specification variables at the source of the state transition s_b, according to the following rules:

unconditional state transitions - As mentioned in section 4.4 the statement corresponding to s_b, the source of an unconditional state transition $\langle s_b, s'_b\rangle$ is an assignment statement. If s_b corresponds to a statement of the form:

$$\mathbf{r_{bk} := exp;}$$

where r_{bk} is one of the specification variables and exp a mathematical or logical expression, then $\langle val'_1, val'_2, \cdots, val'_n\rangle$, the set of symbolic values of the variables at state s'_b, after the transition $\langle s_b, s'_b\rangle$ is defined as:

$$\forall i, 1 \leq i \leq n \;:\; val'_i = \begin{cases} val_i & (i \neq k) \\ sym\_eval(exp, s_b)|_{(\langle r_{b1}, r_{b2},\cdots,r_{bn}\rangle \,:=\, \langle val_1, val_2,\cdots,val_n\rangle)} & (i = k) \end{cases}$$

Behavior State	lt	list[i]	list[j]	temp
bs_4	(list[j](bs_3) < list[j](bs_3))	list[i](bs_3)	list[j](bs_3)	temp(bs_3)
bs_5	lt(bs_4)	list[i](bs_4)	list[j](bs_4)	temp(bs_4)
bs_6	lt(bs_5)	list[i](bs_5)	list[j](bs_5)	list[j](bs_5)
bs_7	lt(bs_6)	list[i](bs_6)	list[i](bs_6)	temp(bs_6)
bs_8	lt(bs_7)	temp(bs_7)	list[j](bs_6)	temp(bs_7)

Table C.1: **Symbolic Evaluation of Specification Variables after State Transitions**

This can easily be extended to the case where a multidimensional variable is the target of an assignment, *e.g.* if the statement corresponding to the state s'_b is of the form:

$$\mathbf{r_{bk_1}[r_{bk_2}] := exp;}$$

in which r_{bk_1} is an array with m elements then $\langle val'_1, val'_2, \cdots, val'_n \rangle$ is defined as follows:
$\forall i \forall j,\ \ (1 \le i \le n)\ \wedge\ (1 \le j \le m)\ :$

$$val'_i = \begin{cases} val_i & (i \neq k_1) \\ val_i[val_j] & (i = k_1 \wedge j \neq k_2) \\ sym\_eval(exp, s_b)|_{(\langle r_{b1}, r_{b2}, \cdots, r_{bn} \rangle \,:=\, \langle val_1, val_2, \cdots, val_n \rangle)} & (i = k_1 \wedge j = k_2) \end{cases}$$

conditional state transitions - According to the discussion of section 4.4, the statement corresponding to s_b the source of a conditional state transition $\langle s_b, s'_b \rangle$ is a conditional statement. A conditional statement is a statement like "**if** (*conditional_exp*) **then**" or "**while** (*conditional_exp*) **do**" , *etc.* The *conditional_exp* can be of the form $f(r_{b1}, r_{b2}, \cdots, r_{bn})$ which can be symbolically evaluated. No operation is performed at the state s_b, so the value of each specification variable at the state s'_b is the same as its value at the state s_b. $\langle val'_1, val'_2, \cdots, val'_n \rangle$, the value set of specification variables at state s'_b is defined as follows:

$$\forall i, 1 \le i \le n\ \ :\ \ val'_i = val_i \tag{C.1}$$

Example - Figure C.1 shows a partial behavior description corresponding to Bubble-Sort algorithm. There are five specification variables and five behavior states in this partial description. The symbolic evaluation of the specification variables after each state transition is given in Table C.1.

C.1.3 Specification Variables and Transition Sequences

The above procedure helps us, to calculate the values of the specification variables at the state that is the target of a state transition in terms of the values (or symbolic values) of the specification

```
($bs_3$)   lt := (list(j) < list(i));

($bs_4$)   **if**  (lt)  **then**
($bs_5$)        temp := list(j);
($bs_6$)        list(j) <= list(i);
($bs_7$)        list(i) <= temp;
            **endif**;
($bs_8$)
```

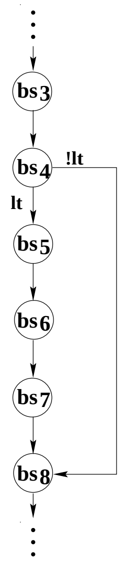


Figure C.1: **Partial Behavioral Specification of Bubble-Sort**

variables at the state that is the source of that state transition. The same concept applies when we want to calculate the values of the specification variables at the final state (target) of a sequence of state transitions in terms of the values (or symbolic values) of the specification variables at the initial state (source) of the sequence of state transitions. The new symbolic values, after the first transition in the sequence, are calculated as it was explained in previous section, then, the symbolic values after the second transition are calculated in terms of the symbolic values after the first transition and these calculations are continued successively until the final state of the sequence is reached. This method of evaluating symbolic values by forward traversal of the critical paths of the behavior automaton is used in our verification process to calculate the values of the specification variables at the final state of each critical path in terms of their values at the initial state of the critical path. We believe that symbolic evaluation is one of the key factors in relatively low verification time associated with this approach. By performing symbolic analysis, a decision about the equivalence of the specification and implementation descriptions can be made without any calculations with real values (exhaustive calculations).

Example - In Figure C.1, the transition sequence $\{\langle bs_4, bs_5\rangle, \langle bs_5, bs_6\rangle, \langle bs_6, bs_7\rangle, \langle bs_7, bs_8\rangle\}$ corresponds to a critical path. The value of the specification variables at bs_8 the final state of the path, can be symbolically evaluated in terms of their values at bs_4 the initial state of the path, as given in Table C.2.

C.2 Register Transfer Operations

By analysis of the register transfer operations in the data-path, we can trace the flow of the data in the RTL design and provide information about the contents of the data-path registers. In our verification exercise, the value transfers are analyzed by considering symbolic values for registers instead of real values. Therefore, we refer to this process as *symbolic analysis of the register transfer operations*. During the verification process, the equivalence of the specification and implementation is established by comparing the values of the critical variables with contents of critical registers. As we considered symbolic values for the critical variables, the contents of critical registers should also be calculated symbolically. These values can only be obtained by the knowledge of the data-path structure and by analysis of the operations performed at each state by following the controller commands. The *symbolic evaluation* of the contents of the registers in the data-path is done through backward traversal of the data-path components, backward trace of the sequence of the state transitions of the controller or both. The goal is to calculate the contents of each critical register at s'_d, the state that is the target of a specific controller state transition $\langle s_d, s'_d\rangle$, in terms of the values of that or some other critical registers at s_d, the state that is the source of the state transition. The value of a critical register at the controller state s'_d, is either equal to its value at

Behavior State	lt	list[i]	list[j]	temp
bs_4	lt(bs_4)	list[i](bs_4)	list[j](bs_4)	temp(bs_4)
bs_5	lt(bs_4)	list[i](bs_4)	list[j](bs_4)	temp(bs_4)
bs_6	lt(bs_4)	list[i](bs_4)	list[j](bs_4)	list[j](bs_4)
bs_7	lt(bs_4)	list[i](bs_4)	list[i](bs_4)	list[j](bs_4)
bs_8	lt(bs_4)	list[j](bs_4)	list[i](bs_4)	list[j](bs_4)

Table C.2: **Symbolic Evaluation of Specification Variables along A Critical Path**

the previous state s_d, or is the same as the value at its input at the current state s'_d, which is the output of another component. To calculate this value in the first case, the value of the register at the previous state, and in the second case, the value of some other component's output at current state should be calculated. In both situations, we need to continue this backward traversal until the symbolic content of the register can be calculated in terms of the symbolic contents of critical components at a critical state, in which case the evaluation is complete. In the following sections, the symbolic analysis of the data transfers in an RTL design will be discussed in detail.

C.3 Combinational and Sequential Components

Let's assume that $\langle r_{d1}, r_{d2}, \cdots, r_{dn}\rangle$ is the set of critical design registers, and $\langle s_d, s_{d'}\rangle$ is a state transition of the controller. Also, assume that $\langle data_1, data_2, \cdots, data_n\rangle$ and $\langle data'_1, data'_2, \cdots, data'_n\rangle$ are the symbolic values of the critical registers at states s_d and $s_{d'}$, respectively. The symbolic analysis of the data-transfers from the input to the output of different components are as follows:

- **Basic Combinational Components -** The value (real or symbolic) at the output of a combinational component at each state, can be calculated in terms of the values (real or symbolic) of its inputs at that state. Assume a component C, with n_i inputs and n_o outputs. This component performs the operations f_1, f_2, $\cdots$ and f_n on the set of inputs $C\_in[1]$, $C\_in[2]$, $\cdots$ and $C\_in[n_i]$ to generate the output values $C\_out[1]$, $C\_out[2]$, $\cdots$ and $C\_out[n_o]$ respectively. The output $C\_out[k]$ may be symbolically evaluated in terms of the inputs according to the following rule:

$$\begin{aligned}&sym\_eval(C\_out[k], s_d) =\\&sym\_eval(f_k(C\_in[1], C\_in[2], \cdots, C\_in[n_i]), s_d) =\\&f_k(sym\_eval(C\_in[1], s_d), sym\_eval(C\_in[2], s_d), \cdots, sym\_eval(C\_in[n_i], s_d))\end{aligned}$$

An example of a combinational component is a multiplier. A multiplier (shown in Figure

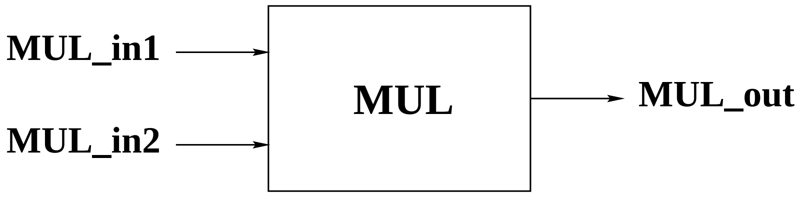


Figure C.2: **Multiplier Component**

C.2) has two inputs and one output. The output of the multiplier can be evaluated in terms of its inputs as follows:

$$sym\_eval(MUL\_out, s_d) = sym\_eval(mul(MUL\_in_1, MUL\_in_2), s_d)$$

or

$$sym\_eval(MUL\_out, s_d) = mul(sym\_eval(MUL\_in_1, s_d), sym\_eval(MUL\_in_2, s_d))$$

- **Basic Sequential Components -** The value (real or symbolic) at the output of a sequential component at each state, may be calculated in terms of the values (real or symbolic) of its inputs at that state, or its output at a previous state. If the value (real or symbolic) of an output C_out of a sequential component C, at the state s'_d is a function f of its inputs $C\_in[1]$, $C\_in[2]$, $\cdots$, $C\_in[n_i]$ at the state s'_d, or its value at s_d, the previous state of s'_d, then C_out can be symbolically evaluated as follows:

$$\begin{aligned}
&sym\_eval(C\_out, s'_d) = \\
&f(sym\_eval(f_1(C\_in[1], C\_in[2], \cdots, C\_in[n_i]), s'_d), sym\_eval(f_2(C\_out), s_d) = \\
&f(f_1(sym\_eval(C\_in[1], s'_d), \\
&\qquad sym\_eval(C\_in[2], s'_d), \\
&\qquad \cdots, \\
&\qquad sym\_eval(C\_in[n_i], s'_d)), \\
&\quad f_2(sym\_eval(C\_out, s_d)))
\end{aligned}$$

The rules of symbolic evaluation for two basic sequential components, Register and RAM are given below.

1. **REG :** Suppose $1 \leq i \leq n$ (n is the number of critical registers at state s'_d), then:

$$sym\_eval(REG\_out, s'_d) = \begin{cases} data_i & ld = 0 \wedge (\exists i : REG = r_{di}) \\ sym\_eval(REG\_out, s_d) & ld = 0 \wedge (\nexists i : REG = r_{di}) \\ sym\_eval(REG\_in, s'_d) & ld = 1 \end{cases}$$

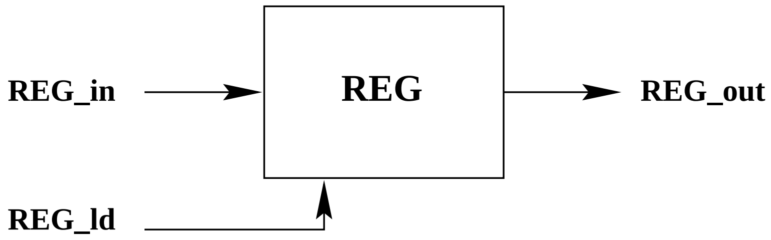


Figure C.3: **Register Component**

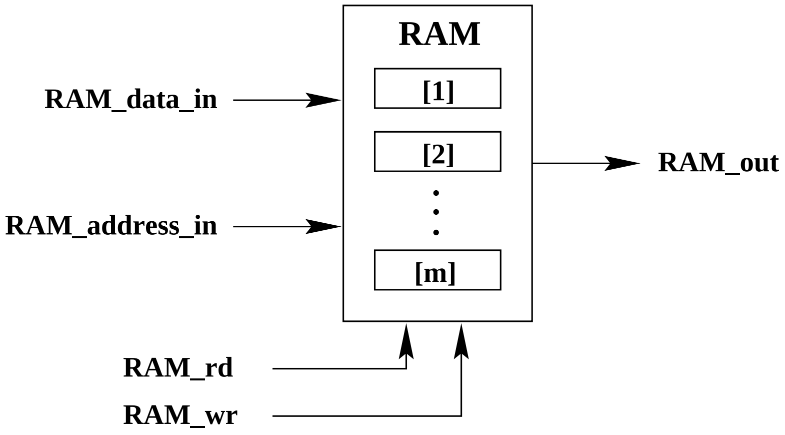


Figure C.4: **Random Access Memory Component**

2. **RAM :** Suppose $1 \leq i \leq n$ and $1 \leq j \leq m$, where n is the number of critical registers at state s'_d and m is the number of elements of RAM. Then :
   $address = sym\_eval(RAM\_address\_in, s'_d)$

   $sym\_eval(RAM[j], s'_d) =$

$$\begin{cases} sym\_eval(RAM[j], s_d) & wr = 0 \wedge \nexists i : RAM = r_{di} \\ data_i[j] & wr = 0 \wedge (\exists i : RAM = r_{di} \\ sym\_eval(RAM\_data\_input, s'_d) & wr = 1 \wedge (\exists j : address = j) \end{cases}$$

   $sym\_eval(RAM\_out, s'_d) =$

$$\begin{cases} sym\_eval(RAM[sym\_eval(RAM\_address\_in, s'_d)], s'_d) & rd = 1 \\ sym\_eval(RAM\_out, s_d) & rd = 0 \end{cases}$$

C.3.1 Data Transfers in an RTL Design

In the previous section, we discussed the evaluation of symbolic values of the outputs of some basic RTL components at certain controller states, in terms of the symbolic values of their inputs at that state, or the symbolic values of their outputs at the previous state. The data-path of an RTL design is an interconnection of different basic components. The rules presented in the previous section can be used to evaluate the symbolic values of the outputs of different data-path components at different states. In these evaluations, the interconnections of the components and the data transfers through these interconnections should be taken into account. To see how the interconnections of the components should be taken into account, consider the simple example of Figure C.5, where C_1, C_2 and C_3 are combinational components.

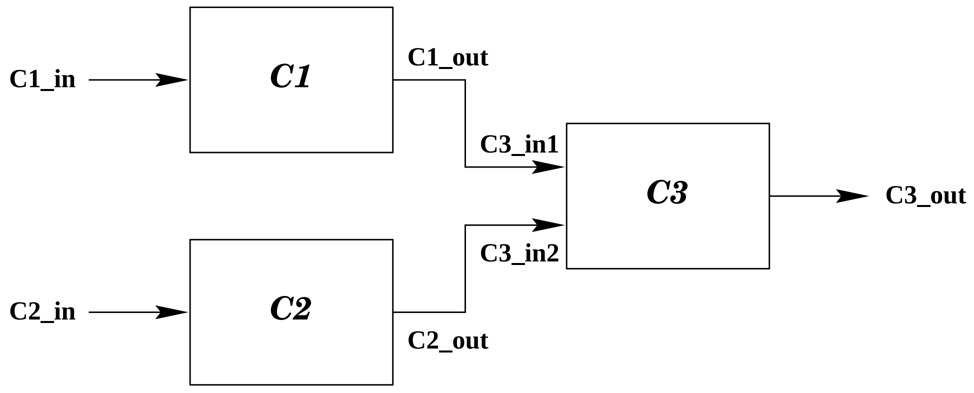


Figure C.5: **Example of Interconnection of the Data-Path Components**

The output of each component can be symbolically evaluated in terms of its inputs:

$$sym\_eval(C_1\_out, s_d) = f_1(sym\_eval(C_1\_in, s_d))$$
$$sym\_eval(C_2\_out, s_d) = f_2(sym\_eval(C_2\_in, s_d))$$
$$sym\_eval(C_3\_out, s_d) = f_3(sym\_eval(C_3\_in_1, s_d), sym\_eval(C_3\_in_2, s_d))$$

Considering the interconnections between the components and the fact that:

$C_3\_in_1 = C_1\_out$ and
$C_3\_in_2 = C_2\_out$

and that:

$\forall s_d \in S_d :$
$sym\_eval(C_3\_in_1, s_d) = sym\_eval(C_1\_out, s_d)$ and
$sym\_eval(C_3\_in_2, s_d) = sym\_eval(C_2\_out, s_d)$

the symbolic value of the output of C_3 can be rewritten as:

$sym\_eval(C_3\_out, s_d) = f_3(f_1(sym\_eval(C_1\_in, s_d)), f_2(sym\_eval(C_2\_in, s_d)))$

A similar approach is used in evaluating the output values of different RTL components in general (combinational or sequential). The symbolic evaluation of the outputs of components is done by (1) backward traversal of the data-path (by calculating the output of a component in terms of its inputs, that in turn are calculated in terms of primary inputs or the outputs of other components (that are calculated in terms of $\cdots$), and/or (2) back-tracing the sequence of state transitions of the controller of the RTL design, in the case of the sequential components, where the output of a component at a certain state s_d, of the controller may be the same as its output at the previous state of s_d. We are not interested in arbitrarily calculating the output of a component in terms of its inputs that are the outputs of some other components, at possibly a different state. Our goal is to calculate these values solely in terms of the values of the critical registers at critical states We want to do a *complete symbolic analysis* of the output values.

Consider an implementation model of an RTL design. Suppose $R_d = \{r_{d1}, r_{d2}, \cdots, r_{dn}\}$ is the set of critical registers of this design. Now, suppose that $cp = \{\langle s_{di}, s_{di+1}\rangle, \langle s_{di+1}, s_{di+2}\rangle, \cdots, \langle s_{dj-1}, s_{dj}\rangle\}$ is a critical path of the controller, and that the symbolic values of the critical registers at s_di, the initial state of the cp is defined as:

$$sym\_eval(\langle r_{d1}, r_{d2}, \cdots, r_{dn}\rangle, s_{di}) = \langle data_1, data_2, \cdots, data_n\rangle$$

The symbolic evaluation of the output of a data-path component at the final state of the critical path s_{dj}, is *complete* if its symbolic value is defined only in terms of $\langle data_1, data_2, \cdots, data_n\rangle$ or constant values, but no other symbolic values.

Example - Figure C.6 shows a part of the data-path and controller of an RTL design. The sequence of states $\{ds_6, ds_7, ds_8, ds_9, ds_{10}\}$ is a critical path of the design and ds_6 and ds_{10} are critical states, and REG_1, REG_2, REG_4 and REG_5 are among critical design registers. REG_3 is not a critical register. In this example we are interested in performing a complete symbolic evaluation of the output of REG_5 at state ds_{10}, i.e. calculating the output of REG_5 at state ds_{10} in terms of the outputs of the critical design registers at state ds_6. We explained before, that this can be done by a backward traversal of the data-path or/and back-trace of controller state transitions as follows:

$REG_5\_out(ds_{10}) = REG_5\_out(ds_9)$
$REG_5\_out(ds_9) = MUX_2\_out(ds_8)$
$MUX_2\_out(ds_8) = ADD\_out(ds_8)$
$ADD\_out(ds_8) = CONS\_REG\_out(ds_8) + REG_3\_out(ds_8)$
$REG_3\_out(ds_8) = MUX_1\_out(ds_7)$

$MUX_1\_out(ds_7) = REG_1\_out(ds_7)$
$REG_1\_out(ds_7) = REG_1\_out(ds_6)$

By performing the necessary substitutions the symbolic value of REG_5 is obtained:

$REG_5\_out(ds_{10}) = CONS\_REG\_out(ds_8) + REG_1\_out(ds_6)$

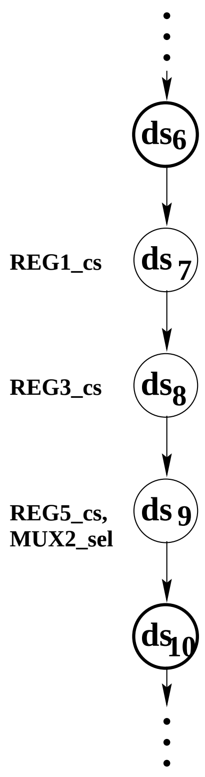
which means:

$REG_5\_out(ds_{10}) = C + REG_1\_out(ds_6)$

and therefore, the symbolic evaluation of the output of REG_5 is complete.

C.3.2 Application of Symbolic Analysis in Verification of Synthesized RTL Designs

In constructing the proof of correctness of RTL designs, we rely on establishing that certain relations between the contents of registers (the values of variables) at some state, and their contents (their values) at some other state exist. We mechanically extract such relations by backward traversal of the data path and/or back-trace of the controller state transitions (forward traversal of the behavior automaton), and use them in our proofs. During the proof checking, the proof checker processes the design descriptions and verifies if such relations may be deduced from the formal descriptions of the RTL design (the behavior specification). The errors in the implementation, as well as any relations that may not be inferred from the design descriptions are detected during the proof checking.



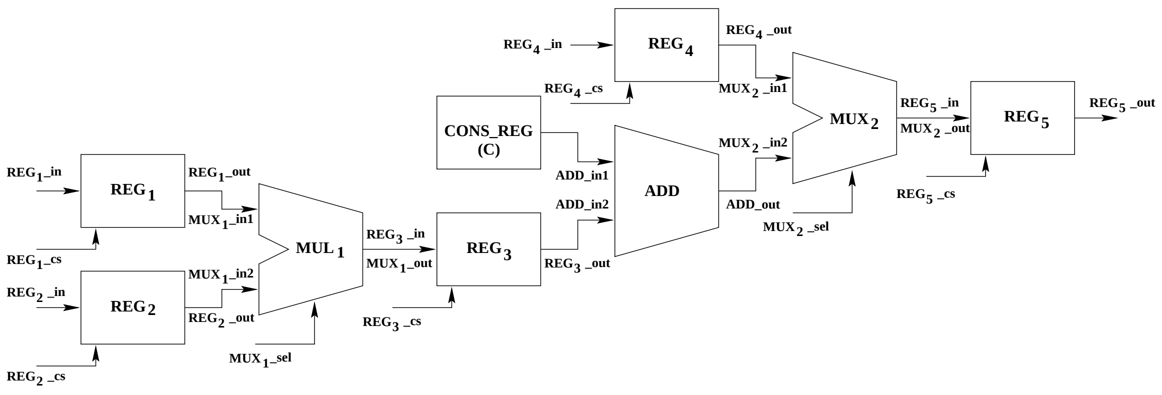


Figure C.6: **Parts of Controller and Data-Path of an RTL Design**

Bibliography

[1] *Guidelines for Formal Verification Systems.* National Computer Security Center, 1989.

[2] *What is Formal Methods?* Nasa Langley Research Center: Formal Methods Program; http://shemesh.larc.nasa.gov/fm/fm-what.html, 2000.

[3] M. Aagaard and M. Leeser, "A Formally Verified System for Logic Synthesis", *Proceedings of 1991 IEEE International Conference on Computer Design*, 1991.

[4] M. Aagaard, M. Leeser, and P. Windley, "Towards a Super Duper Hardware Tactic", *HOL Theorem Proving System and its Applications*, 1993.

[5] S. Basse, "Computer Algorithms", *Addison-Wesley*, 1978.

[6] R. A. Bergamaschi and S. Raje, "Observable Time Windows: Verifying The Results of High-Level Synthesis", *IEEE Design & Test of Computers*, 1997.

[7] R. E. Bryant, "Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams", *Technical Report CMU-CS-92-160,School of Computer Science, Carnegie Mellon University*, 1992.

[8] R. E. Bryant, "Binary Decision Diagrams and Beyond: Enabling Technologies for Formal Verification", *Embedded tutorial at International Conference on Computer-Aided Design*, 1995.

[9] J. R. Burch and D. L. Dill, "Automatic Verification of Pipelined Microprocessor Control", *Proceedings of Computer-Aided Verification*, 1994.

[10] R. Camposano and W. Wolf, "High-Level VLSI Synthesis", *Kluwer Academic Publishers*, 1991.

[11] L. Claesen, M. Genoe, E. Verlind, F. Proesmans, and H. D. Man, "SFG-Tracing: A Methodology of Design for Verifiability", *Proceedings of Advanced Workshop on Correct Hardware Design Methodologies*, 1991.

[12] L. Claesen, F. Proesmans, E. Verlind, and H. D. Man, "SFG-Tracing: A Methodology for the Automatic Verification of MOS Transistor Level Implementations from High-Level Behavioral Specifications", *Proceedings of International Workshop on Formal Methods in VLSI Design*, 1991.

[13] E. Clarke, E. Emerson, and A. Sistla, "Automatic Verification of Finite-State Concurrent Systems using Temporal Logic Specifications", *ACM Transactions on Prog. Lang. Syst.*, 1986.

[14] F. Corella, "Automated Verification of Behavioral Equivalence for Microprocessors", *Research Report, IBM Research division, T.J. Watson Research Center*, 1992.

[15] F. Corella, "Automated High-Level Verification Against Clocked Algorithmic Specifications", *Proceedings of Computer Hardware Description Languages and Their Applications*, 1993.

[16] F. Corella, R. Camposano, R. Bergamaschi, and M. Payer, "Verification of Synchronous Sequential Circuits Obtained from Algorithmic Specifications", *Proceedings of International Workshop on Formal Methods in VLSI Design*, 1991.

[17] G. De-Micheli, "Synthesis and Optimization of Digital Circuits", *McGraw-Hill*, 1994.

[18] R. Dutta, J. Roy, and R. Vemuri, "Distributed Design Space Exploration for High-Level Synthesis Systems", *29th Design Automation Conference*, 1992.

[19] D. Eisnbiegler and R. Kumar, "Formally Embedding Existing High Level Synthesis Algorithms", *Correct Hardware Design and Verification Methods*, 1995.

[20] F. Feldbusch and R. Kumar, "Verification of Synthesized Circuits at Register Transfer Level with Flow Graphs", *Proceedings of IEEE European Design Automation Conference*, 1991.

[21] D. Gajski, N. Dutt, A. Wu, and S. Lin, "High-Level Synthesis, Introduction to Chip and System Design", *Kluwer Academic Publishers*, 1992.

[22] M. Gordon, *Lecture Notes on Specification and Verification.* University of Cambridge Computer Laboratory, 1999.

[23] S. Govindarajan and R. Vemuri, "Cone-Based Clustering Heuristic for List Scheduling Algorithms", *Proceedings of the European Design and Test Conference*, 1997.

[24] S. Govindarajan and R. Vemuri, "Dynamic Bounding of Successor Force Computations in the Force Directed List Scheduling Algorithm", *Proceedings of International Conference on Computer Design*, 1997.

[25] D. Greve, "Symbolic Simulation of the JEM1 Microprocessor", *Proceedings of the 2nd International Conference on Formal Methods in Computer-Aided Design*, November 1998.

[26] J. L. Hennessy and D. A. Patterson, *Computer Architecture A Quantitative Approach.* Morgan Kaufmann Publishers, INC., 1996.

[27] R. Hojati and R. K. Brayton, "Automatic Datapath Abstraction In Hardware Systems", *Proceedings of Computer Aided Verification*, 1995.

[28] A. J. Hu, *Techniques for Efficient Formal Verification Using Binary Decision Diagrams.* Stanford University, 1995.

[29] W. Hunt, *FM8501: A Verified Microprocessor.* 1994.

[30] S. Johnson, "Synthesis of Digital Designs from Recursion Equations", *MIT Press*, 1984.

[31] S. D. Johnson, W. P. Alexander, S.-K. Chin, and G. Gopalakrishnan, "Report on the 21st Century Engineering Consortium Workshop", 1998.

[32] S. Katkoori, J. Roy, and R. Vemuri, "A Hierarchical Register Optimization Algorithm for Behavioral Synthesis", *Proceedings of International Conference on VLSI Design*, 1996.

[33] D. W. Knapp, "Behavioral synthesis : digital system design using the Synopsys Behavioral Compiler"", *Prentice Hall*, 1996.

[34] P. M. Kogge, *The Architecture of Pipelined Computers.* McGraw-Hill Book Company, 1981.

[35] T. Kropf, K. Schneider, and R. Kumar, "A Formal Framework for High Level Synthesis", *Theorem Provers in Circuit Design*, 1994.

[36] F. Kurdahi and A. Parker, "REAL: A Program for REgister ALlocation", *24th Design Automation Conference*, 1987.

[37] M. McFarland, "An Abstract Model of Behavior for Hardware Descriptions", *IEEE Transactions on Computers*, 1983.

[38] K. L. McMillan, "Symbolic Model Checking: An Approach to the State Explosion Problem", *Carnegie Mellon University*, 1992.

[39] T. L. Melham, *Higher Order Logic and Hardware Verification.* Cambridge University Press, 1993.

[40] P. Michel, U. Lauther, and P. Duzy, *The Synthesis Approach to Digital System Design.* Kluwar Academic Publishers, 1992.

[41] R. Milner, *Communication and Concurrency.* Prentice Hall International Series in Computer Science, 1989.

[42] J. S. Moore, "Symbolic Simulation : An ACL2 Approach", *Proceedings of the 2nd International Conference on Formal Methods in Computer-Aided Design*, November 1998.

[43] N. Narasimhan and R. Vemuri, "Specification of Control Flow Properties for Verification of Synthesized VHDL Designs", *Proceedings of International Conference in Formal Methods in Computer Aided Design*, 1996.

[44] N. Narasimhan and R. Vemuri, "Synchronous Controller Models for Synthesis from Communicating VHDL Processes", *Ninth International Conference on VLSI Design*, 1996.

[45] N. Narasimhan and R. Vemuri, "On the Effectiveness of Theorem Proving Guided Discovery of Formal Assertions for a Register Allocator in a High-Level Synthesis System", *Proceedings of 11th Conference on Theorem Proving in Higher Or der Logics (TPHOL'98)*, September 1998.

[46] N. Narasimhan, E. Teica, R. Radhakrishnan, S. Govindarajan, and R. Vemuri, "Theorem Proving Guided Development of Formal Assertions in a R esource-Constrained Scheduler for High-Level Synthesis", *Proceedings of International Conference on Computer Design (ICCD'98)*, pp. 392–399, October 1998.

[47] S. Owre, N. Shankar, J. Rushby, and D. Stringer-Calvert, "PVS Language Reference", *SRI International*, September 1999.

[48] S. Owre, N. Shankar, J. Rushby, and D. Stringer-Calvert, "PVS System Guide", *SRI International*, September 1999.

[49] N. Park and A. C. Parker, "Sehwa: A Software Package for Synthesis of Pipelines from Behavioral Specifications", *IEEE Transactions on Computer Aided Design*, 1988.

[50] P. Pualin, J. Knight, and E. Girczyc, "HAL: A Multi-Paradigm Approach to Automatic Data Path Synthesis", *Proceedindgs of the 24th ACM/IEEE Desgin Automation Conference*, 1986.

[51] S. Rajan, "'Correctness Transformations in High Level Synthesis: Formal Verification", *Proceedings of the International Conference on Computer Hardware Description Languages*, 1995.

[52] S. Rajan, J. Joyce, and C. Seger, "From Abstract Data Types to Shift Registers : A Case Study in Formal Specification and Verification at Differing Levels of Abstraction using Theorem Proving and Symbolic Simulation", *6th International Workshop on Higher Order Logic Theorem Proving and Its Applications*, 1993.

[53] J. Roy, N. Kumar, R. Dutta, and R. Vemuri, "DSS: A Distributed High-Level Synthesis System", *IEEE Design and Test of Computers*, 1992.

[54] R. N. S. Devadas, H. T. Ma, "On Verification of Sequential Machines at Differing Levels of Abstraction", *IEEE Transactions on Computer-Aided Design*, 1988.

[55] N. Shankar, S. Owre, and J. M. Rushby, "The PVS Proof Checker: A Reference Manual (Beta Release)", 1993.

[56] N. Shankar, S. Owre, J. Rushby, and D. Stringer-Calvert, "PVS Prover Guide", *SRI International*, September 1999.

[57] M. K. Srivas and S. P. Miller, "Formal Verification of the AAMP5 Microprocessor", *Industrial Applications of Formal Verification.*

[58] L. Stok and R. V. D. Born, "EASY: Multiprocessor Architecture Optimiztion", *Proceedings of International Workshop on Logic and Architecture Synthesis for Silicon Compilers*, 1998.

[59] D. E. Thomas, R. L. Blackburn, and J. V. Rajan, "Linking the Behavioral and Structural Domains of Representation for Digital System Design", *IEEE Transactions on Computer Aided Design*, 1987.

[60] D. Thomas, C. Y. H. III, T. Kowalski, J. Rajan, and A. Walker, "Automatic Data Path Synthesis", *IEEE Transactions on Computers*, 1983.

[61] D. Thomas, E. Lagnese, R. Walker, J. Nestor, J. Rajan, and R. Blackburn, "Algorithmic and Register Transfer Level Synthesis: The System Architect's Workbench", *Kluwer Academic Publishers*, 1990.

[62] C. Tseng and D. P. Siewiorek, "Facet : A Procedure for the Automated Synthesis of Digital Systems", *Proceedings of 20th ACM/IEEE Design Automation Conference*, 1983.

[63] J. Vanhoof, K. V. Rompaey, I. Bolsens, G. Goossens, and H. D. Man, *Hih-Level Synthesis for Real-Time Digital Signal Processing.* Kluwar Academic Publishers, 1993.

[64] R. Vemuri, "On the Notion of Normal Form Register-Level Structures and Its Applications in Design-Space Exploration", *Proceedings of IEEE European Design Automation Conference*, 1990.

[65] R. Vemuri, P. Mamtora, P. Sinha, N. Kumar, and J. R. R. Vutukuru, "Experiences in Functional Validation of a High Level Synthesis System", *Proceedings of 30th ACM/IEEE Design Automation Conference*, 1993.

[66] R. Walker and R. Camposano, "A Survey of High-Level Synthesis Systems", *Kluwer Academic Publishers*, 1990.

[67] P. Windley, "The practical verification of microprocessor designs", *Compcon*, 1991.

[68] P. J. Windley, "Verifying Pipelined Microprocessors", *Proceedings of the International Conference on Computer Hardware Description Languages*, 1995.

[69] N. Woo, "A Global, Dymanic Register Allocation and Binding for Data Path Synthesis System", *Proceedings of the 27th ACM/IEEE Design Automation Conference*, 1990.