Hand-in-hand Verification and Synthesis of Digital Circuits

THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF

Master of Science (by Research)
in
Computer Science and Engineering

By

Chandan Karfa

under the guidance of

Chittaranjan Mandal
Dipankar Sarkar

Department of Computer Science and Engineering
Indian Institute of Technology
Kharagpur

March 2007

Dedicated to my parents

Certificate

This is to certify that the thesis titled ”Hand-in-hand Verification and Synthesis of Digital Circuits” submitted by Chandan Karfa, to the Department of Computer Science and Engineering, Indian Institute of Technology, Kharagpur, India, for the partial fulfilled for award of the degree of Master of Science (by Research), is a bonafide record of original research work carried out by him under our supervision and guidance. The thesis fulfills all the requirements as per the regulations of this Institute and, in our opinion, has reached the standard needed for submission. Neither this thesis nor any part of it has been submitted for any degree elsewhere.

Place: I.I.T. Kharagpur

Date:

(Chittaranjan Mandal)
Associate Professor
Dept. of Computer Science and Engg
Indian Institute of Technology
Kharagpur 721302, INDIA.

Place: I.I.T. Kharagpur

Date:

(Dipankar Sarkar)
Professor
Dept. of Computer Science and Engg
Indian Institute of Technology
Kharagpur 721302, INDIA.

Acknowledgment

This thesis is the result of my research work under the guidance of Prof. C. R. Mandal and Prof. D. Sarkar at the Department of Computer Science and Engineering of the Indian Institute of Technology, Kharagpur. I am deeply grateful to my research advisors for the huge amount of time and effort they spent guiding me through several difficulties on the way. Without the help, encouragement and patient support I received from my guides, this thesis would never have materialized. I am considered myself extremely lucky for getting the opportunity to work under them. During my long association with them and the other professors in IIT, Kharagpur, I am sure, I have learned a lot. I am also thankful to Dr. Chris Reade, Head of School of Business Information Management, Kingston Business School for providing me some useful suggestions and support. I want to thank Prof. A. Patra, Professor-in-charge, Advanced VLSI Design Laboratory, IIT Kharagpur for allowing me to use the lab resource for my experimentations. I will never forget the support, suggestion and ideas provided by Santosh Biswas through out my research work. I am grateful to Murali, Satyam, Srinivas and Ajay for their full sopport in the development of SAST. My special thanks goes to Sumit and Tirthankar for help me in preparing the GUI of SAST. It was a great fun and source of ideas and energy to have friends like Somnath, Bodhisatwa, Harsha, Sayak, Gopal, Soham, Soumyajit, Sunandan, Pratyush, Barun, Plaban, Anupam and others in my department. I am thankful to our cook Das da for serving me food through out my stay in IIT Kgp. Last but most important are my parents and the other members of my family. Without their constant encouragements, supports and well wishes, this thesis never be materialized.

Chandan Karfa

Department of Computer Science and Engineering

Indian Institute of Technology, Kharagpur

March 2007

Abstract

Advances in VLSI technology have enabled its deployment into complex circuits. Synthesis flow of such circuits comprises various phases where each phase performs the task algorithmically providing for ingenious interventions of experts. The gap between the original behaviour and the finally synthesized circuit is too wide to be analyzed by any reasoning mechanism. The validation tasks, therefore, must be planned to go hand-in-hand with each phase of synthesis with scope to handle the specialties of each synthesis sub-task separately. This thesis is concerned with hand-in-hand verification and (high-level) synthesis of digital circuits. The verification of high-level synthesis is performed in three-phases namely, scheduling verification, allocation and binding verification and data-path and controller verification. The input and output of each phase are represented as finite state machines with data-paths (FSMD); the equivalence of two FSMDs is defined and has been proved. The difficulties of each phase are identified and the verification methods based on equivalence of two FSMDs have been formulated accordingly. The scheduling verification method is strong enough to accommodate merging of path segments in the original behaviour, application of several code motion techniques and some arithmetic transformations employed during scheduling. The allocation and binding verification method is capable of handling register sharing verification. For verification of the data-path and controller synthesis phase, a rewriting method is proposed. The method reveals, from a flat set of control signal assertions, the spatial sequences of data flow over the data-path, each sequence realizing a member of a concurrent set of register transfer operations. A high-level synthesis tool, called *structured architecture synthesis tool (SAST)*, has been developed which support hand-in-hand synthesis and verification. The synthesis flow of SAST and the results for several high-level synthesis benchmarks are provided. The thesis concludes by identifying some directions for future research.

Key Words: High-level Synthesis, Formal Verification, Equivalence Checking, Normalization, FSMD models, Path-Based Scheduling, Code Motion Techniques, Allocation and Binding, Register Sharing, Data-Path and Controller Generation, Rewriting Method, SAST.

Contents

1 Introduction, Background and Motivation **1**

1.1 General Introduction 1

1.1.1 High-level Synthesis 3

1.1.2 Formal Verification 5

1.2 Literature Survey on High-level Synthesis Verification 6

1.2.1 Pre-synthesis Verification 7

1.2.2 Formal Synthesis Verification 7

1.2.3 Post-synthesis Verification 8

1.3 Motivation and Objectives of the Present Work 13

1.4 Contributions of the Present Work 15

1.5 Organization of the Thesis 17

2 The Equivalence Problem Formulation **19**

2.1 Introduction 19

2.2 Finite State Machines with Data-paths 19

2.2.1 Paths and Transformations along a Path 23

2.2.2 Computation of R_α and r_α 25

2.2.3 Characterization of Paths 25

2.2.4 Computations and Path Covers of an FSMD 25

2.3 Equivalence of FSMDs 26

2.3.1 The Basic Equivalence Checking Method 28

2.3.2 Normalization of Arithmetic Expressions 30

2.4 Conclusions 33

3 Scheduling Verification **35**

3.1 Introduction . . . 35
3.2 Objective of Scheduling Verification . . . 37
3.3 Verification Issues . . . 37
3.4 The Scheduling Verification Algorithm . . . 42
3.4.1 The Algorithm . . . 42
3.4.2 Correctness of the algorithm . . . 43
3.4.3 Complexity of the algorithm . . . 44
3.5 Verification of Different Scheduling Algorithms . . . 45
3.5.1 Basic Block Based Scheduling . . . 45
3.5.2 Path Based Scheduling . . . 46
3.6 Performance on Several HLS Transformations . . . 48
3.6.1 Renaming . . . 48
3.6.2 Common Sub-Expression Elimination . . . 49
3.6.3 Code Transformation to Increase Conditional Reuse of Hardware . . . 51
3.6.4 Reverse Speculation . . . 52
3.6.5 Early Condition Execution . . . 55
3.6.6 Conditional Speculation . . . 56
3.6.7 Conditional Branch Balancing . . . 57
3.6.8 Speculation . . . 57
3.6.9 Loop Shifting and Compaction . . . 59
3.7 Conclusions . . . 61

4 Allocation and Binding Verification 63
4.1 Introduction . . . 63
4.2 Objectives . . . 65
4.3 Verification of Allocation and Binding of Functional Units . . . 65
4.4 Register Sharing Verification . . . 65
4.4.1 The Mapping Functions . . . 66
4.4.2 Verification Issues . . . 68
4.4.3 Verification Algorithm . . . 69
4.4.4 An Example . . . 69
4.4.5 Performance of the Algorithm . . . 71

4.5 Conclusions 73

5 Data-path and Controller Verification 75

5.1 Introduction 75

5.2 Verification Goal 78

5.3 Construction of the FSMD M_3 78

5.3.1 Construction of the FSMD M_3: An Example 82

5.3.2 A Rewriting Method 84

5.3.3 Correctness and Complexity of Algorithm 2 86

5.4 Verification During Construction of FSMD M_3 89

5.4.1 Redundancy Optimization in the Data-path and in the Controller 90

5.5 Verification by Equivalence Checking 91

5.6 Conclusions 91

6 Development of a High-level Synthesis Tool (SAST) 95

6.1 Introduction 95

6.2 Target Architecture 95

6.3 SAST Synthesis Steps 98

6.3.1 CDFG Generation 99

6.3.2 Preprocessing 102

6.3.3 Scheduling 106

6.3.4 Register Allocation and Binding 114

6.3.5 Data-path and Controller Generation 118

6.3.6 Verilog Code Generation 121

6.4 Generation of the FSMDs for Verification 122

6.4.1 Construction of FSMD from the CDFG 123

6.4.2 Construction of FSMD from the Scheduled Behaviour 123

6.4.3 Construction of FSMD from the Allocation and Binding Results 125

6.4.4 Construction of FSMD from RTL Design 126

6.5 Conclusions 128

7 Experimental Results 129

7.1 Introduction 129

7.2 Synthesis and Verification Results 129
7.2.1 Effects of the Architectural Parameters on Synthesis Results 129
7.2.2 Comparison with other Synthesis Tools 131
7.2.3 Verification vs. Synthesis 132
7.3 Conclusions 133

8 Conclusions and Future Scope of Work **135**
8.1 Summary of the Work 135
8.2 Future Scope of Work 136

A Synthesis with SAST: A Case Study **139**

B Publications out of this work **155**

C Bio-data **157**

Bibliography **159**

List of Figures

1.1 Hierarchy of Synthesis Flow 2
1.2 The steps of high-level synthesis 4
1.3 Hand-in-hand synthesis and verification 14

2.1 The FSMD corresponding to the GCD behaviour given in example 1 22
2.2 A typical path, its condition of execution and its simple data transformation 24
2.3 FSMD designations in the hand-in-hand synthesis and verification flow 30

3.1 Phase-wise modification of an example input behaviour by a basic block based scheduler 36
3.2 Working of the proposed algorithm on an example. (a) M_0: An FSMD before scheduling (b) M_1: Corresponding FSMD after scheduling 40
3.3 Scheduling of a relational operation 41
3.4 A CDFG and its corresponding FSMD structure 46
3.5 The FSMDs of the GCD example (a) M_0: before scheduling (b) M_1: after scheduling using a path-based scheduler 47
3.6 Scheduling using variable renaming technique: An example 48
3.7 Scheduling using elimination of common sub-expressions: An example 50
3.8 Conditional reuse to reduce execution time: An example 52
3.9 Reverse speculation technique: An example 53
3.10 Reverse speculation: A special case 54
3.11 Early Condition Execution: An example 55
3.12 Conditional speculation technique: An example 56
3.13 Conditional branch balancing: An example 57
3.14 Speculation technique: An example 58

3.15 Loop shifting and compaction: An example 60

4.1 An illustration of allocation and binding process: a. Scheduled behaviour b. After allocation and binding 64
4.2 DIFFEQ Example: a. FSMD after scheduling b. FSMD after allocation & binding . 66

5.1 Data-path generation: An example 76
5.2 The structure of the RTL description produced by any HLS tool 77
5.3 Data-path with control signals 77
5.4 The steps of data-path and controller verification 79

6.1 Schematic of structured architecture. 96
6.2 An Architecture Block 97
6.3 SAST synthesis steps 99
6.4 CDFG representation of GCD behaviour 100
6.5 Steps involved in CDFG generation. 102
6.6 Data flow diagram for preprocessor before scheduling 103
6.7 A sample behavioural description 105
6.8 Partial order for the behavioural description given in figure 6.7. 105
6.9 Generating initial attributes of offspring by crossover. 109
6.10 Completion algorithm. 113
6.11 The block-diagram of the controller 120
6.12 Generating Write Enable signals. 121
6.13 Timing diagram for control and write enable signals. 122
6.14 DIFFEQ example to show how the FSMD M_0 is constructed from CDFG 125
6.15 DIFFEQ example: the scheduled behaviour and the FSMD M_1 126

7.1 Graphical User Interface for SAST 130
7.2 Synopsys DA output for EWF. 133
7.3 The typical number of variables and the number of registers required for different HLS benchmarks 134

List of Tables

2.1 Conditions on c_1 and c_2 for which $(s_1 + c_1)R_1 0$ implies $(s_2 + c_2)R_2 0$ 32

4.1 Mapping of the registers to the variables for DIFFEQ example 67

4.2 Computation of data transformation of the path $q_{1,0} \Rightarrow q_{1,11}$ in M_1 71

4.3 Computation of data transformation of the path $q_{2,0} \Rightarrow q_{2,11}$ in M_2 72

5.1 The function f_{mc} from the set $\mathscr{M}$ to the set $\mathscr{A}$ 83

5.2 Construction of the set $\mathscr{M}_A$ from the function f_{mc} for the control assertion pattern $A = \langle 1, 0, 1, 1, 1, 0 \rangle$ 83

6.1 Schedule of operations of the code for DIFFEQ given in figure 6.7. 106

6.2 Crossover of scheduling attributes of operation '3' of figure 6.7 111

7.1 Synthesis results for some HLS benchmarks for different architectural parameters . 131

7.2 Comparison of results with a few other synthesis tools. 132

7.3 Results for different high-level synthesis benchmarks 132

7.4 Scheduling verification results for different high-level synthesis benchmarks . . . 134

Chapter 1

Introduction, Background and Motivation

1.1 General Introduction

Very Large Scale Integrated Circuit (VLSI) technology provides densities of several million gates of random logic per chip. Chips of such complexity are very difficult, if not impossible, to design using the traditional capture and simulate design methodology. Furthermore, VLSI technology has also reached such a maturity level that it is well understood so as to no longer provide a competitive edge by itself. Instead, time to market is usually equally, if not more, important than area or speed. The industry has started looking at the product development cycle comprehensively to reduce the design time and to gain a competitive edge in the time-to-market race. Automation of the entire design process from conceptualization to silicon or a describe-and-synthesize design methodology has become necessary [1].

As the complexity of the chips increases, so does the need for design automation on higher levels of abstraction where functionality is easier to understand and trade-offs are more influential. There are several advantages to automating part or all of the design process and expanding the scope of automation to higher levels. First, automation assures a much shorter design cycle. Secondly, it allows for more exploration of different design styles since different designs can be generated and evaluated quickly. Finally, if synthesis algorithms are well understood, design automation tools may out-perform average human designers in meeting most of the design constraints and requirements.

Synthesis is the process of interconnecting primitive components at a certain level of abstraction (target level) to realize a specification at a higher level of abstraction (source level). Synthesis,

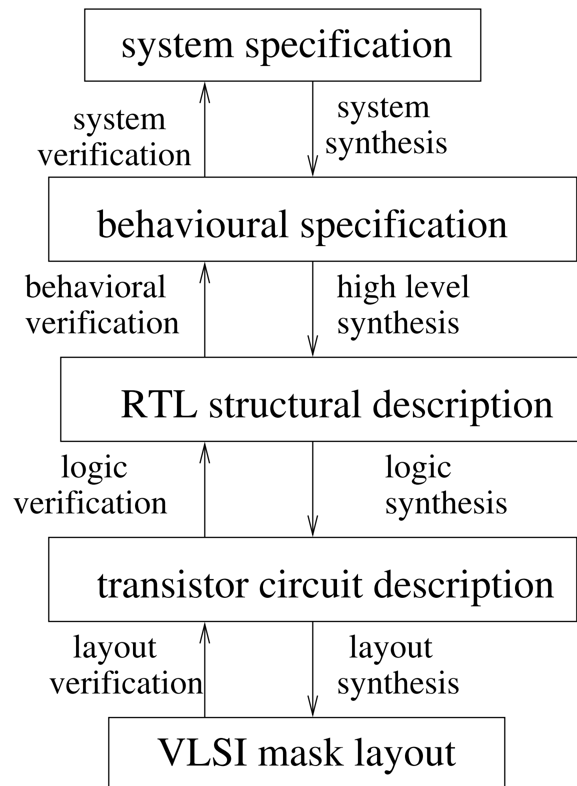


Figure 1.1: Hierarchy of Synthesis Flow

sometimes called design refinement, adds an additional level of detail that provides information needed for the next level of synthesis or for manufacturing the design. This more detailed design must satisfy design constraints supplied along with the original behavioural description or generated by a previous synthesis step. The source and target levels categorize the various synthesis systems. Several levels of the synthesis process is shown in figure 1.1. The system synthesis step takes as input a system specification involving processors and memories and outputs an equivalent functional specification. The high level synthesis (HLS) step takes an algorithmic or high level behaviour as input and outputs the register transfer level (RTL) behaviour consisting of functional units, storage and interconnection units. The logic synthesis step takes Boolean equations as inputs and generates a gate level design after performing logic optimizations of the input. The layout synthesis step takes the gate level specification and outputs the physical layout implementing the gate level specification.

1.1.1 High-level Synthesis

A behavioural description is used as the starting point for HLS. It specifies the behaviour in terms of operations, assignment statements, and control constructs in a hardware description language (HDL) (e.g., VHDL [2] or Verilog [3]). The output from a high-level synthesizer consists of two parts: a data path structure at the register-transfer level (RTL) and a specification of the finite state machine to control the data path. At the RTL level, a data path is composed of three types of components: functional units (e.g., ALUs, multipliers, and shifters), storage units (e.g., registers and memory), and interconnection units (e.g., buses and multiplexors). The finite state machine specifies every set of micro-operations for the data path to be performed during every control step.

In the first step of HLS, the behavioural description is compiled into an internal representation. This process usually includes a series of compiler like optimizations. In addition, it may also apply some hardware-specific transformations such as, syntactic variances minimization, retiming and those exploiting the associativity and commutativity properties of certain operations. A control/data flow graph (CDFG) is a commonly used internal representation to capture the behaviour. The control-flow graph (CFG) of the CDFG captures sequencing, conditional branching and looping constructs in the behavioural description; the data-flow graph (DFG) captures data-manipulation activities described by a set of assignment statements (operations).

The following three steps form the core of transforming a behaviour into an RTL description: scheduling, allocation and binding. Scheduling assigns operations of the behavioural description into control steps. A control step usually corresponds to a cycle of the system clock, the basic time unit of a synchronous digital system. Allocation chooses functional units and storage elements from the component library. There may be several alternatives among which the synthesizer must select the one that matches the design constraints best and maximizes the optimization objective. Binding assigns operations to functional units, variables to storage elements and data transfers to wires or buses such that data can be correctly moved around according to the scheduling.

The final step of high-level synthesis is data-path and controller generation. Depending upon the scheduling and the binding information of the operations and the variables, proper interconnection between the data-path components is set up. Finally, a finite state machine is generated to control all the micro-operations over the date-path. The high-level synthesis steps are depicted in figure 1.2.

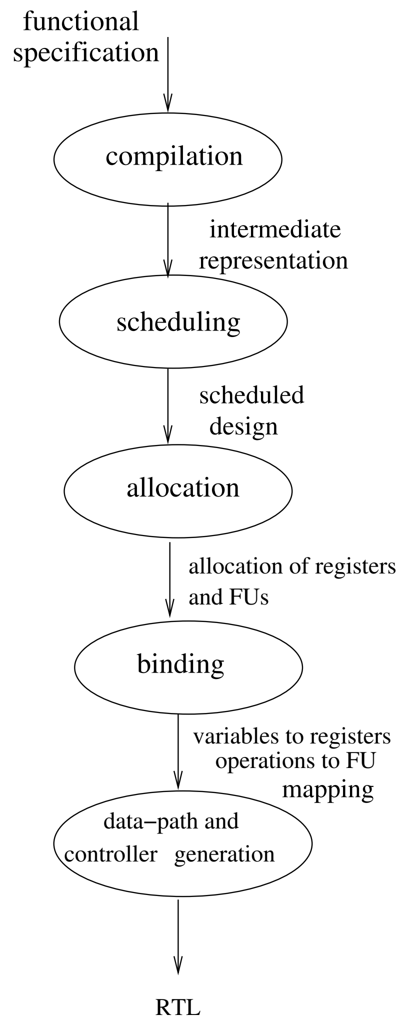


Figure 1.2: The steps of high-level synthesis

Requirements of High-level Synthesis Verification

The use of high-level synthesis systems becomes increasingly crucial to deal with the increasing complexity of today's VLSI designs, shorten the design cycle, search the design space and reduce the design errors. Several HLS systems like Maha [4], Hercules [5], HAL [6], Chippe [7], STAR [8], SAM [9], GABIND [10], SPARK [11] are now available to support the HLS of digital systems. Constant evolution and concurrent changes in the HLS synthesis process and in the component libraries, however, have made the synthesis steps so intricate that neither the synthesis procedures nor the component libraries can be assumed to be correct by construction. Also, there is no one-to-one correspondence between the input and the output of the HLS process. Moreover, there are several optimizations that are performed during the synthesis process, like minimization

of the control steps in scheduling, that of registers and functional units in allocation and binding, optimization of the data-path interconnections in data-path generation, minimization of control signals in the controller generation, etc. It is, therefore, necessary to carry out extensive verification of the RTL design before going to the next level synthesis.

1.1.2 Formal Verification

Formal verification consists in formally establishing a relationship between an implementation and a specification. The fact that this reasoning has to be formal requires that some kind of formalism be used to express all three entities, implementation, specification, and the relationship between them.

An implementation consists of a description of actual hardware design that is to be verified. A specification is a description of the intended/required behaviour of a hardware. Formal verification involves furnishing a proof that an implementation "satisfies" a specification. The notion of satisfaction also has to be formalized, typically in the form of requiring that a certain formal relationship holds between the descriptions of the implementation and the specification. Various notions have been used by the researchers, the semantics for each of these ensuring that the intended satisfaction relation is met. The formal verification techniques can be classified in four categories:

- *Theorem proving*: The 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 that the proof can draw upon. For most of the cases, the logic has to be beyond propositional logic. Thus, we confront the undecidability result ruling out a completely automated prover.

- *Model Checking*: The specification is in the form of logic formula, the truth of which is determined with respect to a semantic model provided by an implementation. This approach is more suitable for property checking whereas the HLS verification is a problem of establishing behavioural equivalence.

- *Language Containment*: The language accepted by the automaton representing an implementation is shown to be in the language accepted by the automaton representing a specification. This approach cannot encompass integers or beyond and hence does not apply to HLS of say, arithmetic circuits.

- *Equivalence Checking*: The equivalence of a specification and an implementation is checked. In the synthesis process, the design representations are transformed from one abstraction level to its lower abstraction level. Naturally, the equivalence checking is a more suitable choice than the three approaches mentioned above for verifying the synthesis process.

Another aspect in formal verification is the model used to represent the specification and the implementation. In this work, we use FSMD (*finite state machine with data-paths*) instead of an FSM (finite state machine) for modeling purpose. The FSMD is a universal specification model [12], that can represent all hardware designs. An FSM (finite state machine) model works well for up to several hundred states. Beyond that, the model becomes incomprehensible to human designers. The designer tackles this problem by resorting to a *control path - data path partitioning* of the circuit in which all the data storage registers and the data transformation / status detection circuits are put in the data path (DP) and the sequential aspects of the behaviour are taken care of in the control path (CP) of the circuit. Starting from the initial state, the control path invokes signals which set up paths for the register transfer operations in the data path as specified in the behavioural specification; the results of these operations are available to the control path through certain status outputs of the data-path; depending upon the states of these lines, the control path assumes the next state. The entire data path state space is partitioned by some data predicates captured by the status output lines. In fact, the data path space influences the control flow of the underlying algorithm only at some time steps; at each of these steps, the data path contains at most 2^n subsets where n is the number of status lines. This notion is captured in the FSMD model. If the data path contains m k-bit registers, then the data space cardinality is 2^{mk}. In practice, n is never more than ten, whereas the product mk can easily be as high as one thousand. These figures clearly indicate the reduction in input state space achieved by resorting to such CD-DP partitioning. The control path, accordingly, can be modeled as an FSM with has much less number of states than what would have been possible had the entire circuit been modeled as an FSM.

1.2 Literature Survey on High-level Synthesis Verification

The correctness of synthesis can be ascertained before, during or after the synthesis process. Hence, synthesis verification can be classified into three categories [13, 14] - *pre-synthesis verification, formal synthesis verification, and post synthesis verification*. In this section, we give a brief survey of literature on each of these categories.

1.2.1 Pre-synthesis Verification

Pre-synthesis verification means proving the correctness of the synthesis procedure, i.e., it has to be proved, that the synthesis program always produces correct synthesis results with respect to the synthesis input. This is to be done by means of software verification techniques [15, 16]. In pre-synthesis verification the correctness is proved once and for all instead of deriving the correctness proof for each synthesis run separately. Software verification, however, is extremely tedious especially, for large sized programs such as the synthesis tools. Therefore, there are only a few works reported where this method is adopted for circuit synthesis verification.

Chapman et al. proposed a pre-synthesis verification technique and employed it in BEDROC HLS tool [15]. Using this tool, part of the high-level synthesis process such as, translation from high-level behavioral description language called Hardware-pal to *Dependence flow graphs*, and some scheduling algorithms were verified.

1.2.2 Formal Synthesis Verification

Formal synthesis means deriving formally the synthesis result within some logical calculus. In conventional synthesis, hardware is represented by arbitrary data structures and there are no restrictions on the transformations on these data structures. In formal synthesis, hardware is represented by means of terms and formulae and only correctness preserving logical transformations are allowed. Restricting synthesis to only correctness preserving logical transformations guarantees the correctness of the synthesis procedure in an implicit manner. In contrast to conventional synthesis, the result is not only some hardware implementation but also a proof of its correctness with respect to the specification.

In [17], a formal synthesis tool, called HASH, is developed and applied in scheduling verification. Each transformation in HASH takes the current design state and the result of some synthesis heuristic and returns a new design state along with the correctness theorem, starting that the old design state is equivalent to (or implied by) the new design state. Exploiting the idea of separating the design space exploration from the formal derivation, the HASH project defines a set of guidelines to formalize both the behaviour and the design decisions taken as higher-order λ-expressions. The system uses theorem prover HOL to decide the soundness of those decisions. Although HASH performs the verification automatically, the formalism used to represent the behaviours and the circuits has limited expressive power and the verification time grows exponentially unless HOL

kernel is modified.

Another formal synthesis system called *FRESH* was built and a new technique for verification was proposed in [18]. In this method, the input behaviour is described by using equational specification (ES) and a set of derivation rules is applied consecutively on the ES. This formal synthesis verification technique is used to check the correctness of scheduling, allocation and binding steps. The distinctive characteristic of these formal synthesis systems is the development of formal design environments where the mathematical representation of a behaviour can be transformed only by the application of sound rules. But the weak relationship between the formal transformations and their underlying hardware concepts forces these systems to be manually driven by the designer, who has to understand the mathematical formalism.

1.2.3 Post-synthesis Verification

Post-synthesis verification is the most frequently used approach today [13]. In post-synthesis verification the synthesis step is first performed in a conventional manner and then the correctness of the synthesis output with respect to the synthesis input is proved. It is independent of the synthesis procedure. The only information available is the synthesis input and the synthesized output. There is no information on how the output was derived from the input. The work, proposed in this thesis, belongs to this category. Several post-synthesis verification techniques of high-level synthesis are reviewed in the following.

Simulation-based Verification Approaches

In simulation based verification, the design to be verified is simulated with a suite of test vectors and the output responses are compared with that of a *golden model*. Ernst et al. proposed a simulation-based verification for high-level synthesis in [19]. Their system, called *Satya*, maps an algorithmic description to a logic circuit description, compares the two descriptions to detect semantic errors and identify the causes of those errors. The authors have used *Satya* to verify the *Bridge* synthesis system [20]. Limitation of this system is that verification is at the level of scheduled specification and in the original behavioural level [21].

In [22], a technique for augmenting simulation based verification at RTL level is reported. In this method, the designer of an RTL circuit embeds a small amount of well understood extra functionality or behaviour into the circuit verification. This extra behaviour is inserted into both

the golden model and the also the circuit under verification. During simulation based verification, this extra behaviour is used along with the existing behaviour of the circuit to exercise the design more throughly. Once the circuit is throughly verified for the functionality, the extra behavioural constructs can be removed to produce the original verified design. Embedding the extra behaviour automatically in the RTL design produced by an HLS, however, is the main bottleneck of this approach.

Another simulation-based methodology, called *Observable time windows (OWT)*, is proposed in [21] for verification of high-level synthesis results. In this work, the notion of equivalence between the the behaviour before scheduling (specification) and the implementation is defined. OWTs correspond to the instants in the simulation run of the implementation which can be directly compared with those in the simulation result of the specification. This approach is implemented in the HIS system reported in [23].

As the complexity of digital systems increases, normal simulation based methods of design validation are becoming impractical due to the large number of internal states [24]. Also, a simulation based approach cannot prove that the result of synthesis is correct as formal verification would do because the former is non-exhaustive.

End-to-end Verification of High-level Synthesis

A set of high-level synthesis systems is validated using formally verified transformations in [25]. This tool examines the output of a high-level synthesis system and derives a sequence of behaviour-preserving (correct) transformations (witness) that leads to the same effect as the applied HLS algorithm. If every transformation, identical in the derived sequence, is applied in the presence of a set of preconditions (which are proved to lead to a correct design), then the resulting RTL design is correct. The approach is as follows. A set of elementary structural transformations is specified and proved for correctness in PVS. Now, given a high-level behaviour, an initial design is derived by interpreting the control and data flow graph (CDFG) to form a unique ALU in the data-path to implement each operation in the behaviour, a unique register to store each value and a unique wire to carry each data flow. The transformations are applied to this initial design. It has a *witness generator* which takes the output of an existing synthesis tool for that given high-level behaviour as input and generates a sequence of elementary transformations which, when applied to the RTL design, achieve the same outcome.

A formal approach to address the correctness of transformations in high-level synthesis is proposed in [26]. This work was part of the SPRITE project at *philips research laboratories*. Both the specification at the behavioural level and the implementation at the RTL level are encoded in SIL [27] in this work. A small set of properties (axioms) corresponding to the SIL graph is asserted to be true. These axioms capture the general notion of refinement of the CDFG used in various synthesis frameworks. Other properties are checked to be true by applying a small number of inference rules on properties that have already been verified. The practicality of the tool, however, was not established as it was not integrated with any HLS CAD tool.

An equivalence checking between behavioural and RTL descriptions with virtual controllers and data-paths is reported in [28]. In order to compare the behavioral and the RTL descriptions in a uniform way, a framework for verification and reasoning of the descriptions based on mapping to virtual datapaths/controllers from these descriptions is developed in this work. Once those mappings have been established, the real comparison can be based on the data transfer analysis controlled by finite state machines; and the verification of arithmetic/logic functionality can be done separately. Virtual datapaths are essentially in net-list form and virtual controllers are represented as finite state machines. There are two key ideas behind this approach. One is to convert equivalence checking on the two descriptions to equivalence checking on their virtual controllers by using the same virtual datapaths for the two descriptions to be checked. The other is to reduce the comparison between two finite state machines to pure topological analysis on the structure of the finite state machines by utilizing equivalence classes on the sets of sub-machines. Normally, equivalence checking between the two descriptions can be made by reachability analysis which is very expensive. In this work, the same virtual data path has been used for the two virtual controllers to be compared thereby easily reducing the problem to the one of the comparison of two virtual controllers only. How the equivalence checking works when the two descriptions are very different and they cannot be mapped to the same data path, however, is not discussed in this work.

In [29], the RTL generated by an HLS tool is verified against its input specification. This technique of determining the correctness of the RTL design depends upon comparing the values of certain critical specification variables in certain critical behavioural states with those of certain critical RTL registers in certain critical controller states, provided they match at the start state. It is accomplished by showing that for each critical path between a pair of critical states, if the critical variables match the critical registers at the originating state, then they will match them at the terminating state. The technique is integrated with DSS [30] high-level synthesis system. This

approach, however, necessitates that the control flow branches in the behaviour specification are preserved and no new control flow branches are introduced. This condition may not hold for the path-based scheduler.

The end-to-end HLS verification techniques are not efficient enough as it is not only error prone but also unable to find the exact sub-task in which the error occurs. Also, the sub-tasks of HLS are interdependent on each other. If an error occurs in the early steps in the synthesis process, it has to be fixed at that moment. Otherwise, it will propagate to the subsequent steps resulting in avoidable wastage of time and longer design cycles.

Verification of the High-level Synthesis Sub-tasks

To overcome the problem discussed above, several approaches are proposed in the literature to verify the sub-tasks of the HLS process. They are typically scheduling verification, allocation and binding verification and RTL verification. Some of the allocation and binding verification methods treat the allocation, binding and the data-path and controller generation steps into one by verifying the final RTL against the scheduled behaviour. In the following, the verification of different sub-tasks of HLS are discussed.

Scheduling Verification

Verification of *As-Soon-As-Possible (ASAP)* scheduling algorithm is reported in [31]. The high-level design description is represented as a state table format (STF) here. The STF semantics are embedded using a library of some inductively defined relations and functions which make up the scheduling algorithm. The relations and the functions are defined using the standard recursive and non-recursive definition mechanisms of HOL [32].

A formal specification and proof of correctness of widely used Force-Directed List Scheduling (FDLS) algorithm for resource-constrained scheduling in high-level synthesis systems is presented in [33]. In this approach, the base specification model for the scheduling task is identified first. Next, the specification model is formalized as a collection of theorems in a higher-order logic theorem proving environment. Finally, the formal description of the algorithm is verified against the base theorems.

The above two approaches, however, are applicable only to a certain class of scheduling algorithms.

A formal verification of scheduling process using finite state machines with data-path (FSMD) is reported in [34]. In this paper, break-points are introduced in both the FSMDs followed by construction of the respective path sets. Each path of one set is then shown to be equivalent to some path of the other set. This approach necessities that the path structure of the input FSMD is not disturbed by the scheduling algorithm in the sense that the respective path sets obtained from the break points are assumed to be bijective. This property, however, does not necessarily hold because the scheduler may merge the segments of the original specification into one segment or distribute operations of a segment over various segments for optimization of time steps.

An automatic verification of scheduling by using symbolic simulation of labeled segments of behavioural descriptions has been proposed in [14]. In this paper, both the inputs to the verifier, namely the specification and the implementation, are represented in the *Language of Labeled Segments (LLS)*. Two labeled segments S_1 and S_2 are bisimilar iff the same data-operations are performed in them and control is transformed to the bisimilar segments. The method described in this paper transforms the original description into one which is bisimilar with the scheduled description.

Most of the methods discussed above are well suited for basic block based scheduling [35, 36] where control structure of the input does not get changed by the scheduler; they fail for path-based scheduling algorithms [37, 38] as well as when some code motion techniques [39, 40, 41, 42] are used by the scheduler.

Allocation and Binding Verification

A compositional model for the functional verification of high-level synthesis is proposed in [43]. The method is applicable to the verification of the final synthesis steps in which both the inputs to the verifier, the specification and the implementation, are encoded as FSMDs. The FSMD corresponding to the implementation comprises two parts, namely the operative part FSMD or the data-path and the controller FSMD. Demonstrating the equivalence of the composed FSMD (functional composition of control and operative part) with the scheduled FSMD accomplishes the functional verification.

A formal methodology for verification of various register allocation schemes was proposed in [44]. The scheduled and the RTL description are encoded as *extended finite state machines (EFSMs)* in this work. The method consists in determining the equivalence of critical states, critical variables and critical paths of two EFSMs. A preliminary version of this work, reported in [29],

which can be integrated with synthesis systems which performs little register optimization. One may, however, encounter an infinite number of execution paths from the initial state during showing the equivalence between two critical states in the presence of loops in the behaviour.

The method in [45] checks the correctness of the register transfer level (RTL) description with respect to the scheduled behaviour. The major contribution of this work is the partitioning of the equivalence checking task into two simpler subtasks, verifying the validity of register sharing and verifying the correctness of synthesis of the RTL interconnection and the controller. The irrelevant portions of the design are automatically abstracted out simplifying the register sharing verification task, the later being performed by a model checker. Verifying the RTL is reduced to a combinational equivalence check; a novel and fast RTL technique for combinational equivalence check instead of using slower global gate level checking is also presented.

The work proposed in [46] handles the high-level verification in two steps: verification of scheduling and verification of allocation and final architecture generation. This paper mainly highlighted the verification of allocation and final architecture generation tasks. The goal is achieved by ensuring that the correct FU has been chosen, the correct functionality of the FU has been chosen, the communication network has been correctly generated to allow the necessary data flow for a specified operation and the control signals have been assigned for each operation of the behaviour. The approach, however, ignores register sharing verification.

1.3 Motivation and Objectives of the Present Work

An end-to-end verification method for HLS is very tough and also inadequate in locating the exact source(s) of errors. A phase-wise verification technique with scope to handle the difficulties of each synthesis sub-task separately is necessary for high-level synthesis because of the following reasons. First, the input behavioural specification is given at a very high abstraction level compared to the abstraction level of the output (RTL); secondly, the complexity of the present day VLSI systems is very high; thirdly, the HLS process consists of several sub-tasks such as, scheduling, allocation, binding, data-path and controller design, etc., which are performed sequentially and each sub-task depends mainly on the result of the previous stages. In contrast to end-to-end verification if the correctness of the intermediate results is verified after every sub-task of the HLS process, then we can also easily find the origin of the errors.

- In this work, the correctness of the HLS process is verified in three phases. Phase-I verifies

the scheduling process. In phase-II, the allocation and binding process is verified against the scheduled behaviour. Phase-III ensures the correctness of the data-path interconnections and the controller. The proposed hand-in-hand synthesis and verification framework is shown in figure 1.3.

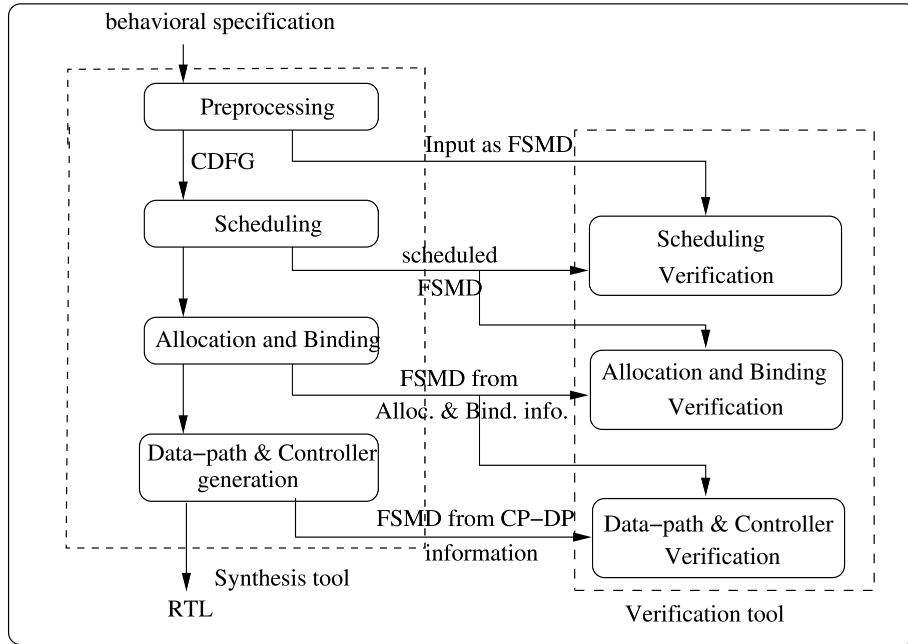


Figure 1.3: Hand-in-hand synthesis and verification

Several works proposed in the literature for verification of different phases of HLS are not strong enough in the sense that they fail in some circumstances which are very common to the modern HLS tools. For example, the proposed methods in the literature for scheduling verification are likely to fail for path-based scheduling algorithms as well as when code motion techniques [47] are used by the scheduler. Hence, more sophisticated verification methodologies are required to cope with these difficulties.

- In this work, an equivalence checking method is proposed which is equipped to handle these difficulties. This method can be applied to the first two phases of HLS verification. The input and output of each of these two phases are encoded as finite state machines with data-paths (FSMDs) and our proposed method finds the equivalence between these two FSMDs.

The verification of the data-path interconnections and the controller behaviour is not an easy task. It consists in analyzing the data-path to find a *spatial sequence* of concurrent micro-operations

needed to accomplish a register transfer (RT) operation and ensuring that the controller generates correct control signals in each control step to perform the required RT operations in the data-path. To the best of our knowledge, no work has been reported in the literature for handling these tasks separately.

- In this work, a state based equivalence checking method is proposed for this phase to accomplish all these tasks. The method uses a *rewriting procedure* to find the spatial sequence of the concurrent micro-operations to establish that the control assertion pattern in each control step is correct.

1.4 Contributions of the Present Work

- **Equivalence problem formulation**

  A formal method for checking equivalence between two FSMDs is formulated. The method consists in introducing cutpoints in one FSMD, visualizing its computations as concatenations of paths from cutpoints to cutpoints and finally, identifying equivalent finite path segments in the other FSMD; the process is then repeated with the FSMDs interchanged.

- **Normalization of arithmetic expression over integers**

  Finding equivalence between two paths involves checking equivalence between two sets of arithmetic expressions. Hence, checking equivalence of two paths reduces to the validity problem of first-order logic which is undecidable; thus, a canonical form does not exist for integer arithmetic. Instead, in this work we adapt a normal form [48] for the arithmetic expressions over integers. The normalization process renders many computationally equivalent formulas syntactically identical as it forces all the formulas to follow a uniform structure.

- **Hand-in-hand verification of high-level synthesis**

  Our goal is to find equivalence between the behavioural description given as input to any HLS tool and the RTL output of that HLS tool. This verification goal is achieved in three phases as discussed in subsection 1.3 and the verification process needs to proceed hand-in-hand with the synthesis process. The hand-in-hand synthesis and verification framework is depicted in figure 1.3. An overview of these three phases of verification is as follows:

Scheduling verification: The goal of the scheduling process is to optimize the number of control steps required to execute all the operations in the input behaviour meeting all the constraints regarding the number of control steps, the delay, the power and the hardware resources. A *path-based scheduler* [37] [38] may modify the control structure of the input behaviour as it tries to merge some consecutive path segments of the input behaviour. Also, use of several code motion techniques like *speculation, reverse speculation, early condition execution, branch balancing, common sub-expression elimination, loop shifting, renaming,* etc, [47, 49, 39, 41], in the scheduling process leads to different transformations in the input behaviour of HLS. The goal of the scheduling verification process is to ensure that the scheduling process preserves the input behaviour, irrespective of the scheduling technique used. The FSMD equivalence checking method proposed in this work meets the requirements for this phase of verification.

Allocation and binding verification: The number of functional units (FUs) selected for performing the operations is less than or equal to the number of operations. Similarly, the number of registers used to store the variables of the behaviour is less than or equal to the number of variables. It is required to ensure that enough number of FUs and registers are allocated and their binding with the operations and variables, respectively, are also proper. The verification of this phase consists of two tasks: (i) verification of the functional unit allocation and binding, and (ii) verification of register sharing among the behavioural variables. Our proposed equivalence checking method has been shown to be applicable to register sharing verification.

Data-path and controller verification: The objective of the data-path generation is to maximize sharing of interconnections and thus minimize the interconnection cost avoiding conflict during data transfers required by the register transfer (RT) operations. Similarly, minimum number of control signals are used by the controller to control all the data transfers among the data-path elements. The controller, represented as an FSM, assigns a value to each control signal in each control step to execute all the required data-transfers and proper operations in the FUs. The verification task in this phase is to ensure the correctness of the data-path interconnections and the controller. The verification involves the following tasks: (i) construction of an FSMD from the data-path and the controller FSM and (ii) showing equivalence between the input FSMD and the output FSMD.

- **Development of a high-level synthesis tool**

  We have developed the above hand-in-hand synthesis and verification framework, called *structured architecture synthesis tool (SAST)*. It takes a synthesizable behavioural description and produces an RTL code in Verilog. SAST is an interconnection aware high-level synthesis tool as it produces a *structured data-path (structured architecture)* by avoiding random interconnections among the data-path components. The tool provides required information for verification from its intermediary synthesis results and supports the hand-in-hand synthesis and verification. Finally, it produces a *correct* RTL behaviour in Verilog.

1.5 Organization of the Thesis

The rest of the thesis is organized as follows.

Chapter 2: In this chapter, the equivalence problem of FSMDs has been formulated, the basic method has been devised and the correctness of the method has been proved. A normal form for the expressions and several simplifications that have been carried out on the normalized expressions are discussed next.

Chapter 3: This chapter discusses the scheduling verification phase. The objective of the scheduling verification is identified first. The required modifications on the basic equivalence checking method are given. An algorithm for verifying the correctness of the scheduling result is proposed and its correctness and complexity have been analyzed subsequently. The performance of this algorithm for different scheduling algorithms and on various code motion techniques are discussed next. Some upgradations of the algorithm, needed to handle the above tasks, are also proposed in this chapter.

Chapter 4: This chapter discusses the allocation and binding verification phase. The objectives of the this phase of verification is enlisted. The verification of FU allocation and binding is given next. It is followed by the register sharing verification. Some mapping functions are defined and also the modification required on definition of equivalence of paths is discussed. A register sharing verification method is proposed next. The working of the algorithm is discussed with an example. The performance of this method for different register optimization schemes as well as for various nature of the input specification have been provided.

Chapter 5: This chapter discusses the verification of data-path and controller synthesis. The objectives of this phase are identified first. The construction of the FSMD from the control-path and

data-path information is discussed. A rewriting algorithm involved in the construction mechanism is given next. Termination, soundness and completeness of the method have been proved and the complexity of the method is also analyzed. The construction process is described with an illustrative example. The verification of the data-path and the controller during FSMD construction is given next. Finally, a state-based verification method for verifying the functionality of the RTL behaviour is proposed.

Chapter 6: This chapter discusses the proposed target data-path, the details of the synthesis phases of SAST and the method for generating the FSMD from the input and the output of each synthesis phase.

Chapter 7: This chapter provides some synthesis and verification results.

Chapter 8: The chapter contains some concluding remarks and identifies some future research directions.

Chapter 2

The Equivalence Problem Formulation

2.1 Introduction

This chapter is concerned with the theoretical issues behind our proposed verification methodologies of different phases of high-level synthesis. The finite state machine with data-path (FSMD) proposed in [12] has been adapted for design representation in this work. The notion of paths in FSMD, the transformations along a path and the computation and path cover of an FSMD are defined. The equivalence of two FSMDs has been derived and proved. The basic method for checking equivalence of two FSMDs is also provided. The normal form of the expression and the simplifications over the normalized expressions are also discussed in this chapter.

2.2 Finite State Machines with Data-paths

An FSMD (*finite state machine with data-path*) is a universal specification model, proposed by Gajski et al. in [12], which can represent all hardware designs. The model is used in the present work with the addition of a reset state, for encoding the designs to be verified. This reset state is also called the start state of the FSMD. The FSMD is defined as an ordered tuple $\langle Q, q_0, I, V, O, f, h \rangle$, where

1. $Q = \{q_0, q_1, q_2, \ldots, q_n\}$ is the finite set of control states,

2. $q_0 \in Q$ is the reset state,

3. I is the set of primary input signals,

4. V is the set of storage variables,

5. O is the set of primary output signals,

6. $f : Q \times 2^S \rightarrow Q$, is the state transition function and

7. $h : Q \times 2^S \rightarrow U$, is the update function of the output and the storage variables, where S and U are as defined below.

   (a) $S = \{L \cup E_R \mid L$ is the set of Boolean literals of the form b or $\neg b$, $b \in B \subseteq V$ is a Boolean variable and $E_R = \{e\ R\ 0 \mid\ e \in E_A\}\}$; it represents a set of status expressions over $I \cup V$, where E_A represents a set of arithmetic expressions over the set $I \cup V$ of input and storage variables and R is any arithmetic relation. $R \in \{==,\ \neq,\ >,\ \geq,\ <,\ \leq\}$.

   (b) $U = \{x \Leftarrow e \mid x \in O \cup V$ and $e \in E_A \cup E_R\}$ represents a set of storage or output assignments.

Conjunction is assumed to be the implicit connective among the relational expressions belonging to 2^S. Parallel edges between two states capture disjunction of status expressions. Thus, the next (control and data) state and the output depend not only on the present state and the input signals but also on the conjunction of the status expressions that indicate whether a predicate holds on the data state of the storage and the input variables. An FSMD is inherently deterministic, that is, the state transition function f satisfies the property $s_j = s_k \Rightarrow f(q_i,\ s_j) = f(q_i,\ s_k)$. It may be noted that we have not introduced final states in the FSMD model as we assume that a system works in an infinite outer loop.

The behavioural description of the *extended euclidean algorithm for finding the greatest common divisor (GCD) of two integer numbers* [50] is given below and its FSMD is shown in the figure 2.1.

Example 1 `Behavioural description of GCD is:`

```
begin
main_process: process
Input: P0, P1;
Output: yout;
Variable : res, y1, y2 : integer;
begin
```

```
    mainloop: loop
     y1 ⇐ P0;
     y2 ⇐ P1;
     res ⇐ 1;
     while(!(y1 == y2)) loop
          if(even(y1)) then
           if(even(y2)) then
            begin
                res ⇐ res*2;
                y1 ⇐ y1/2 ;
                y2 ⇐ y2/2 ;
            end
            else
                y1 ⇐ y1/2;
          else if(even(y2)) then
                   y2 ⇐ y2/2;
               else if(y1 > y2) then
                    y1 ⇐ y1 − y2;
                    else
                    y2 ⇐ y2 − y1;
          end if;
     end loop;
     res ⇐ res*y1;
     yout ⇐ res;
   end loop mainloop;
  end process main_process;
end behv;
```

□

Example 2 *The FSMD model M_0 for the behavioural specification of* GCD *example is given below and depicted in figure 2.1.*

- M_0=$\langle Q, q_0, I, V, O, f, h\rangle$, *where*

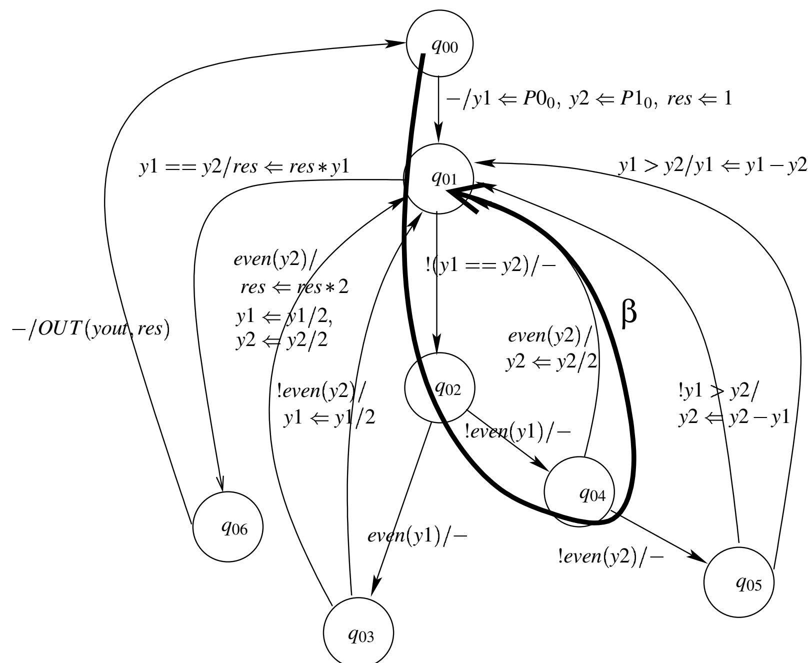


Figure 2.1: The FSMD corresponding to the GCD behaviour given in example 1

- $Q = \{q_{00},\ q_{01},\ q_{02},\ q_{03},\ q_{04},\ q_{05},\ q_{06}\}$,
- $q_0 = q_{00}$,
- $V = \{res,\ y1,\ y2\}$,
- $I = \{P0_0,\ P1_0\}$,
- $O = \{yout\}$,
- $U = \{y1 \Leftarrow P0_0,\ y2 \Leftarrow P1_0,\ res \Leftarrow res * 2,\ y1 \Leftarrow y1/2,\ y2 \Leftarrow y2/2,\ y1 \Leftarrow y1 - y2,\ y2 \Leftarrow y2 - y1\}$,
- $S = \{even(y1),\ even(y2),\ y1 > y2\}$, *where even(y) indicates y mod* $2 = 0$,
- *f and h are defined as in the transition graph shown in figure 2.1.*
- *Some typical values of f and h are as follows:*
  - $f(q_{00},\ true) = q_{01}$,
  - $f(q_{05},\ y1 > y2) = q_{02}$,

- $h(q_{05},\ y1 > y2) = \{y1 \Leftarrow y1 - y2\}$,
- $h(q_{03},\ even(y2)) = \{res \Leftarrow res * 2\}$.

□

2.2.1 Paths and Transformations along a Path

Definition 1 *Path* α *from* q_i *to* q_j*:*

A (finite) path α *from* q_i *to* q_j*, where* $q_i, q_j \in Q$*, is a finite transition sequence of states of the form* $\langle q_i = q_1 \xrightarrow[c_1]{} q_2 \xrightarrow[c_2]{} \cdots \xrightarrow[c_{n-1}]{} q_n = q_j \rangle$ *such that* $\forall l, 1 \leq l \leq n-1, \exists c_l \in 2^S$ *such that* $f(q_l, c_l) = q_{l+1}$*, and* q_k*,* $1 \leq k \leq n-1$*, are all distinct. The end state of the path, i.e.,* q_n*, may be identical to any state* q_k*,* $i \leq k \leq n-1$*, along the path.*

A path, say β, is indicated by the bold arrows in figure 2.1. Here, β is $q_{00} \rightarrow q_{01} \xrightarrow[!(y1==y2)]{} q_{02} \xrightarrow[!(even(y1)]{} q_{04} \xrightarrow[even(y2)]{} q_{01}$. Since there may be more than one transition from a state q_i to a state q_j (with different conditions c_l associated with them), a sequence of states alone does not uniquely characterize a path and has been used only when there is no ambiguity. The state sequence $q_{01} \rightarrow q_{02} \rightarrow q_{03} \rightarrow q_{01}$ in figure 2.1, for example, does not uniquely represent a path as there is more than one transition between the two states q_{03} and q_{01}. Whereas, the state sequence $q_{00} \rightarrow q_{01} \rightarrow q_{02} \rightarrow q_{03}$ in this figure represents a path uniquely.

Definition 2 *The condition of execution of a path* α *(*R_α*):*

Let $\alpha = \langle q_{l_0} \xrightarrow[c_0]{} q_{l_1} \xrightarrow[c_1]{} q_{l_2} \cdots \xrightarrow[c_{k-1}]{} q_{l_k} \rangle$ *be a path. The* condition of execution R_α of the path α *is a logical expression over* $I \cup V$ *such that* R_α *is satisfied by the (initial) data state at* q_{l_0} *iff the path* α *is traversed.*

Thus, R_α is the weakest precondition of the path α [51].
For example, the condition of execution R_β of the path β of figure 2.1 is $!(P0_0 == P1_0) \wedge !even(P0_0) \wedge even(P0_0)$. It is assumed that the inputs and the outputs occur through named ports. The ith input from port P is a value represented as P_i. Thus, if some variable v stores an input from port P (for the ith time along a path), it is equivalent to the assignment $v \Leftarrow P_i$. The variable '$y1$' in the GCD algorithm of example 1 is updated by the 0th input from the port $P0$. This is shown as $y1 \Leftarrow P0_0$ in the corresponding FSMD in figure 2.1. Similarly, the variable '$y2$' stores the 0th input from the port $P1$ and is shown accordingly as $y2 \Leftarrow P1_0$ in the FSMD in figure 2.1. In essence, P_i's comprise the input variable set I.

Definition 3 *The simple data transformation of a path* α *over* V (s_α)*:*

It is an ordered tuple $\langle e_i \rangle$ *of algebraic expressions over the variables in* V *and the inputs in* I *such that the expression* e_i *represents the value of the variable* v_i *after the execution of the path in terms of the initial data state (i.e., the values of the variables at the initial control state) of the path.*

The data transformation s_β of the path β in figure 2.1 is $\langle P0_0,\ P1_0/2,\ 1\rangle$, where the order of the variables is $y1 \prec y2 \prec res$.
The above definition does not take into account the outputs that may occur in a path. The output of an expression e to a port P is represented as $OUT(P,\ e)$ and put as a member of a list preserved for each path. Taking into account the outputs that may occur in a path, the data transformation of a path may be defined as follows.

Definition 4 The data transformation r_α of a path α *over* V *is the ordered pair* $\langle s_\alpha, O_\alpha \rangle$*, where* s_α *is the simple data transformation of* α *and the output list* $O_\alpha = [OUT(P_{i_1}, e_1),\ OUT(P_{i_2}, e_2), \ldots]$*. For every expression e output to port P along the path* α*, there is a member of the form* $OUT(P, e)$ *in the list, in the order in which the outputs occurred.*

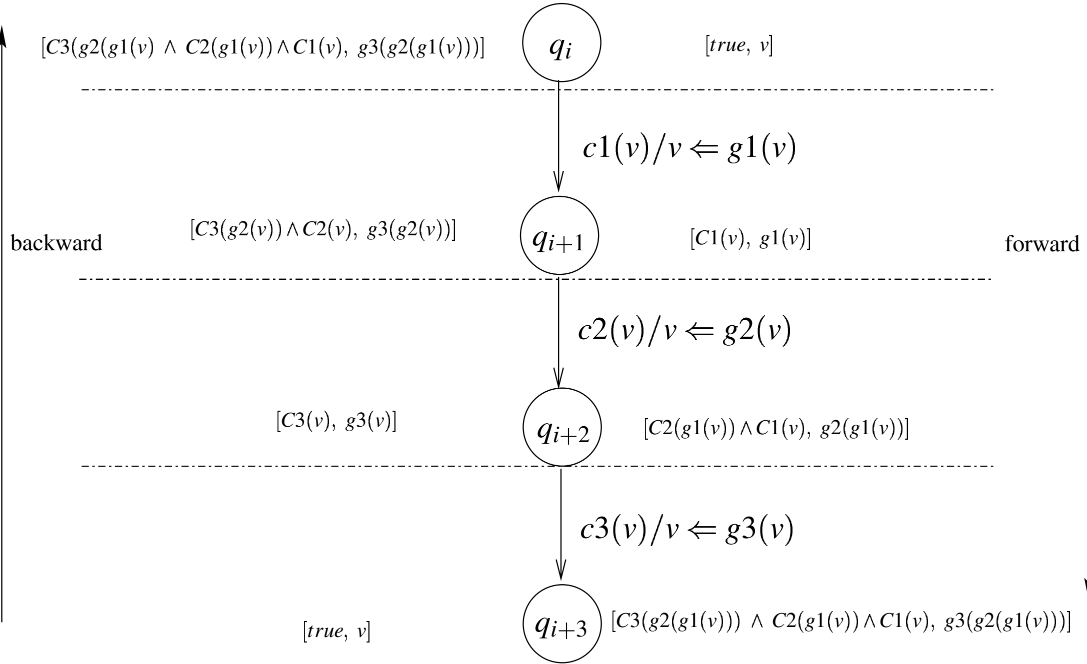


Figure 2.2: A typical path, its condition of execution and its simple data transformation

2.2.2 Computation of R_α and r_α

Computation of the condition of execution R_α can be by *backward* substitution or by *forward* substitution. The former is based on the following rule: If a predicate $c(y)$ is true after assignment $y \leftarrow g(y)$, then the predicate $c(g(y))$ must have been true before the assignment [52]. The transformation s_α is found indirectly using the same principle. The forward substitution method of finding R_α is based on symbolic execution. The ordered pairs at various points in figure 2.2 represents the values of (R_α, s_α) at that point.

2.2.3 Characterization of Paths

The characteristic formula $\tau_\alpha(\overline{v}, \overline{v}_f, O)$ of the path α is $R_\alpha(\overline{v}) \wedge (\overline{v}_f = s_\alpha(\overline{v})) \wedge (O = O_\alpha(\overline{v}))$, where s_α is the data transformation and O_α is the output list in the path α, $\overline{v}$ represents a vector of values of the variables of $I \cup V$, $\overline{v}_f$ represents a vector of values of the variables of V. The formula captures the following: if the condition of execution R_α of the path α is satisfied by the (initial) vector $\overline{v}$ at the beginning of the path, then the path is executed and after execution, the final vector $\overline{v}_f$ of variable values becomes $s_\alpha(\overline{v}_f)$ and the output $O_\alpha(\overline{v})$ is produced.

Let $\tau_\alpha(\overline{v}, \overline{v}_f, O) : R_\alpha(\overline{v}) \wedge (\overline{v}_f = s_\alpha(\overline{v})) \wedge (O = O_\alpha(\overline{v}))$ be the characteristic formula of the path α and $\tau_\beta(\overline{v}, \overline{v}_f, O) : R_\beta(\overline{v}) \wedge (\overline{v}_f = s_\beta(\overline{v})) \wedge (O = O_\beta(\overline{v}))$ be the characteristic formula of the path β. The characteristic formula for the concatenated path $\alpha\beta$ is $\tau_{\alpha\beta}(\overline{v}, \overline{v}_f, O) = \exists \overline{v}_\alpha \exists O_1 \exists O_2 (\tau_\alpha(\overline{v}, \overline{v}_\alpha, O_1) \wedge \tau_\beta(\overline{v}_\alpha, \overline{v}_f, O_2)) = R_\alpha(\overline{v}) \wedge R_\beta(s_\alpha(\overline{v})) \wedge (\overline{v}_f = s_\beta(s_\alpha(\overline{v}))) \wedge (O = O_\alpha(\overline{v}) O_\beta(s_\alpha(\overline{v})))$. O is the concatenated output list of $O_\alpha(\overline{v})$ and $O_\beta(s_\alpha(\overline{v}))$.

2.2.4 Computations and Path Covers of an FSMD

A computation of an FSMD is a finite walk from the reset state q_0 back to itself without having any intermediary occurrence of q_0. Such a computational semantics of an FSMD is based on the assumption that a revisit of the reset state means the beginning of a new computation and each computation terminates. In other words, the behavioural representation has a non-terminating outermost loop from the reset state and each inner loop has a state from which there is a transition out of the loop. A computation μ of an FSMD M may be characterized as $\tau_\mu(\overline{v}_i, \overline{v}_f, O) : R_\mu(\overline{v}_i) \wedge (\overline{v}_f = s_\mu(\overline{v}_i)) \wedge (O = O_\mu(\overline{v}_i))$, where $\overline{v}_i$ is the vector of initial input with which the computation is started, R_μ is a satisfiable condition over the domain of I, s_μ is a function over this domain to the codomain of values over V and O_μ is the concatenation of the output lists resulting from output

operations along μ.

Definition 5 *Two computations μ_1 and μ_2 having the characteristic formulae τ_{μ_1} and τ_{μ_2}, respectively, are said to be equivalent if $R_{\mu_1} = R_{\mu_2}$ and $r_{\mu_1} = r_{\mu_2}$.*

The computational equivalence of two computations μ_1 and μ_2 is denoted as $\mu_1 \simeq \mu_2$. The computational equivalence of two paths p_1 and p_2 can be defined in a similar manner and is denoted as $p_1 \simeq p_2$. Equivalence checking of paths, therefore, consists in establishing the computational equivalence of the respective conditions of execution and the respective data transformations.

Any computation μ of an FSMD M can be looked upon as a computation along some concatenated path $[\alpha_1\alpha_2\alpha_3...\alpha_k]$ of M such that the path α_1 emanates from and the path α_k terminates in the reset state q_0 of M, for $1 \leq i \leq k$, α_i terminates in the initial state of the path α_{i+1} and α_i's may not be all distinct. If $\tau_{\alpha_i}(\overline{v}_i, \overline{v}_{i+1}), 1 \leq i \leq k$, be the characteristic formula of the path α_i, then the characteristic formula of the concatenated path $[\alpha_1\alpha_2\alpha_3...\alpha_k]$ is the formula
$R_{\alpha_1}(\overline{v}_1) \wedge R_{\alpha_2}(s_{\alpha_1}(\overline{v}_1)) \wedge \cdots \wedge R_{\alpha_k}(s_{\alpha_{k-1}}(\cdots(s_{\alpha_1}(\overline{v}_1))\cdots)) \wedge [\overline{v}_{k+1} = s_{\alpha_k}(s_{\alpha_{k-1}}(\cdots(s_{\alpha_1}(\overline{v}_1))\cdots))]$
$\wedge O = O_{\alpha_1}(\overline{v}_1)O_{\alpha_2}(s_{\alpha_1}(\overline{v}_1))\cdots O_{\alpha_k}(s_{\alpha_{k-1}}(\cdots(s_{\alpha_1}(\overline{v}_1))\cdots)$,
where $\overline{v}_i$ and $\overline{v}_{i+1}$ represent the vectors of inputs and the data variables before and after the path α_i, $1 \leq i \leq k$. Recalling that the characteristic formula of μ is $R_\mu(\overline{v}_1) \wedge [\overline{v}_{k+1} = s_\mu(\overline{v}_1) \wedge O = O_\mu(\overline{v}_1)]$, clearly
$R_\mu(\overline{v}_1) = R_{\alpha_1}(\overline{v}_1) \wedge R_{\alpha_2}(s_{\alpha_1}(\overline{v}_1)) \wedge \ \cdots \ \wedge R_{\alpha_k}(s_{\alpha_{k-1}}...(s_{\alpha_1}(\overline{v}_1))...))$,
$s_\mu(\overline{v}_1) = s_{\alpha_k}(s_{\alpha_{k-1}}(...(s_{\alpha_1}(\overline{v}_1))...))$ and
$O_\mu = O_{\alpha_1}(\overline{v}_1)O_{\alpha_2}(s_{\alpha_1}(\overline{v}_1))\cdots O_{\alpha_k}(s_{\alpha_{k-1}}(\cdots(s_{\alpha_1}(\overline{v}_1))\cdots)$.

Definition 6 **Path cover of an FSMD:** *A finite set of paths $P = \{p_0, p_1, p_2, \ldots, \ p_k\}$ is said to cover an FSMD M if any computation μ of M can be looked upon as a concatenation of paths from P. P is said to be a* path cover *of the FSMD M.*

2.3 Equivalence of FSMDs

Let $M_0 = \langle Q_0, q_{00}, I, V_0, O, f_0, h_0 \rangle$ and $M_1 = \langle Q_1, q_{10}, I, V_1, O, f_1, h_1 \rangle$ be the two FSMD representations corresponding to the input and the output of any phase of high-level synthesis. Our main goal is to verify whether M_0 behaves exactly as M_1. This means that for all possible input sequences, M_0 and M_1 produce the same sequences of output values and eventually, when the respective reset states are re-visited, they are visited with the same storage element values. In other words,

for every computation from the reset state back to itself of one FSMD, there exists an equivalent computation from the reset state back to itself in the other FSMD and vice-versa.

Definition 7 *An FSMD M_0 is said to be contained in an FSMD M_1, symbolically $M_0 \sqsubseteq M_1$, if for any computation μ_0 of M_0, there exists a computation μ_1 of M_1 such that $\mu_0 \simeq \mu_1$.*

Definition 8 *Two FSMDs M_0 and M_1 are said to be computationally equivalent, if $M_0 \sqsubseteq M_1$ and $M_1 \sqsubseteq M_0$.*

An FSMD may contain an infinite number of computations. So, it is not feasible to enumerate all possible computations in one FSMD and find their equivalent computations in the other FSMD. To overcome this problem, the following theorem is deduced.

Theorem 1 *For any two FSMDs M_0 and M_1, $M_0 \sqsubseteq M_1$, if there exists a finite cover $P_0 = \{p_{00}, p_{01}, \ldots, p_{0l}\}$ of M_0 for which there exists a set $P_1^0 = \{p_{10}^0,\ p_{11}^0, \ldots,\ p_{1l}^0\}$ of paths of M_1 such that $p_{0i} \simeq p_{1i}^0$, $0 \leq i \leq l$.*

Proof 1 *$M_0 \sqsubseteq M_1$, if for any computation μ_0 of M_0, there exists a computation μ_1 of M_1 such that μ_0 and μ_1 are computationally equivalent [by definition 7].*

Now, let there exist a finite cover $P_0 = \{p_{00}, p_{01}, \cdots, p_{0l}\}$ of M_0. Corresponding to P_0, let a set $P_1^0 = \{p_{10}^0,\ p_{11}^0,\ \cdots,\ p_{1l}^0\}$ of paths of M_1 exist such that $p_{0i} \simeq p_{1i}^0$, $0 \leq i \leq l$.

Since P_0 covers M_0, any computation μ_0 of M_0 can be looked upon as a concatenated path $[p_{0i_1} p_{0i_2} \cdots p_{0i_n}]$ from P_0 starting from the reset state q_{00} and ending again at this reset state of M_0. From the above hypothesis, it follows that there exists a sequence Π_1 of paths $[p_{1j_1}^0 p_{1j_2}^0 \cdots p_{1j_n}^0]$ of P_1^0 where $p_{0i_k} \simeq p_{1j_k}^0$, $1 \leq k \leq n$. So, in order that Π_1 represents a computation of M_1, it is required to prove that Π_1 is a concatenated path of M_1 from its reset state q_{10} to the reset state. The following definition is in order.

Definition 9 **Corresponding states:** *Let $M_0 = \langle Q_0,\ q_{00},\ I,\ V_0,\ O, f_0, h_0\rangle$ and $M_1 = \langle Q_1,\ q_{10},\ I,\ V_1,\ O, f_1, h_1\rangle$ be the two FSMDs having identical input and output sets, I and O, respectively, and $q_{0i}, q_{0k} \in Q_0$ and $q_{1j}, q_{1l} \in Q_1$.*

- *The respective reset states q_{00}, q_{10} are corresponding states.*
- *If $q_{0i} \in Q_0$ and $q_{1j} \in Q_1$ are corresponding states and there exist $q_{0k} \in Q_0$ and $q_{1l} \in Q_1$ such that, for some path α from q_{0i} to q_{0k} in M_0, there exists a path β from q_{1j} to q_{1l} in M_1 such that $\alpha \simeq \beta$, then q_{0k} and q_{1l} are corresponding states.*

Now, let $p_{0i_1} : [q_{00} \Rightarrow q_{0f_1}]$*. Since* $p^0_{1j_1} \simeq p_{0i_1}$*, from the above definition of corresponding states,* $p^0_{1j_1}$ *must be of the form* $[q_{10} \Rightarrow q_{1f_1}]$*, where* $\langle q_{00}, q_{10}\rangle$ *and* $\langle q_{0f_1}, q_{1f_1}\rangle$ *are corresponding states. Thus, by repetitive application of the above argument it follows that if* $p_{0i_1} : [q_{00} \Rightarrow q_{0f_1}]$*,* $p_{0i_2} : [q_{0f_1} \Rightarrow q_{0f_2}], \cdots, p_{0i_n} : [q_{0f_{n-1}} \Rightarrow q_{0f_n} = q_{00}]$*, then* $p^0_{1i_1} : [q_{10} \Rightarrow q_{1f_1}]$*,* $p^0_{1i_2} : [q_{1f_1} \Rightarrow q_{1f_2}], \cdots, p^0_{1i_n} : [q_{1f_{n-1}} \Rightarrow q_{1f_n} = q_{10}]$*, where* $\langle q_{0f_m}, q_{1f_m}\rangle$*,* $1 \leq m \leq n$*, are pairs of corresponding states. Hence,* Π_1 *is a concatenated path representing a computation of* M_1*.*

2.3.1 The Basic Equivalence Checking Method

Theorem 1 suggests a verification method for checking equivalence of two FSMDs which consists of the following steps:

1. Construct the set P_0 of paths of M_0 so that P_0 covers M_0. Let $P_0 = \{p_{00}, p_{01}, \cdots, p_{0k}\}$.
2. Show that $\forall p_{0i} \in P_0$, *there exists a path* p_{1j} of M_1 such that $p_{0i} \simeq p_{1j}$.
3. Repeat steps 1 and 2 with M_0 and M_1 interchanged.

Because of loops it is difficult to find a path cover of the FSMD comprising only finite paths. So any computation is split into paths by putting *cutpoints* at various places in the FSMD so that each loop is cut in at least one cutpoint. The set of all paths from a cutpoint to another cutpoint without having any intermediary cutpoint is a path cover of the FSMD. The method of decomposing an FSMD by putting cutpoints is identical to the Floyd-Hoare's method of program verification [53, 54, 55]. Choice of cutpoints, however, is non-unique and it is not guaranteed that a path cover of one FSMD obtained from any choice of cutpoints in itself will have the corresponding set of equivalent paths for other FSMD. Therefore, it will be necessary to search for a suitable choice of cutpoints. The question remains whether such a choice can be algorithmically hit upon, the answer to which is no [52].

The equivalence problem of FSMDs (EPFSMD) is the same as the equivalence problem of flowchart schemas [56, 52] which is undecidable and not even partially decidable [52]. However, since the final targeted hardware has only a finite data-path, the restricted problem can be reduced to the equivalence problem of finite state machine models (EPFSM) which is decidable. Unfortunately, an FSMD with an n-bit data-path results in a number of states of the order of 2^{kn}, where k is the number of storage elements of n bits. The value of kn easily exceeds several hundreds. Thus, deciding EPFSMD with a finite data-path by reducing them to EPFSM is of little use in

practice. On the other hand, specialized analytical treatments, such as the work described here, may aid in revealing problems in the working of the algorithm which may never use the finiteness in producing the output which is to be checked. In this case, the equivalence checking algorithm would identify paths that are not matched, which could be particularly helpful in fixing the errors in different phases of the high-level synthesis process. This benefit would normally be lost by reducing a finite EPFSMD to EPFSM.

Therefore, we devise a good strategy for setting the cutpoints which would work for many cases but not for all cases. We choose the cutpoints in any FSMD as follows.

1. The reset state is chosen.
2. A state q_i is a cutpoint if there is a divergence of flow from q_i. More formally, q_i is a cutpoint if $\exists c_1, c_2 \in S$ such that $c_1 \neq c_2$ *and* $\langle q_i, c_1, q_j \rangle \in f$ *and* $\langle q_i, c_2, q_l \rangle \in f$ and q_j, q_l are not necessarily distinct.

Obviously, cutpoints chosen by the above rules cut each loop of the FSMD in at least one cutpoint, because each internal loop has an exit point (ensured by our notion of computation in §2).

This basic steps of equivalence checking, i.e., constructing the path cover by inserting cutpoints in one FSMD and finding the equivalent path of each member of this path cover in the other FSMD, can be applied to the first two phases of high-level synthesis verification. The choice of cutpoints in the scheduling verification phase may differ from the way they will be chosen described in this section. The computation of equivalence in register sharing verification phase involves additional entities like mapping from the variables to the registers, mapping from the states of one FSMD to the states other FSMD. Also, the definition of equivalence of paths has to modify in these two phases depending upon the requirements of each phase. These issues along with the verification methodology based on equivalence checking are discussed in the subsequent two chapters. The last phase of HLS verification, i.e., the data-path and the controller verification, however, does not require path-based equivalence checking. Checking the condition and the register transfer (RT) operations performed in each transition of the two FSMD suffices for equivalence checking. In other words, the state based equivalence checking is capable of verifying the correctness of this phase. In the rest of the dissertation, the FSMDs corresponding to the behaviours at the input to HLS, after scheduling, after allocation and binding and at the output of HLS will be designated as $M_0 = \langle Q_0, q_{00}, I, V_0, O, f_0, h_0 \rangle$, $M_1 = \langle Q_1, q_{10}, I, V_1, O, f_1, h_1 \rangle$, $M_2 = \langle Q_2, q_{20}, I, V_2, O, f_2, h_2 \rangle$ and $M_3 = \langle Q_3, q_{30}, I, V_3, O, f_3, h_3 \rangle$, respectively. They are shown in the figure 2.3.

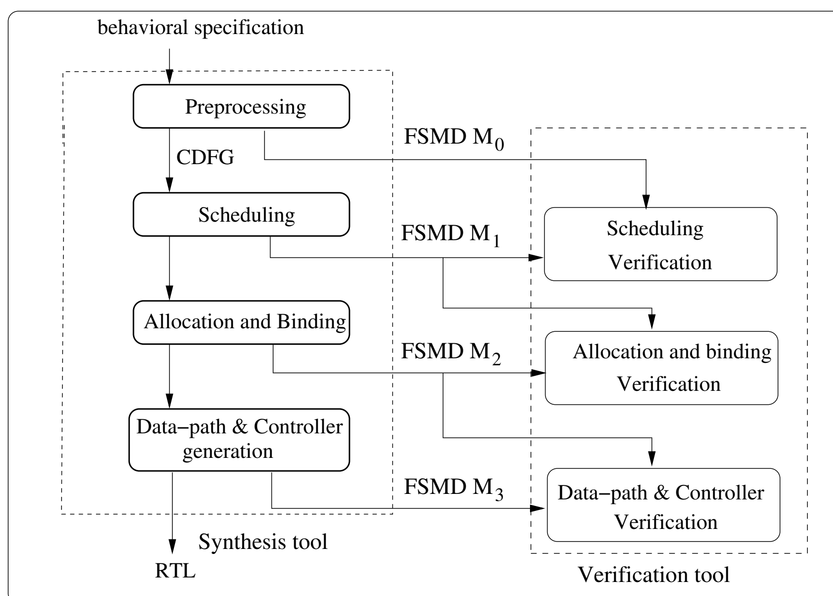


Figure 2.3: FSMD designations in the hand-in-hand synthesis and verification flow

2.3.2 Normalization of Arithmetic Expressions

While finding the equivalent path for a path, it is required to check the equivalence of the respective conditions as well as the data transformations of the paths. Since the condition of execution and the data transformation of a path involve the whole of integer arithmetic, checking equivalence of paths reduces to the validity problem of first order logic which is undecidable; thus, a canonical form does not exist for integer arithmetic. Instead, in this work we use the following normal form adapted from [48, 57]. The normalization process reduces many computationally equivalent formulas syntactically identical as it forces all the formulas to follow a uniform structure [48]. In the following, the normal form chosen for the formulas and the simplification carried out on the normal form during the normalization phase are briefly described.

A condition of execution (formula) of a path is a conjunction of relational and Boolean literals. A Boolean literal is a Boolean variable or its negation. A relational literal is an arithmetic relation of the form $s\ r\ 0$, where s is a normalized sum and $r \in \{\leq, \geq, =, \neq\}$. The relation $>$ ($<$) can be reduced to $\geq$ ($\leq$) over integers. For example, $x - y > 0$ can be reduced to $x - (y - 1) >= 0$. Negated relational literals are suitably modified to absorb the negation.

The data transformation of a path is an ordered tuple $\langle e_i \rangle$ of algebraic expressions such that the expression e_i represents the value of the variable v_i after execution of the path in terms of the initial data state. So, each arithmetic expression in data transformation can be represented in the

normalized sum form. A normalized sum is a sum of terms with at least one constant term; each term is a product of primaries with a non-zero constant primary; each primary is a storage variable, an input variable or of the form $abs(s)$, $mod(s_1, s_2)$, $exp(s_1, s_2)$ or $div(s_1, s_2)$, where s, s_1, and s_2 are normalized sums. This syntactic entities are defined by means of production of the following grammar.

Definition 10 *Grammar of normalized sum:*

1. $S \rightarrow S + T \mid c_s$, *where* c_s *is any integer.*
2. $T \rightarrow T * P \mid c_t$, *where* c_t *is any integer.*
3. $P \rightarrow S \uparrow C_e \mid abs\ (S) \mid (S)\ mod\ (S) \mid S \div C_d \mid c_m$, *where* c_m *is a symbolic constant.*
4. $C_e \rightarrow S \uparrow C_e \mid S$
5. $C_d \rightarrow S \div C_d \mid S$.

Thus, the exponentiation and the (integer) division are depicted by infix notation and all functions have arguments in the form of normalized sums.

In addition to the above structure, any normalized sum is arranged by lexicographic ordering of its constituent subexpressions from the bottom-most level, i.e., from the level of simple primaries.

Example 3 *The expression* $(x + 3y + 7 \geq 0 \wedge 4x^2 + 3yz + 2 \neq 0 \ \wedge x \uparrow y \geq 0)$ *will have the normal form*
$[1 * x + 3 * y + 7 \geq 0] \wedge [4 * x * x + 3 * y * z + 2 \neq 0] \wedge [(1 * x + 0) \uparrow (1 * y + 0) + 1 \leq 0]$
□

Various simplifications that can be carried out at the normalization phase are as follows:

- Simplification at the arithmetic expression (normalized sum) level:
  - Any expression involving only integer constants is immediately evaluated, e.g., $(5 \div 2)$ is evaluated to 2.
  - In an expression, common sub-expressions are collected together. For example, $x^2 + 3x + 5z + 4x$ is reduced to $x^2 + 7x + 5z$.
- Simplifications at the relational expression (relational literal) level:

  - Any relational expression built from constant arithmetic expressions may be immediately evaluated to "true" or "false". For example, $4 - 1 \geq 0$ is evaluated to *true*.
  - Common constant factors are extracted from the normalized sum and the relational expression is consequently simplified. For example, $3x^2 + 9xy + 6z + 7 \geq 0$ is mapped to $x^2 + 3xy + 2z + 2 \geq 0$, where $\lfloor 7 \div 3 \rfloor = 2$.

- Simplification at the formula level:
  - Some literals of the formula can be deleted by the rule "if $(A \rightarrow B)$ then $(A \wedge B \equiv A)$". For this step of simplification, it becomes necessary to detect implication among literals. It is possible to detect whether a relational literal implies another relational literal when they involve the same non-constant sums. Let the literals be $l_1 : (s_1 + c_1)R_1 0$ and $l_2 : (s_2 + c_2)R_2 0$. If $s_1 = s_2 = s$, then table 2.1 depicts the relationship between the constants c_1 and c_2 depending upon R_1 and R_2, which must be satisfied for l_1 to imply l_2. Removal of repetitions of literals in a formula is possible using this rule as for any literal l_1, $l_1 \rightarrow l_1$ is always *true*. For example, the literal $a \geq b$ has multiple occurrences in the formula $a \geq b \wedge c \leq d \wedge a \geq b$. So, this formula is simplified to $a \geq b \wedge c \leq d$.

    During this phase of simplification it is also checked whether $l_1 \rightarrow \neg l_2$, whereupon $l_1 \wedge l_2$ is reduced to false. For example, the formula $a \geq b \wedge c \leq d \wedge !(a \geq b)$ is evaluated to "false".

		$R_2 \rightarrow$			
		$=$	$\geq$	$\neq$	$\leq$
	$=$	$c_1 = c_2$	$c_2 \geq c_1$	$c_1 \neq c_2$	$c_2 \leq c_1$
R_1	$\geq$		$c_2 \geq c_1$	$c_2 > c_1$	
$\downarrow$	$\neq$			$c_1 = c_2$	
	$\leq$			$c_2 < c_1$	$c_2 \leq c_1$

Table 2.1: Conditions on c_1 and c_2 for which $(s_1 + c_1)R_1 0$ implies $(s_2 + c_2)R_2 0$

It can be easily seen that some of the properties of $\geq$, $\leq$, $=$ and $\neq$ are accommodated in course of normalization and simplification of formulas. For example, the symmetry property of $\{=, \neq\}$ is taken care of by choosing the normal form of relational literals as $s\{R\}0$ and by imposing ordering on the constituent sub-expressions at all levels. Thus, both $x = y$ and $y = x$ will be expressed either

as $x-y=0$ or $y-x=0$, consistently. Similar is case for $\neq$. Again, the reflexivity of $\{\geq, =, \leq\}$ and the irreflexivity of $\{\neq\}$ are accommodated by collecting the common sub-expressions in a sum. For example, $x \geq x$ changes to $x-x \geq 0$, whereupon the left-hand side reduces to "0" by collecting the common subexpressions x; accordingly, $x \geq x$ reduces to "true" by normalization.

2.4 Conclusions

The equivalence problem of two FSMDs is formulated in this chapter. The path, condition of execution of a path and data transformation along a path, computations on an FSMD and the path cover of an FSMD are defined. The equivalence of two FSMDs is defined and has been proved. The basic equivalence checking method is given. This method involves checking equivalence between two expressions over integers which is undecidable as canonical form does not exist for integer arithmetic. In this work, a normal form is adapted from [48, 57] to represent the arithmetic expressions over integers. This normal form is discussed in this chapter. Several simplifications that are carried out during normalization phase are also discussed in this chapter.

Chapter 3

Scheduling Verification

3.1 Introduction

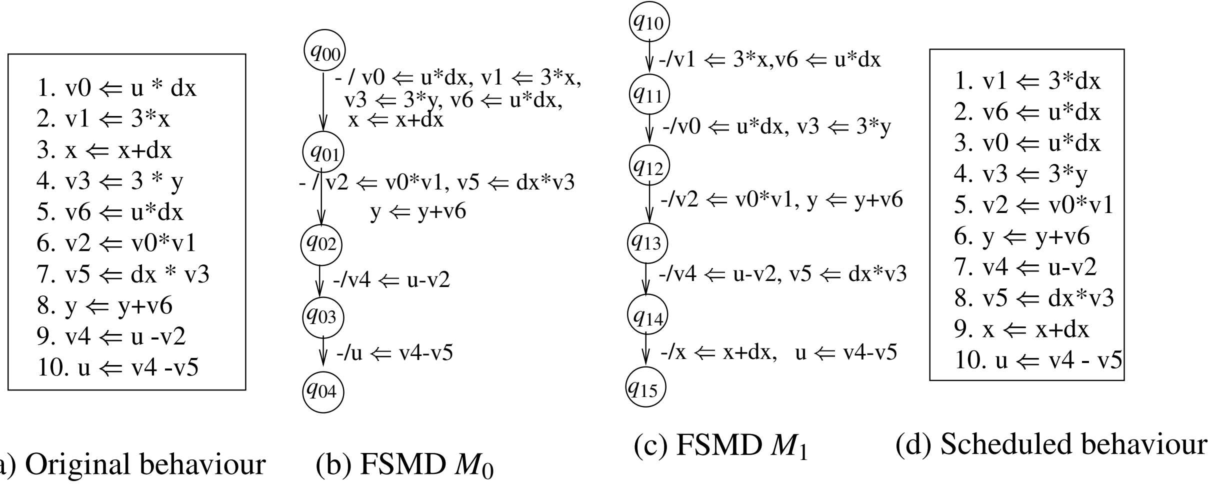
The goal of the scheduler is to optimize the number of control steps required to execute all the operations in the input behaviour meeting all the constraints regarding the number of control steps, the delay, the power and the hardware resources. The input to the scheduler is a CDFG of the input behaviour. Most of the scheduling algorithms [35, 6, 58, 59, 60, 61] are basic block based in the sense that they consider the basic blocks of the input CDFG one by one (starting from the initial block) and schedule the operations of each basic block independent of the operations of the other blocks. As a result, the input behaviour may be modified in the following ways:

1. The order of the operations can be changed.

2. New state(s) can be inserted in the FSMD corresponding to the the output of the scheduler.

Example 4 *Let us consider the example given in figure 3.1. It is assumed that in this example, the input behaviour consists of only one basic block. The order of the operations in the basic block is shown in figure 3.1(a). The FSMD M_0 corresponding to the input behaviour is shown in figure 3.1(b). The FSMD M_0 is constructed from the data flow graph (DFG) of the input behaviour. The scheduled FSMD M_1 is shown in figure 3.1(c) and the corresponding ordering of the operations is shown in figure 3.1(d). It is clear from figures 3.1(a) and 3.1(d) that the order of the operations are changed by the scheduler; for example, the operations $x \Leftarrow x + dx$ is moved from the 3rd step to the 9th step. Also, the scheduler introduces one state in the scheduled behaviour; specifically, the FSMD M_0 of figure 3.1(b) has five states whereas the FSMD M_1 of figure 3.1(c) has six states.*

New state is inserted in the scheduled behaviour due to resource constraints.



(a) Original behaviour (b) FSMD M_0 (c) FSMD M_1 (d) Scheduled behaviour

Figure 3.1: Phase-wise modification of an example input behaviour by a basic block based scheduler

□

There is another type of scheduler called *path-based scheduler* [37, 38]. As a result of this type of scheduling algorithm, the control structure of the input behaviour may be modified as the scheduler tries to merge some consecutive path segments of the input behaviour. Let us consider, for example, the FSMDs given in figure 3.2. Here, the scheduler merged the path segment p_{00} of the FSMD M_0 with the path segments p_{01} and p_{02}. Consequently, the control structure of the scheduled FSMD M_1 is different from that of the FSMD M_0.

There are several modern high-level synthesis tools, like SPARK [11], wavesched [62, 39], etc., which incorporate in the scheduling process several code motion techniques like, *speculation, reverse speculation, early condition execution, branch balancing, common sub-expression elimination, loop shifting, renaming,* etc., [47, 49, 39, 40, 41, 42, 63, 64]; these techniques lead to different transformations in the input behaviour. For example, some of the operations may be moved beyond the conditional statement (speculation, reverse speculation) [65, 40], the control structure of the input may be modified (conditional speculation, branch balancing) [66, 47], extra variables may be used to rename some of the variables in the input behaviour (renaming)[47], some of the variables and operations of the input behaviour may be eliminated (dead code elimination) [47], common sub-expressions may be eliminated [67] and also some extra operations may be added in the behaviour (speculation, loop shifting) [65, 68].

3.2 Objective of Scheduling Verification

As discussed above, the results of scheduling do not always have a one-to-one correspondence with the input. So, the goal of this verification phase is to ensure that the scheduling process preserves the behaviour of the original specification, irrespective of the scheduling technique used. The input and the output of the scheduler are encoded as FSMDs and the correctness of the scheduling process is ensured by checking for equivalence of these two FSMDs.

3.3 Verification Issues

It is clear from the discussion of the last section that the input to the scheduler may get transformed significantly during the scheduling process. Due to such transformations, the output FSMD M_1 may not have a one-to-one correspondence with the input FSMD M_0. Consequently, some modifications are needed in the equivalence checking method formulated in chapter 2 in order to verify the scheduling process. These modifications are as follows.

Case 1: The sets of variables V_0 of M_0 and V_1 of M_1 are not equal.

Some of the code motion techniques, like renaming and dead-code elimination during scheduling, may result in different storage variable sets V_0 of M_0 and V_1 of M_1. Since a path can start from any cutpoint of an FSMD, its condition of execution and the data transformation are in terms of the input signals and the variables. Thus, a path p_0 of M_0 may involve variables from the set V_0 while the corresponding path p_1 of M_1 may involve variables from V_1, where $V_0 \neq V_1$. To handle this difficulty, we have to consider the following restrictions. The condition of execution of any path of M_0 (M_1) is a logical expression which should be restricted over the set I of input signals and the set $V_0 \cap V_1$ of common variables. Similarly, the algebraic expressions that represent the final values of the variables in the data transformation should also be restricted over the set I and $V_0 \cap V_1$. Also, we only consider the final values of the variables that reside in $V_0 \cap V_1$ while checking the equivalence of the data transformations of the two paths. Restriction of the condition of execution and the data transformation of a path α to the variable set $V_0 \cap V_1$ are denoted as $R_\alpha|_{V_0 \cap V_1}$ and $r_\alpha|_{V_0 \cap V_1}$, respectively.

The restricted condition of execution and the restricted data transformation of a path in M_0 and in M_1 are *defined* if they are expressed in terms of the variables in $V_0 \cap V_1$. For example, let $V_0 = \{v_0,\ v_1,\ v_2\}$ and $V_1 = \{v_1,\ v_2,\ v_3\}$. So, $V_0 \cap V_1$ is $\{v_1,\ v_2\}$. Let the condition of execution of

a path in M_1 be $v_2 - v_1 \leq 10$. So, this condition of execution is *defined* when restricted to $V_0 \cap V_1$. Let us consider another path in M_1 with condition of execution $(v_1 - v_2 > 0 \wedge v_1 \leq v_2 + v_3)$ which is undefined when restricted to $V_0 \cap V_1$ as v_3 occurs in the expression. Let the data transformation of a path in M_1 be $\langle\langle v_1 - v_2,\ v_2 + 2,\ v_3 - v_1\rangle, -\rangle$, where the order of the variables is $v_1 \prec v_2 \prec v_3$. Under restriction to $\{v_1, v_2\}$, the transformation is $\langle\langle v_1 - v_2, v_2 + 2\rangle,\ -\rangle$. It may, therefore, be noted that the transformation of this path is defined even if the final value of the variable v_3 is not restricted to $V_0 \cap V_1$, because the final values of v_0 and v_3 are not considered during checking the equality of data transformations restricted to $V_0 \cap V_1$ of the two paths of M_0 and M_1. Consider another path in M_1 whose data transformation is $\langle\langle v_1 - 1,\ v_2 + v_3,\ v_2 + 1\rangle, -\rangle$. This transformation becomes undefined when restricted to $\{v_1, v_2\}$ as v_3 occurs in the expression that represents the value of the variable $v_2 \in V_0 \cap V_1$ after execution of the path.

However, the situations when the condition of execution or the data transformation of a path becomes undefined usually do not occur as argued below.

- Some of the variables of V_0 may not exist in V_1:

  It happens when the scheduler eliminates some dead code involving the variables in $(V_0 - V_1)$. Clearly, they have no effect on the condition of execution or in the data transformation of any path in M_0; that is the reason why the scheduler removes that part of the code. Hence, they will not occur in the condition of execution or in data transformation of any path in M_0.

- Some of the variables of V_1 may not exist in V_0:

  The scheduler generally uses some extra variables to reduce the data dependencies among the variables to increase the parallelism among the operations in the behaviour. These variables are first assigned some values in terms of $V_0 \cap V_1$ and I, and used subsequently. Obviously, these variables (in $V_1 - V_0$) will not occur in the condition of execution or the data transformation of any path in M_1.

- $V_0 \cap V_1$ is empty:

  This case does not occur for any scheduling algorithm. The scheduler may exclude some variables from V_0 (not all of them) or may introduce some extra variables.

It is clear from the above discussion that the restriction on the condition of execution and on the data transformation of any path has no effect in the equivalence checking. The following definition is in order.

Definition 11 *A path* α *of* $M_0 = \langle Q_0, q_{00}, I, V_0, O, f_0, h_0 \rangle$ *and a path* β *of* $M_1 = \langle Q_1, q_{10}, I, V_1, O, f_1, h_1 \rangle$ *are said to be equivalent if* R_α*,* r_α*,* R_β *and* r_β *are defined over* $V_0 \cap V_1$ *and* $R_\alpha|_{V_0 \cap V_1} = R_\beta|_{V_0 \cap V_1}$ *and* $r_\alpha|_{V_0 \cap V_1} = r_\beta|_{V_0 \cap V_1}$.

Case 2: The control structure of the input behaviour is modified or some operations are moved beyond the conditional statements.

As the control structure of the input behaviour may be modified by the path based scheduler and also some operations in the behaviour may be moved beyond the conditional statements by application of code motion techniques like speculation, reverse speculation, early condition execution, etc., the rules to find the cutpoints defined in subsection 2.3.1 do not work always. In the following, one algorithm is proposed which combines the first two steps of the equivalence checking method described in subsection 2.3.1 into one. More specifically, the method constructs a path cover by inserting cutpoints in M_0 initially using the rules given in subsection 2.3.1 and in course of finding its equivalent path set in M_1, the actual set of cutpoints of M_0 is identified dynamically.

The process starts with selecting cutpoints in M_0 according to the rules defined in subsection 2.3.1 and finding the initial set of paths, say P_0', from one cutpoint to another without having any intermediary cutpoint. Now, the actual path cover P_0 of M_0 and an equivalent path of M_1 for each member of P_0 are to be found. The algorithm takes each member of P_0' and tries to find an equivalent path in M_1. If an equivalent path is found for a path, p say, of P_0', then p is put in P_0. However, no path may exist in M_1 which is equivalent to p. Let p be $[q_{0i} \Rightarrow q_{0j}]$. The path p is extended by concatenating to itself all the paths of P_0' which emanate from q_{0j} in P_0'. It should be noted that all such extended paths should not violate the basic definition of a path that only the initial and the the final states can be identical; in other words, the extension process should not move through (and beyond) q_{0i}. All these concatenated paths are added to P_0' and p is deleted from P_0'. This process terminates when the equivalent paths for all the paths of P_0' are found in M_1. The fact that the constructed set P_0 is also a path cover of M_0 is proved shortly.

The following example illustrates the process of path extension.

Example 5 *Let us consider the FSMDs in figure 3.2. The cutpoints are denoted as shaded nodes in this figure and* $P_0' = \{p_{00}, p_{01}, p_{02}\}$*. The path* p_{00} *of* P_0' *does not have any equivalent path in FSMD* M_1*. So,* p_{00} *will be concatenated with* p_{01} *and* p_{02} *of* P_0'*. The concatenated paths are* p_{00}' *and* p_{01}' *as denoted in figure 3.2(a). Now, the equivalent path exists for each of these concatenated paths. The computed path cover* P_0 *consists of* p_{00}' *and* p_{01}' *and their respective equivalent paths*

in M_1 are p_{10} and p_{11}.

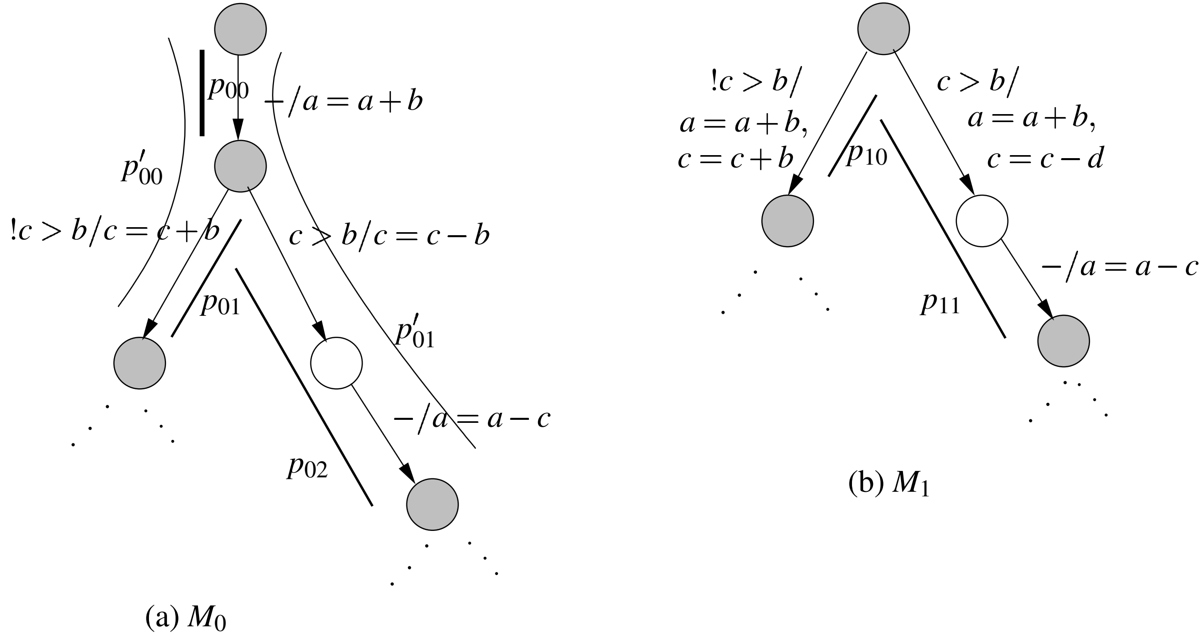


Figure 3.2: Working of the proposed algorithm on an example. (a) M_0: An FSMD before scheduling (b) M_1: Corresponding FSMD after scheduling

□

Case 3: The condition of state transitions in M_1 is Boolean.

The scheduler schedules not only the arithmetic operations of the behaviour but also the relational operations over the set $I \cup V_0$ that represents the condition of state transition in M_0. The relational operation used in a state s has to be scheduled prior to s. It is assumed here that the scheduler introduces a Boolean variable for each relational operation to store the result of that operation. This Boolean variable is used in the subsequent conditional statement(s) in the scheduled FSMD M_1.

Example 6 *Let us consider the input behaviour M_0 in figure 3.3. The relational operation $v1 \leq v2$ controls the state transitions from the state q_{0i_1} in figure 3.3(a). This operation is scheduled before the state q_{0i_1} and the result is stored in the Boolean variable 'le', as shown in the FSMD M_1 of figure 3.3(b). The variable 'le' now controls the state transition.*

□

Since the conditions for state transitions in the FSMD M_1 are represented in terms of Boolean variable(s), the condition of execution of each path in the FSMDs M_1 would involve only the

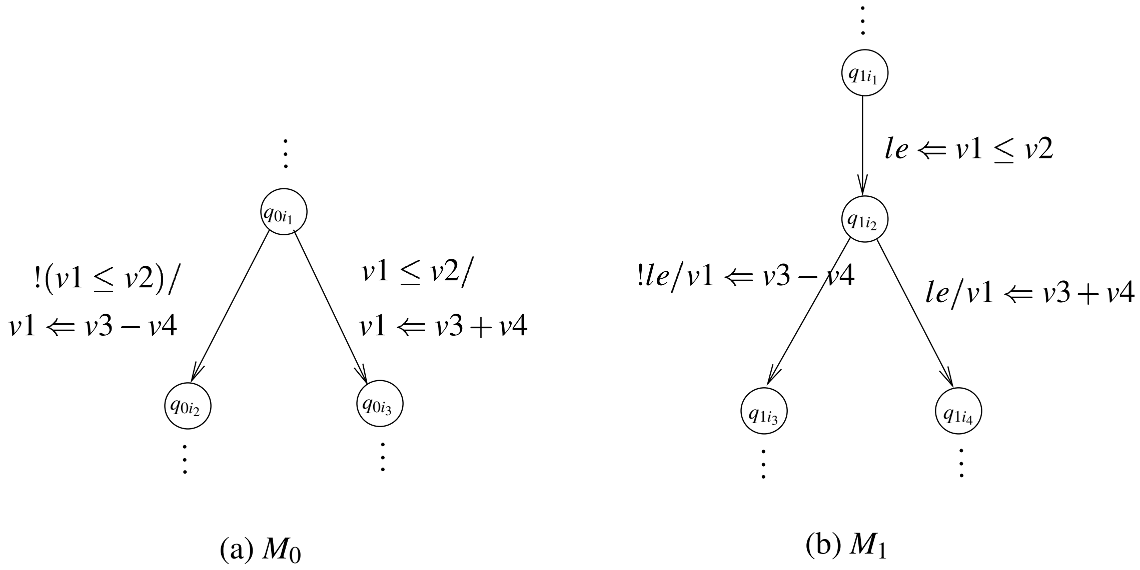


Figure 3.3: Scheduling of a relational operation

Boolean variable(s) whereas the same for M_0 would be over $I \cup V_0$. Due to this reason, the condition of execution of any path β in M_0 never matches with the same of any path α of M_1 even if the (Boolean) expression R_α actually represents R_β. For example, the path $\beta = q_{0i_1} \xrightarrow[v1 \leq v2]{} q_{0i_3}$ in the FSMD M_0 (with $R_\beta = v1 \leq v2$) and its equivalent path $\alpha = q_{1i_2} \xrightarrow[le]{} q_{1i_4}$ in the FSMD M_1 (with $R_\alpha = le$) of figure 3.3 provides a case in point.

It is, therefore, necessary to find the value of each Boolean variable that occurs in the condition of execution of any path p, say, in the FSMD M_1 during equivalence checking between the FSMDs M_0 and M_1. In other words, the relational expression that defines a Boolean variable in the condition of execution of p needs to be found out. This is achieved by performing a *backward breadth first search* from the start node of p. Next, the occurrences of the Boolean variable in the condition of execution of p is replaced by the the corresponding relational expression found by the backward breadth first search. For example, the Boolean variable 'le'occurs in the condition of execution of the path α (i.e., $R_\alpha = le$) in the FSMD M_1 of figure 3.3. The *backward breadth first search* from the state q_{1i_2}, the start node of the path α, finds the relational expression $le \Leftarrow v1 \leq v2$ as the value of the Boolean variable 'le'. So, R_α becomes $v1 \leq v2$. A Boolean variable is said to be undefined if more than one relational expressions are found during the backward breadth first search.

3.4 The Scheduling Verification Algorithm

3.4.1 The Algorithm

Input: The FSMDs M_0 and M_1.

Output: P_0: a path cover of M_0,

E: ordered pairs $\langle\beta,\alpha\rangle$ of paths of M_0 and M_1, respectively, such that $\beta \in P_0$ and $\beta \simeq \alpha$.

Step 1: Let η be the set of corresponding state pairs. Let $\eta \leftarrow \langle q_{00}, q_{10}\rangle$. Insert cutpoints in M_0 using the rules stated in the subsection 2.3.1. Let P_0' be the set of all the paths of M_0 from a cutpoint to a cutpoint having no intermediary cutpoint. Let P_0 and E be empty.

Step 2: If $P_0' =$ *empty*, then return P_0 as a path cover of M_0 and E as a set of ordered pairs of equivalent paths of M_0 (from P_0) and M_1 and *exit (success)*; else go to step 3.

Step 3: Find a path of the form $\langle q_{0i} \Rightarrow q_{0f}\rangle$ from P_0' s.t. q_{0i} has a corresponding state q_{1j}. If no path is obtained, then go to step 4; else go to step 5.

Step 4: If $P_0' \neq$ *empty*, then report "M_0 *may not be contained* in M_1" and *exit (failure)*; else return P_0 as a path cover of M_0 and E as a set of ordered pairs of equivalent paths of M_0 (from P_0) and M_1 and *exit (success)*.

Step 5: Let the path obtained in step 3 be $\beta = \langle q_{0i} \Rightarrow q_{0f}\rangle$. Let $\langle q_{0i}, q_{1j}\rangle$ be the corresponding state pair in η. If R_β or r_β is undefined, then report "*The* R_β *and/or* r_β *of* β *is not defined and exit (failure)*; else find a path of M_1 emanating from q_{1j} which is equivalent to the path β. If such a path is found, then go to step 6; else go to step 7.

Step 6: Let this path of M_1 be α. $\eta \leftarrow \eta \bigcup \{\langle endState(\beta), endState(\alpha)\rangle\}, E \leftarrow E \bigcup \{\langle\beta,\alpha\rangle\}$, $P_0 \leftarrow P_0 \bigcup \{\beta\}, P_0' \leftarrow P_0' - \{\beta\}$. go to step 2.

Step 7: $P_0' \leftarrow P_0' - \{\beta\}$. Extend β $(= \langle q_{0i} \Rightarrow q_{0f}\rangle)$ in M_0 by moving through the cutpoint q_{0f} till the next cutpoints but without moving through the reset state or any cutpoint more than once. Let B_m be the set of all such extensions of the path β. $P_0' \leftarrow P_0' -$ {paths of P_0' from q_{0f} which got appended to β }. $P_0' \leftarrow P_0' \bigcup B_m$. go to step 8.

Step 8: If $B_m =$ *empty*, then report "β *does not have any equivalent in* M_1 *and cannot be extended"* and *exit (failure)*; else go to step 2.

The above algorithm examines whether $M_0 \sqsubseteq M_1$. In order to examine the computational equivalence between M_0 and M_1, the above algorithm is rerun with M_0 and M_1 interchanged to determine whether $M_1 \sqsubseteq M_0$ or not.

3.4.2 Correctness of the algorithm

Theorem 2 *(*Termination*): The algorithm always terminates.*

Proof 2 *There are two loops namely,* $\langle 2,3,5,6,2\rangle$ *and* $\langle 2,3,5,7,8,2\rangle$. *The first loop pertains to the situation where the equivalent of a path* $\beta \in P'_0$ *is found and* β *is deleted from* P'_0. *Thus, when step 2 is revisited after one execution of this loop, the cardinality* $\|P'_0\|$ *of* P'_0 *decreases by 1. The second loop pertains to the situation where the equivalent of* $\beta = [q_{0i} \Rightarrow q_{0f}] \in P'_0$ *is not found. In this case* β *is extended by concatenating to itself all the paths of* P'_0 *which emanate from* q_{0f} *and satisfy the condition stated in step 7 of the algorithm. There are* $\|B_m\|$ *such paths. In step 7, the path* β *and the original* $\|B_m\|$ *paths emanating from* q_{0f} *are deleted and* $\|B_m\|$ *extended paths are added amounting to a net reduction of* $\|P'_0\|$ *by 1. Hence,each execution of either loop reduces* $\|P'_0\|$ *by 1. Since* $\|P'_0\|$ *is in the well-founded set [52] of non-negative numbers having no infinite decreasing sequence, the algorithm cannot execute (any combination of) the loops indefinitely.*

Theorem 3 *If the algorithm terminates in step* 2 *or in the else-clause of step* 4*, then* $M_0 \sqsubseteq M_1$.

Proof 3 *From theorem 1, it follows that* $M_0 \sqsubseteq M_1$ *if* P_0 *is a path cover of* M_0 *and* E *is the set of ordered pairs of equivalent paths of* P_0 *and those of* M_1. *Step* 5 *and step* 6 *of the algorithm ensure that* E *contains only pairs of equivalent paths of* M_0 *(belonging to* P_0*) and* M_1*; this property of* E*, therefore, is an invariant. So, what remains to be proved is that* P_0 *is a path cover of* M_0 *which follows from the following lemma.*

Lemma 1 *When the algorithm terminates successfully the set* P_0 *gives a path cover of* M_0.

Proof: *Let* C *be the set of cutpoints in* M_0. *Let* $C' \subseteq C$ *such that every cutpoint in* C' *has a corresponding state in* M_1*, that is, they form pairs of the set* η.
There are two "success"-exits, one in step 2 *and the other in step* 4*. The algorithm ensures that on its "success"-exits,* P_0 *contains all the paths of the form* $\langle q_{0l} \Rightarrow q_{0m}\rangle$*, where* $q_{0l}, q_{0m} \in C'$*, and in the paths, there is no other intermediary cutpoint belonging to* C'*. This follows from the following observations. If* $C' = C$*, then the final set* P_0 *is the same as the initial set* P'_0 *whose members are ensured to satisfy the above property in step* 1*. If* $C' \subsetneq C$*, then some of the original paths had to be extended in step* 7*. Since such extensions took place in all possible ways, that is, up to all possible successor cutpoints deleting the intermediary cutpoints, the above assertion again holds.*

Let P_0 be not a path cover of M_0, i.e., there exists some computation c_1, say, of M_0 that cannot be represented by concatenation of the members of P_0. Now, since P'_0 is known to be a path cover of M_0, there exists a concatenated path $\Pi_1 = [p'_{j_1} p'_{j_2} \cdots p'_{j_n}]$, $p'_{j_m} \in P'_0, 1 \leq m \leq n$ such that $c_1 \simeq \Pi_1$. Let p'_{j_m} be $\langle q_{0j_{m-1}} \Rightarrow q_{0j_m} \rangle$. In particular, $p'_{j_1} = \langle q_{0j_0} \Rightarrow q_{0j_1} \rangle$ and $p'_{j_n} = \langle q_{0j_{n-1}} \Rightarrow q_{0j_n} \rangle$, where $q_{0j_0}, q_{0j_n} = q_{00} \in C'$.
Let $\Pi_2 = [p'_{j_k} p'_{j_{k+1}} \cdots p'_{j_l}]$ be the first subsequence in Π_1 such that the start state of p'_{j_k} and the end state of p'_{j_l} are in C' i.e, $q_{0j_{k-1}}, q_{0l} \in C'$ and the end state of p'_{j_k}, the start state of p'_{j_l} and the terminal states of $p'_{j_{k+1}}, \cdots, p'_{j_{l-1}}$ are not in C'.
It is obvious that the prefix sequence in Π_1 preceding Π_2 is composed of paths from P_0. The fact that Π_2 can be replaced by a subsequence of paths from P_0 is established as follows. None of $p'_{j_k}, \cdots, p'_{j_l}$ contains any intermediary cutpoint from C and hence, from C'. Therefore Π_2 is essentially a path between two cutpoints in C' without having any intermediary cutpoint from C'. Since P_0 contains all such paths, Π_2 belongs to P_0.
By repeated applications of above argument, all such subsequent instances of Π_2 in Π_1 can be shown to be paths of P_0. Thus, Π_1 is rendered a concatenation of paths from P_0 (Contradiction).

3.4.3 Complexity of the algorithm

The overall complexity of the algorithm depends on two issues; the complexity of finding an equivalent path for a given path from a given state (step 5) and the number of iterations of the algorithm. So, the complexity of the algorithm is of the order of *number of iterations times the complexity of finding an equivalent path*.
Let the function $findequivalent$ find the path of M_1 which starts from the state q_{1j} of M_1 and is equivalent to the path β of M_0. Let us assume that there are n number of states in the FSMD M_1 and the maximum number of parallel edges between any two states is k. So, the maximum possible state transitions from a state is $k.n$. All the transitions emanating from a state have distinct conditions of execution. The function checks all transitions from q_{1j}. The condition of at most one of these transitions matches (may be partially) with R_β. Let the transition be $q_{1j} \rightarrow q_{1k}$. The partially constructed equivalent path α' then becomes $q_{1j} \rightarrow q_{1k}$. If the condition and the data transformation of α' match totally, then the equivalent path has been found. If it matches partially, the function will concatenate the transitions from the end node of α' with this path one by one and check for the equivalence. This process will continue until any repetition of nodes occurs

in α' other than between the start node and the end node or the equivalent path has been found. In the worst case, the function iterates n times. So, the complexity of finding equivalent path is $O(kn.n) = O(kn^2)$. However, the equivalent path can be found in $O(1)$ time when there is only one transition from q_{1j} which is equivalent to β.
It is required to find the equivalent path for every path in P'_0. Initially, this set contains at most $O(n^2)$ paths because in the worst case all the nodes of M_0 are cutpoints and the number of paths from one cutpoint to another without any intermediary cutpoint are $n(n-1)/2$. In the best case, the equivalent path for each member of this set can be found directly and no path extension is required. In the worst case, one path may be required to be extended n times. In this case, we have to consider $k.(n-1) + k^2.(n-1).(n-2) + \ldots + k^{(n-1)}.(n-1).(n-2). \ldots .2.1 \simeq k^{(n-1)}.(n-1)^{(n-1)}$ number of paths.

So, the complexity of our algorithm $O(k^{(n-1)}(n-1)^{(n-1)}.kn^2) = O(k^n n^{(n+1)})$ in the worst case and $O(n^2.1) = O(n^2)$ in the best case.

3.5 Verification of Different Scheduling Algorithms

3.5.1 Basic Block Based Scheduling

The input to the scheduler is a CDFG whereas the input to the verifier is its corresponding FSMD. Hence, it is indeed important to find the relation between this two. A CDFG consists of a set of basic blocks (BB) and a set of control blocks (CB). Each BB is essentially a data flow graph (DFG) and the CB decides the control flow among the basic blocks. So, the state corresponding to each CB is a cutpoint in the FSMD. Also, each path from one cutpoint to another without having any intermediary cutpoints in the FSMD represents a BB of the CDFG. This fact is depicted in figure 3.4.

It is already discussed that the basic block based schedulers consider each BB of the input CDFG independently and schedule the operations within that BB. As a result, whatever transformation occurs within the BB during scheduling would be limited to a path boundary. Consequently, the control structure of the output FSMD remains the same as that of the input.

During verification, the initial path cover P'_0 of M_0 is computed in step 1 of the algorithm. As the control structure of the input FSMD is not modified, no path extension is required here. The equivalent of each path of P'_0 will be found in step 5 of each iteration of the algorithm. Finally,

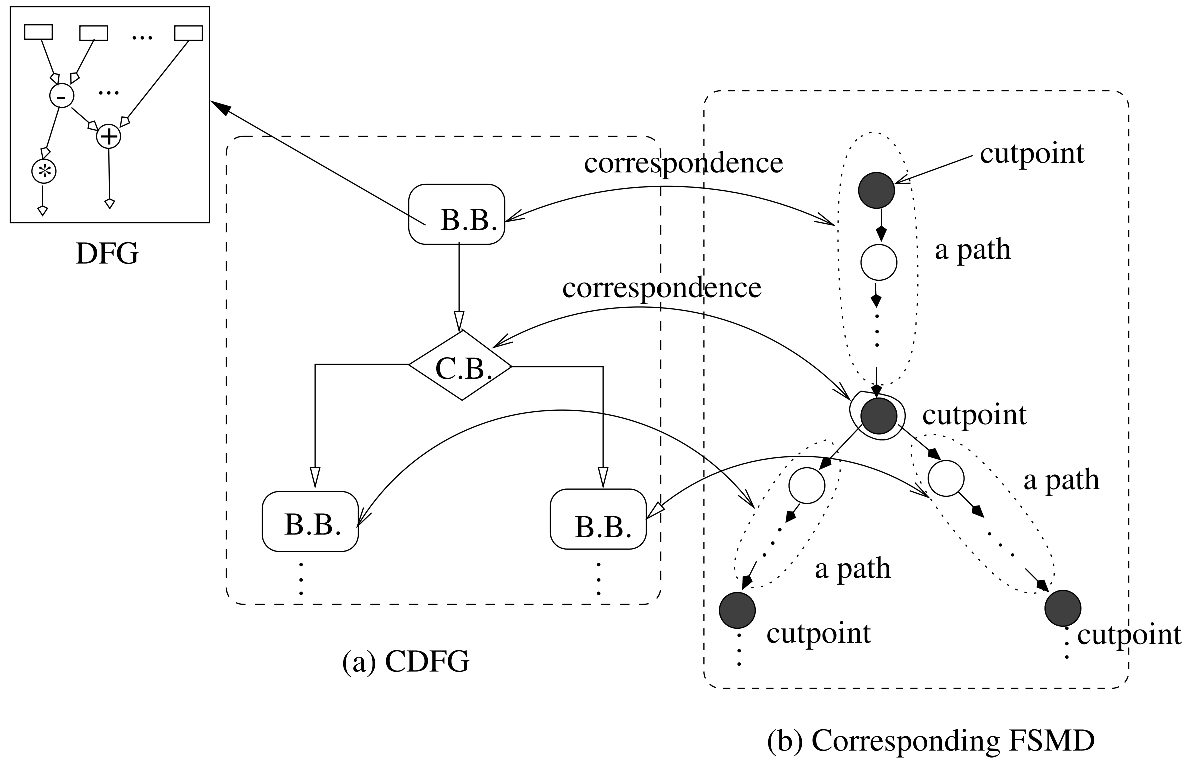


Figure 3.4: A CDFG and its corresponding FSMD structure

if the scheduling process is correct, then the algorithm terminates successfully in step 2 of the algorithm by producing P_0 ($\Leftarrow P_0'$) as the path cover of M_0.

3.5.2 Path Based Scheduling

In this subsection, the working of the algorithm for the path-based scheduler is briefly discussed with the GCD example depicted in figure 3.5 (a). The basic steps involved in a path-based scheduling algorithm [37] are as follows.

1. Each loop is broken by removing the feedback edge.

2. All paths that start from the start node of the CDFG or from the start node of a loop and end with nodes having no successors are computed.

3. The paths are scheduled independently for a given set of constraints.

4. The schedules for different paths are then combined to generate the final schedule of the design.

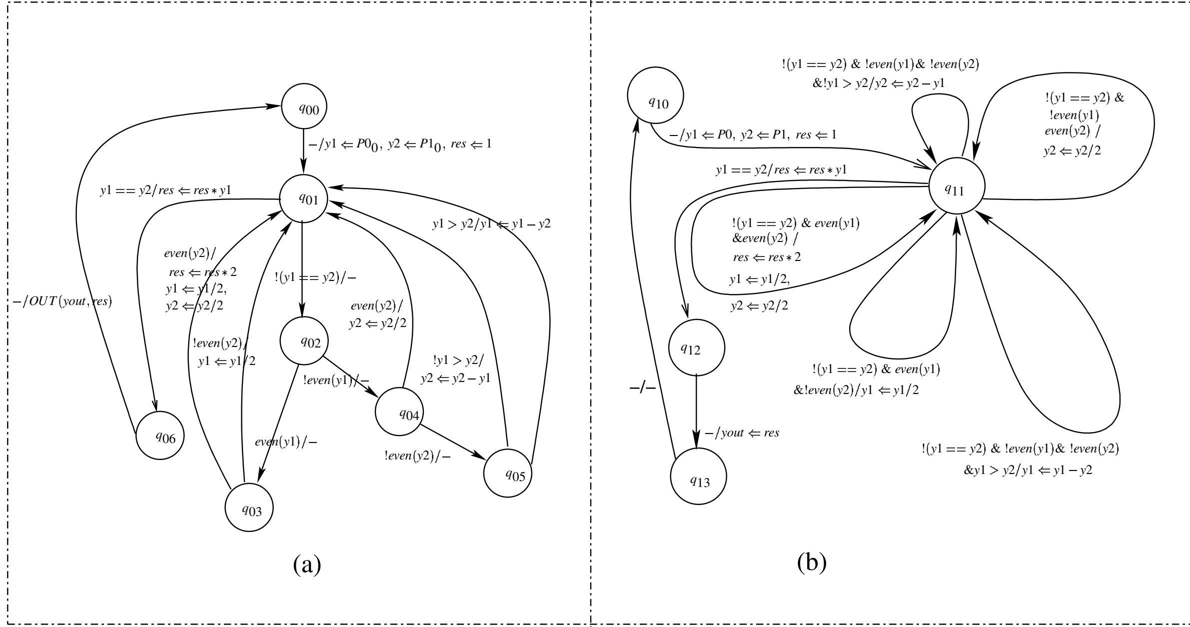


Figure 3.5: The FSMDs of the GCD example (a) M_0: before scheduling (b) M_1: after scheduling using a path-based scheduler

The path-based scheduling algorithm has been applied on the GCD example in figure 3.5 (a) and the scheduled FSMD is shown in figure 3.5 (b). It is clear from figure 3.5 that the consecutive path segments of M_0 are merged by the scheduling algorithm. Next, we will discuss how our algorithm works when the path segments are merged by the scheduler. The initial set of cutpoints is $\{q_{00},\ q_{01},\ q_{02},\ q_{03},\ q_{04},\ q_{05}\}$. The algorithm first finds $q_{10} \rightarrow q_{11}$ as the equivalent path of $q_{00} \rightarrow q_{01}$. It next takes $q_{01} \xrightarrow[(y1==y2)]{} q_{0e} \rightarrow q_{00}$ and finds $q_{11} \xrightarrow[(y1==y2)]{} q_{12} \rightarrow q_{1e} \rightarrow q_{10}$ as its equivalent path. The algorithm next considers the path $q_{01} \xrightarrow[!(y1==y2)]{} q_{02}$ and fails to find its equivalent path as this path has been merged with its successor paths by the scheduler. So, this path will be extended. The extended paths are $q_{01} \xrightarrow[!(y1==y2)]{} q_{02} \xrightarrow[even(y1)]{} q_{03}$ and $q_{01} \xrightarrow[!(y1==y2)]{} q_{02} \xrightarrow[!even(y1)]{} q_{04}$. The algorithm then considers the path $q_{01} \xrightarrow[!(y1==y2)]{} q_{02} \xrightarrow[even(y1)]{} q_{03}$. This path also needs to be extended and the extended paths are $q_{01} \xrightarrow[!(y1==y2)]{} q_{02} \xrightarrow[even(y1)]{} q_{03} \xrightarrow[even(y2)]{} q_{01}$ and $q_{01} \xrightarrow[!(y1==y2)]{} q_{02} \xrightarrow[even(y1)]{} q_{03} \xrightarrow[!even(y2)]{} q_{01}$. The algorithm finds the paths $q_{11} \xrightarrow[!(y1==y2)\ \wedge\ even(y1)\ \wedge\ even(y2)]{} q_{11}$ and $q_{11} \xrightarrow[!(y1==y2)\ \wedge\ even(y1)\ \wedge\ !even(y2)]{} q_{11}$ as the respective equivalent paths. Similarly, the path $q_{01} \xrightarrow[!(y1==y2)]{} q_{02} \xrightarrow[!even(y1)]{} q_{04}$ is also extended. The extended paths are $q_{01} \xrightarrow[!(y1==y2)]{} q_{02} \xrightarrow[!even(y1)]{} q_{04} \xrightarrow[even(y2)]{} q_{01}$ and $q_{01} \xrightarrow[!(y1==y2)]{} q_{02} \xrightarrow[!even(y1)]{} q_{04} \xrightarrow[!even(y2)]{} q_{05}$. The equivalent path of $q_{01} \xrightarrow[!(y1==y2)]{} q_{02} \xrightarrow[!even(y1)]{} q_{04} \xrightarrow[even(y2)]{} q_{01}$ is $q_{11} \xrightarrow[!(y1==y2)\ \wedge\ !even(y1)\ \wedge\ even(y2)]{} q_{11}$. The path $q_{01} \xrightarrow[!(y1==y2)]{} q_{02} \xrightarrow[!even(y1)]{} q_{04} \xrightarrow[!even(y2)]{} q_{05}$ will again be extended and the extended paths are $q_{01} \xrightarrow[!(y1==y2)]{} q_{02} \xrightarrow[!even(y1)]{} q_{04}$

$\xrightarrow[!even(y2)]{} q_{05} \xrightarrow[y1>y2]{} q_{01}$ and $q_{01} \xrightarrow[!(y1==y2)]{} q_{02} \xrightarrow[!even(y1)]{} q_{04} \xrightarrow[!even(y2)]{} q_{05} \xrightarrow[!(y1>y2)]{} q_{01}$. The paths $q_{11} \xrightarrow[!(y1==y2)\ \wedge\ !even(y1)\ \wedge\ !even(y2)\ \wedge\ (y1>y2)]{} q_{11}$ and $q_{11} \xrightarrow[!(y1==y2)\ \wedge\ !even(y1)\ \wedge\ !even(y2)\ \wedge\ !(y1>y2)]{} q_{11}$ are found as the respective equivalent paths by the algorithm.

3.6 Performance on Several HLS Transformations

The quality of synthesis results for most high-level approaches is strongly affected by the control structure and the data dependencies among the variables in the input behaviour. It might be possible to transform the input behaviour to some equivalent description which results in a more efficient implementation. This fact underlines the need for incorporating high-level compiler transformations in the scheduling phase of synthesis to overcome the effects of programming style on the quality of generated circuits. Needless to say, these transformations increase the scheduling verification challenges. In this section, several code transformation techniques, along with how these can be handled by our algorithm, are discussed. In the examples of this section, the FSMDs that are considered are segments from a cutpoint to cutpoint(s) of the original FSMDs.

3.6.1 Renaming

Extra variables are used to rename some variables of the original behaviour. It provides for parallel execution of some operations which were sequential due to data dependency among the variables in the input behaviour [47]. As a result, the total execution time of the scheduled behaviour is reduced.

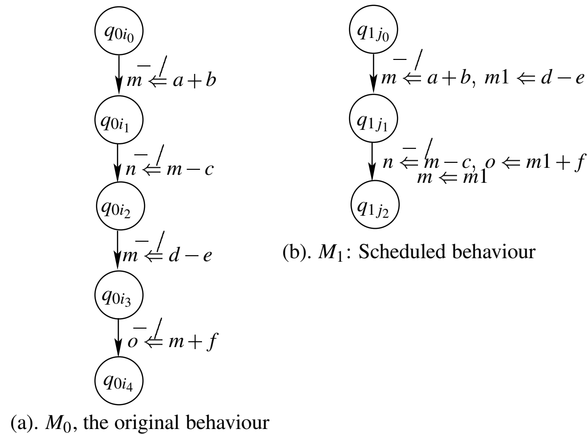


Figure 3.6: Scheduling using variable renaming technique: An example

Consider the FSMD M_0 in figure 3.6. There is a *read-after-write* dependency between the operations $m \Leftarrow a+b$ and $n \Leftarrow m-c$ as well as between $m \Leftarrow d-e$ and $o \Leftarrow m+f$. Also, there is a *write-after-write* dependency between $m \Leftarrow a+b$ and $m \Leftarrow d-e$ in the behaviour. So, no parallel execution is possible for the code and it requires four time steps to schedule the behaviour. In contrast, use of an extra variable $m1$ to store the result of the operation $d-e$ removes the *write-after-write* dependency in the behaviour. Consequently, $m \Leftarrow a+b$ and $m1 \Leftarrow d-e$ can be scheduled in parallel in the first time step as well as $n \Leftarrow m-c$ and $o \Leftarrow m1+f$ can be scheduled in parallel in the second time step. The scheduled behaviour is shown in figure 3.6 (b). As a result, the execution time is reduced by two clock cycles.

The proposed scheduling verification algorithm demonstrates the equivalence between the FSMDs M_0 and M_1 of figure 3.6 in the following way. Here, V_0, the (ordered) set of variables in M_0, is $\langle a, b, c, d, e, m, n, o\rangle$ and V_1, the (ordered) set of variables in M_1, is $\langle a, b, c, d, e, m, n, o, m1\rangle$. We are only concerned about the data transformation of the variables in $V_0 \cap V_i = \langle a, b, c, d, e, m, n, o\rangle$; the extra variable(s) $(m1)$ used to reduce the data dependencies are not considered for equivalence checking of paths. There is only one path in each FSMD. The condition of execution is $TRUE$ for both the paths. The data transformation is $\langle\, \langle a, b, c, d, e, d-e, a+b-c, d-e+f\rangle, -\rangle$ for the path in M_0 and $\langle\, \langle a, b, c, d, e, d-e, a+b-c, d-e+f, d-e\rangle, -\rangle$ for the path in M_1. So, the final values of the common variables are the same for both the paths. Hence, they are adjudged to be equivalent by the algorithm.

3.6.2 Common Sub-Expression Elimination

Common Sub-expression elimination (CSE) is a well-known transformation that detects the repeating sub-expressions in a piece of code, stores each of them in a variable and reuses the variable wherever the corresponding sub-expression occurs subsequently [67]. Consider the FSMD M_0 in figure 3.7. Let two adder/subtracters and a multiplier be available for the design. Then, it requires at least four time steps to schedule the operations in M_0. On the other hand, if the sub-expression $b+c$ is stored in a variable e' and is reused subsequently, then this behaviour can be scheduled in three time steps as shown in the figure 3.7(b). Note that the sub-expression $b*c$ of the operations $q \Leftarrow b*c$ and $f \Leftarrow b*c$ cannot be replaced by a common variable as the variable b is updated by the operation $b \Leftarrow b+c$ in between these two operations. The situation obtained by ignoring this aspect is shown in the FSMD M_1' in figure 3.7 (c). Our algorithm can find the non-equivalence

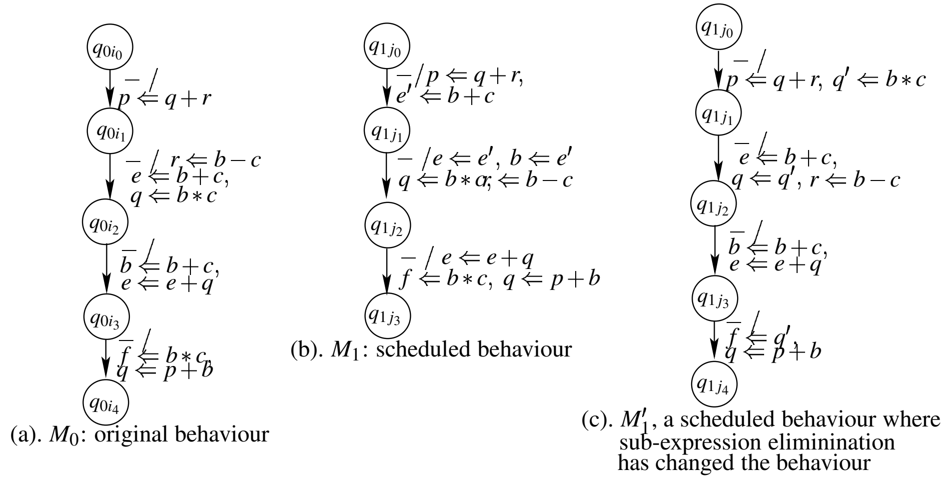


Figure 3.7: Scheduling using elimination of common sub-expressions: An example

between paths if the replacement of any sub-expression is not proper.

The algorithm considers only the variables belonging to $V_0 \cap V_1$. Since, the extra variables used to store the common sub-expressions do not belong to $V_0 \cap V_1$, they have no effect in equivalence checking by the algorithm.

For instance, the computation of data transformation by *forward substitution method* for the path $\beta = \langle q_{0i_0} \Rightarrow q_{0i_4} \rangle$ of M_0 comprises the following steps: $\langle \langle b, c, e, f, p, q, r \rangle, - \rangle \rightarrow \langle \langle b, c, e, f, q+r, q, r \rangle, - \rangle \rightarrow \langle \langle b, c, b+c, f, q+r, b*c, b-c \rangle, - \rangle \rightarrow \langle \langle b+c, c, b+c+b*c, f, q+r, b*c, b-c \rangle, - \rangle \rightarrow \langle \langle b+c, c, b+c+b*c, (b+c)*c, q+r, q+r+b+c, b-c \rangle, - \rangle$. For the path $\langle q_{1j_0} \Rightarrow q_{1j_3} \rangle$ of M_1, the computation consists of the following steps. $\langle \langle b, c, e, f, p, q, r, e' \rangle, - \rangle \rightarrow \langle \langle b, c, e, f, q+r, q, r, b+c \rangle, - \rangle \rightarrow \langle \langle b+c, c, b+c, f, q+r, b*c, b-c, b+c \rangle, - \rangle \rightarrow \langle \langle b+c, c, b+c+b*c, (b+c)*c, q+r, q+r+b+c, b-c, b+c \rangle, - \rangle$. The final values of all the variables of $V_0 \cap V_1 = \{b, c, e, f, p, q, r\}$ are the same for both the paths. Hence, they are adjudged to be equivalent by the algorithm. On the other hand, the final values of the variables for the path $\langle q_{1j_0} \Rightarrow q_{1j_4} \rangle$ of M'_1 are $\langle \langle b+c, c, b+c+b*c, b*c, q+r, q+r+b+c, b-c, b*c \rangle, - \rangle$ where the variables are in the order $b \prec c \prec e \prec f \prec p \prec q \prec r \prec q'$. Here, the final value of the variable f differs from that for β and the algorithm will report this non-equivalence.

3.6.3 Code Transformation to Increase Conditional Reuse of Hardware

An FSMD model is deterministic. So, all the conditional branches from a state, q_{0i} say, are mutually exclusive, i.e., at any instant when the system is in the state q_{0i}, the condition of only one branch from q_{0i} evaluates to true by the data state of the variable. So, the operations in different conditional branches can be scheduled in the same time step and executed by the same hardware. It means that the same hardware is *conditionally reused* [69] to execute several operations of different conditional branches in the same time step. Consider, for example, the FSMD M_0 in figure 3.8 (a). Let the condition c_1, c_2 do not change due to the data transformations. Here, the conditions of two branches from the state q_{0i_0} (or from the state q_{0i_1}) are mutually exclusive. So, the operations $t \Leftarrow a+b$ and $t \Leftarrow c+d$ can be executed by an adder in the first time step and the operations $s \Leftarrow c*d$ and $s \Leftarrow a*b$ can be executed by a multiplier in the second time step. The possibility of *conditional reuse*, however, is restricted by the way in which specifications are written by the designers. It might be possible to transform the original behaviour to some equivalent one for which there is a better possibility of conditional reuse of resources. For example, the FSMD M_0 of figure 3.8 can be transformed to the equivalent one shown in M_0'. Here, the conditions of the transitions $q'_{0i_0} \rightarrow q'_{0i_1}$ and $q'_{0i_2} \xrightarrow[c1]{} q'_{0i_3}$ are the same with no constraint of data dependency in the corresponding operations. So, the operations of these two transitions can be scheduled in the same time step. The same situation is also reflected for the transitions $q'_{0i_0} \rightarrow q'_{0i_2}$ and $q'_{0i_2} \xrightarrow[!c1]{} q'_{0i_3}$. The modified FSMD is represented by M_0'' and the corresponding scheduled behaviour is shown in M_1. It is clear that the conditional reuse of resources (the adder and the multiplier) are more in M_1 compared to M_0. Also, the transformed behaviour (M_0'') is scheduled in one time step (M_1) whereas the behaviour in M_0 will take two time steps to execute using the same hardware.

The verification task here is to check the correctness of transformation as well as the scheduling process involved. Thus, the algorithm is given to check the equivalence between the original behaviour M_0 and the scheduled behaviour M_1.

Consider the execution of the algorithm for the example described in figure 3.8. The cut-points in M_0 are q_{0i_0}, q_{0i_1}, q_{0i_2}. Let $\langle q_{0i_0},\ q_{1j_0} \rangle$ be the corresponding state pair. The paths starting from the state q_{0i_0} will be checked first. Step 5 of our algorithm fails to find the equivalent path of $q_{0i_0} \xrightarrow[!c1\ \vee\ !c2]{} q_{0i_1}$ in M_1. So, this path is extended through its immediate successor paths. The extended paths are $q_{0i_0} \xrightarrow[!c1\ \vee\ !c2]{} q_{0i_1} \xrightarrow[c1]{} q_{0i_2}$ and $q_{0i_0} \xrightarrow[!c1\ \vee\ !c2]{} q_{0i_1} \xrightarrow[!c1]{} q_{0i_2}$. For the path $q_{0i_0} \xrightarrow[!c1\ \vee\ !c2]{} q_{0i_1} \xrightarrow[c1]{} q_{0i_2}$, the condition of execution is $(!c1\ \vee\ !c2)\ \wedge\ c1 \equiv c1\ \wedge !c2$. The corresponding equivalent path in the scheduled behaviour is $q_{1j_0} \xrightarrow[c1\ \wedge\ !c2]{} q_{1j_1}$. For the path

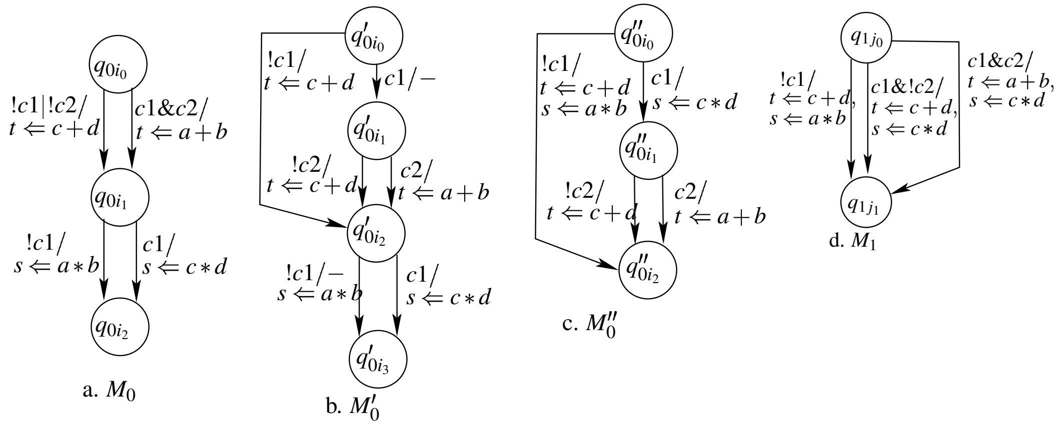


Figure 3.8: Conditional reuse to reduce execution time: An example

$q_{0i_0} \xrightarrow[!c1 \vee !c2]{} q_{0i_1} \xrightarrow[!c1]{} q_{0i_2}$, the condition of execution is $(!c1 \vee !c2) \wedge !c1 \equiv !c1$. Its equivalent path is $q_{1j_0} \xrightarrow[!c1]{} q_{1j_1}$. These equivalent paths are found successfully by the algorithm. Similarly, while trying to find the path of M_1 equivalent to the other path $q_{0i_0} \xrightarrow[c1 \wedge c2]{} q_{0i_1}$, the algorithm detects the need to extend this path. The extended path $q_{0i_0} \xrightarrow[c1 \wedge c2]{} q_{0i_1} \xrightarrow[!c1]{} q_{0i_2}$ becomes infeasible because its condition of execution is computed to be $c1 \wedge c2 \wedge !c1 \equiv false$. The other extended path $q_{0i_0} \xrightarrow[c1 \wedge c2]{} q_{0i_1} \xrightarrow[c1]{} q_{0i_2}$ is successfully processed by the algorithm by finding its equivalent path $q_{1j_0} \xrightarrow[c1 \wedge c2]{} q_{1j_1}$.

3.6.4 Reverse Speculation

In *reverse speculation*, the operations before a conditional are moved into the blocks subsequent to the conditional [65]. In general, reverse speculation leads to duplication of operations into both the conditional branches. This technique is also known as *lazy execution* [40]. For example, in figure 3.9, the operation d is reverse speculated into the conditional branches. From now onwards, the operations are represented as alphabets in the figures for brevity. An operation a indicates that the variable a is updated by this operation. For example, the operation $x \Leftarrow y+z$ is represented by x and an operation $c \Leftarrow x < y$ is represented as c. Obviously, a conditional state (from which the conditional branches start) is a cutpoint in our algorithm. Let q_{0i} be such a state and α be a path that ends in q_{0i}. As some of the operations from α are moved into the conditional branches by reverse speculation, the equivalent of path α does not exist in the scheduled FSMD. Hence, the algorithm extends the path α through q_{0i}. If the equivalent of the the paths obtained by extending α exist in the scheduled FSMD, they can be found by our algorithm. For example, let α be the

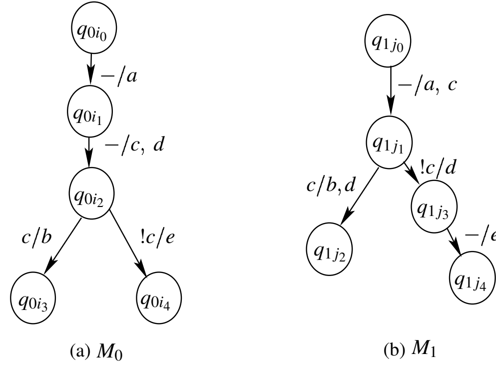


Figure 3.9: Reverse speculation technique: An example

path $q_{0i_0} \rightarrow q_{0i_1} \rightarrow q_{0i_2}$ in figure 3.9(a). No equivalent of this path exists in M_1 of figure 3.9(b). Hence α will be extended through q_{0i_2}. The extended paths are $q_{0i_0} \rightarrow q_{0i_1} \rightarrow q_{0i_2} \xrightarrow[c]{} q_{0i_3}$ and $q_{0i_0} \rightarrow q_{0i_1} \rightarrow q_{0i_2} \xrightarrow[!c]{} q_{0i_4}$. The corresponding equivalent paths in M_1 are $q_{1j_0} \rightarrow q_{1j_1} \xrightarrow[c]{} q_{1j_2}$ and $q_{1j_0} \rightarrow q_{1j_1} \xrightarrow[!c]{} q_{1j_3} \rightarrow q_{1j_4}$, respectively and are found by the algorithm.

However, the scheduler may move an operation, o say, before the conditional into only one conditional branch in some special case of reverse speculation. This is possible when the operations in the other branch as well as all the operations following the merging of the conditional branches are not dependent on the result of operation o. Consider the example in figure 3.10. The operation '$d \Leftarrow a+b$' of the original behaviour is moved to only one conditional branch with condition $!b > c$. This is possible because the operations in the conditional branch with condition $b > c$ and all possible execution paths following the merging node q_{0i_5} of M_0 do not use the value of d which is $a+b$. Our algorithm fails in this case. However, one modification in step 5 of our algorithm will suffice for equivalence checking. Let β be a path in M_0 of the form $\langle q_{0i} \Rightarrow q_{0j} \rangle$ and $\langle q_{0i}, q_{1k} \rangle$ be a corresponding state pair. Let the step 5 of the algorithm fail to find the equivalent path of β. Let there exist a path starting from q_{1k} in M_1, say α, whose condition of execution matches with that of β but the data transformation does not match. In such a case, we will check whether there is any variable of $V_0 \cap V_1$ which is modified along β but not modified in the path α. Let v_l be such a variable. Without any loss of generality, let the values of all the variables in $V_0 \cap V_1$ other than v_l at the end of execution of α be the same as those for β. Now, if we can show that the transformed value of v_l in β is not used in any execution path starting from q_{0j}, then α is equivalent to β even if their respective data transformations match partially (only on the other variables). In other words, if the variable v_l is always used only after it is defined in the subsequent execution paths from q_{0j},

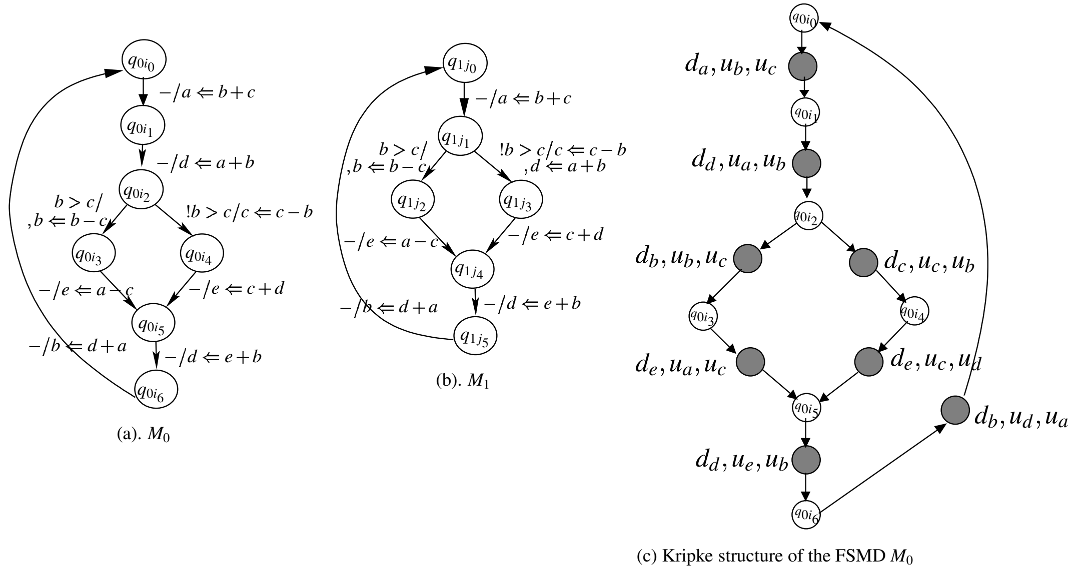


Figure 3.10: Reverse speculation: A special case

then there is no use of the operation that updates v_l in β and we can remove this operation during scheduling.

So, the question arises how we can ensure that the variable v_l is not used before it is defined in any execution path starting from q_{0j}. We convert the FSMD M_0 into an equivalent Kripke structure [70] by some logical transformations. A dummy state will be added for every transition of the FSMD. For example, figure 3.10 (c) represents the Kripke structure of the FSMD in figure 3.10 (a). The dummy states are denoted as shaded circles in the model. There would be two propositions, d_v and u_v, for each variable in $V_0 \cap V_1$, where d_v and u_v represent *defined* v and *used* v, respectively. The proposition d_v will be true in a dummy state if the variable v is defined by some operation in the corresponding transition in the FSMD. Similarly, u_v will be true in a dummy state if the variable v is used in some operation in the corresponding transition in the FSMD. For example, the propositions d_a, u_b, u_c are true in the dummy state between the states q_{0i_0} and q_{0i_1} in figure 3.10 (c) as the variable a is defined and b, c are used in the operation $a \Leftarrow b + c$ in the corresponding transition in M_0 in figure 3.10 (a). By convention, if any proposition is not present in any state of the Kripke structure, then the negation of the proposition is true in that state. Now, the required property that there does not exist any path in which v_l has not been defined before it is used can be written as the CTL formula $\neg$ E$[(\neg d_{v_l})$ W $u_{v_l}]$, where W represents the *week-until* operator. The formula E$[\phi$ W $\psi]$ true in a state s of a Kripke structure K iff there is a path $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \ldots$ in K, where s_1 equals s, and either $\forall i$, $s_i \models \phi$ or there is some s_i along

the path such that $s_i \models \psi$ and $\forall j < i$, we have $s_j \models \phi$. This formula can be easily verified using any CTL model checker such as NuSMV [71]. If this formula is true in the state q_{0j}, then step 5 declares β as the equivalent path of α.

Consider the example in figure 3.10. The algorithm considers $\beta = q_{0i_0} \rightarrow q_{0i_1} \rightarrow q_{0i_2}$ of M_0 and fails to find the equivalent path in M_1 in step 5. It finds $\alpha = q_{1j_0} \rightarrow q_{1j_1}$ in M_1 which has the same condition (true) as that of β but the variable d is transformed along β but not along α; the other variable a gets transformed identically. It, next, finds that the formula $\neg$ E$[(\neg d_d)$ W $u_d]$ is not true in state q_{0i_2} in the Kripke structure of the FSMD M_0. So, the control goes to step 7 and extends β. The extended paths are $\beta = q_{0i_0} \rightarrow q_{0i_1} \rightarrow q_{0i_2} \xrightarrow[b>c]{} q_{0i_3} \rightarrow q_{0i_5} \rightarrow q_{0i_6} \rightarrow q_{0i_0}$ and $q_{0i_0} \rightarrow q_{0i_1} \rightarrow q_{0i_2} \xrightarrow[!b>c]{} q_{0i_4} \rightarrow q_{0i_5} \rightarrow q_{0i_6} \rightarrow q_{0i_0}$. The equivalent path of the latter one in M_1 is $q_{1j_0} \rightarrow q_{1j_1} \xrightarrow[!b>c]{} q_{1j_3} \rightarrow q_{1j_4} \rightarrow q_{1j_5} \rightarrow q_{1j_0}$. Step 5 fails to find the equivalent path of the former path; it then finds the path $\alpha = q_{1j_0} \rightarrow q_{1j_1} \xrightarrow[b>c]{} q_{1j_2} \rightarrow q_{1j_4} \rightarrow q_{1j_5} \rightarrow q_{1j_0}$ in M_1 which has the same condition of execution with β. Again, the variable d is transformed along β but not along α. It, next, finds that the formula $\neg$ E$[(\neg d_d)$ W $u_d]$ is true in state q_{0i_0} in the Kripke structure of the FSMD M_0. So, α is equivalent to β. In this way, the equivalence of M_0 and M_1 can be established.

3.6.5 Early Condition Execution

This transformation involves restructuring the original code so as to execute the conditional operations as soon as possible. This, in effect, means that the conditional operation is "moved-up" in the

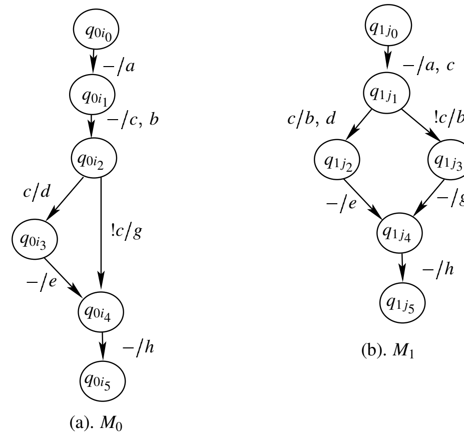


Figure 3.11: Early Condition Execution: An example

design, and hence, all the operations before the conditional operation are *reverse speculated* into

the conditional branches [65]. In figure 3.11, the conditional statement c is executed one step early in the scheduled behaviour and the operation b is reverse speculated in the conditional branches.

This is also a kind of reverse speculation which, however, can be handled in its entirety by our algorithm.

3.6.6 Conditional Speculation

There may be idle resources in some control steps of the conditional branches. To utilize such idle resources, the operations that lie after the conditional branches can be duplicated up into the conditional branches. This technique is known as *conditional speculation* [47]. Consider the example in figure 3.12. Let there be one adder/subtracter and one multiplier available for the design. It means that the multiplier is idle in both the conditional branches. Hence, the operation $z \Leftarrow x * y$ that lies after the conditional branch can be *duplicated-up* or *conditionally speculated* into both branches thereby reducing the total execution time by 1 unit.

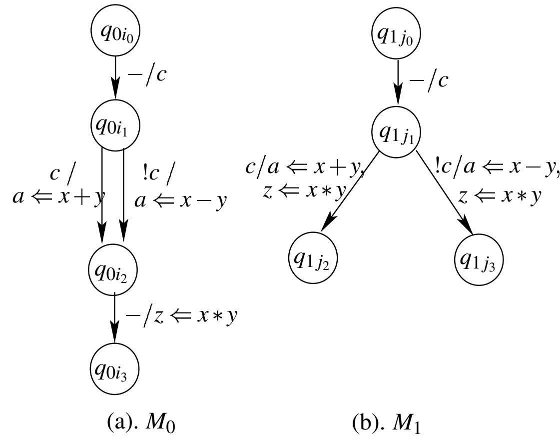


Figure 3.12: Conditional speculation technique: An example

The conditional speculation essentially involves moving up some operations within a path ensuring that the paths remain the same in both the FSMDs, that is, each path has the same condition of execution and data transformation. Hence, the equivalence can be found by the algorithm. For the FSMD M_0 in figure 3.12, the paths are $q_{0i_0} \rightarrow q_{0i_1}$, $q_{0i_1} \xrightarrow[c]{} q_{0i_2} \rightarrow q_{0i_3}$ and $q_{0i_1} \xrightarrow[!c]{} q_{0i_2} \rightarrow q_{0i_3}$. The corresponding equivalent paths in figure 3.12 (b) are $q_{1j_0} \rightarrow q_{1j_1}$, $q_{1j_1} \xrightarrow[c]{} q_{1j_2}$ and $q_{1j_1} \xrightarrow[!c]{} q_{1j_2}$, respectively and are found successfully by the algorithm.

3.6.7 Conditional Branch Balancing

Design descriptions often have situations where one conditional branch has more operation(s) than the other. This is known as *unbalanced* conditional branches [66]. A typical situation is depicted in the FSMD M_0 in figure 3.13(a). The data dependencies between the operations of M_0 are shown within the dotted rectangle in figure 3.13 (b). In a data dependency graph, the expression $a \rightarrow b$ indicates that the operation b depends on the result of the operation a.

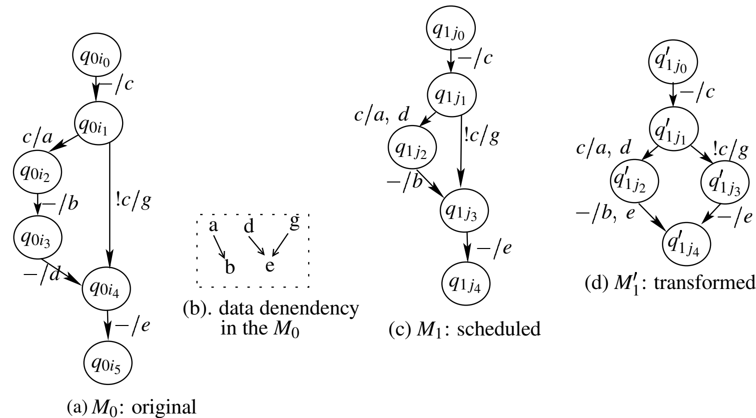


Figure 3.13: Conditional branch balancing: An example

The scheduled FSMD M_1 is shown in figure 3.13(c). In this case, it might be possible to insert a new scheduling step in the branch with fewer scheduling steps, that is, the branch with condition $!c$ without affecting the longest delay path. This extra step enables the conditional speculation of operation e (figure 3.13(d)) which effectively shortens the longest delay path by one scheduling step. The conditional branch balancing is a special type of conditional speculation. So, this can also be handled by our algorithm.

3.6.8 Speculation

Speculation execution refers to the unconditional execution of instructions that were originally supposed to be executed conditionally [65]. In this approach, the result of a speculated operation is stored in a new variable. If the condition under which the operation was to execute evaluates to true, then the stored result is copied to the variable from the original operation, else the stored result is discarded. In figure 3.14, the operation $d \Leftarrow x+y$ is speculated out of the branch with condition $!c$ of the FSMD M_0 and the result of the operation is stored in d'. It may be noted that if we do not

store the value in d', then the variable a gets the wrong value (by the operation $a \Leftarrow b + d$) when the execution is through the branch with condition c of the FSMD M_1.

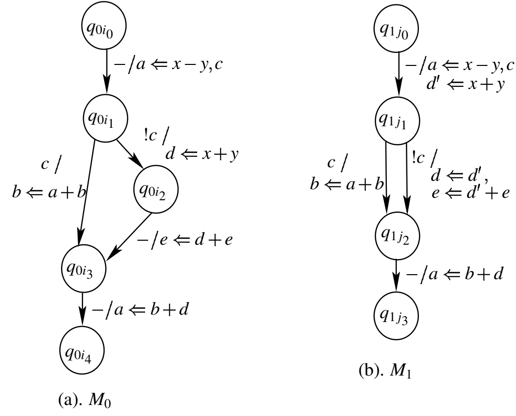


Figure 3.14: Speculation technique: An example

Our algorithm fails in this case. The algorithm uses any node with more than one outward transition as a cutpoint. So the node q_{0i_1} in M_0 of figure 3.14 is a cutpoint. The algorithm first tries to find an equivalent of the path $\beta = q_{0i_0} \rightarrow q_{0i_1}$. It finds $\alpha = q_{1j_0} \rightarrow q_{1j_1}$ as the equivalent path of β because the condition of execution (TRUE) and the data transformation of the common variables (a, b, c, d, e, x and y) are the same for β and α. Next, the algorithm tries to find the equivalent of the path $\beta = q_{0i_1} \xrightarrow[c]{} q_{0i_3} \rightarrow q_{0i_4}$. The corresponding equivalent path found by the algorithm is $\alpha = q_{1j_1} \xrightarrow[c]{} q_{1j_2} \rightarrow q_{1j_3}$. Now, it tries to find the equivalent of the path $\beta = q_{0i_1} \xrightarrow[!c]{} q_{0i_2} \rightarrow q_{0i_3} \rightarrow q_{0i_4}$. The algorithm fails to find the equivalent path of β because the data transformation of β is $\langle \langle b+x+y, b, c, x+y, x+y+e, x, y \rangle, - \rangle$ where the variables are in the order $a \prec b \prec c \prec d \prec e \prec x \prec y$. There is no path in M_1 starting from q_{1j_1} with the same data transformation.

Actually, the path $\alpha = q_{1j_1} \xrightarrow[!c]{} q_{1j_2} \rightarrow q_{1j_3}$ of M_1 is equivalent to the path β. As we are using *symbolic simulation* to find the data transformation, the final values of d, a and e in α will be in terms of d'. Specifically, the data transformation of α is $\langle \langle b+d', b, c, d', d'+e, x, y, d' \rangle, - \rangle$, where the variables are in the order $a \prec b \prec c \prec d \prec e \prec x \prec y \prec d'$ and will not match that of β. But if we use the right hand side expression $x+y$ of the operation $d' \Leftarrow x+y$ that defines d' in the path $q_{1j_0} \rightarrow q_{1j_1}$ as the initial symbolic value of d', then the data transformation in the path α becomes $\langle \langle b+x+y, b, c, x+y, x+y+e, x, y, x+y \rangle, - \rangle$ which is equal to that of β. Thus, α can be ascertained to be equivalent to β.

The following simple modifications in our algorithm can handle this code motion technique. We put cutpoints in M_1 by the rules proposed in the subsection 2.3.1 and find the set P_1' of paths from one cutpoint to another without having any intermediate cutpoints. While finding the equivalent of a path, say β, of M_0 in M_1, paths starting from the corresponding state of the start node of β are considered one by one. Let β be of the form $\langle q_{0i} \Rightarrow q_{0j} \rangle$ and $\langle q_{0i}, q_{1k} \rangle$ be the corresponding state pair. So, the paths starting from q_{1k} will be checked one by one until an equivalent path is found, failing which it is concluded that no equivalent path exists for that path. Let α be a path which starts from q_{1k}. If it is found that the some variables not belonging to $V_0 \cap V_1$ are used before they are defined along α during computation of R_α and r_α, then we will find the set of paths from P_1' which terminate in q_{1k}. Next, the last operations defining these variables in those paths will be found out. This can be done by *backward breadth first search* from q_{1k}. The right hand side expressions of those operations will be used as the initial symbolic values of these variables. Consider, for example, the path $q_{0i_1} \xrightarrow[!c]{} q_{0i_2} \rightarrow q_{0i_3} \rightarrow q_{0i_4}$ as β in figure 3.14. The state q_{1j_1} is the corresponding state of q_{0i_1}. Consider the path $q_{1j_1} \xrightarrow[!c]{} q_{1j_2} \rightarrow q_{1j_3}$ as α. The variable d' ($\notin V_0 \cap V_1$) is used (in the operation $d \Leftarrow d'$) before it is defined along β. Now, the paths $q_{1j_0} \rightarrow q_{1j_1}$ is the only path which terminates in q_{1j_1} and the operation $d' \Leftarrow x + y$ defines d' in that path. So, the expression $x + y$ will be used as the initial symbolic value of d' while computing the condition of execution and the data transformation of α. With this modification, the path α will be found as the equivalent path of β. However, the initial symbolic value obtained for every variable should be unique in all the paths that terminate in q_{1k}. If more than one expression are found for a particular variable or no operation is found which defines a variable in one path, we will ignore α for equivalence checking and consider the next path from q_{1k}.

3.6.9 Loop Shifting and Compaction

Loop shifting [68] is a technique whereby an operation *op* is moved from the beginning of the loop body to the end of the loop body. To preserve the correctness of the program, a copy op_c of the operation *op* is also placed before the start of the loop. Consider the example in figure 3.15. Operations a and c of the original behaviour are shifted to the end of the loop body as well as placed at the entry edge(s) of the loop. The modified FSMD is shown in figure 3.15 (b).

It is important to note that shifting operation(s) in a loop reduces the data dependencies among the operations within the loop body. For example, if we consider the ith iteration of the loop in

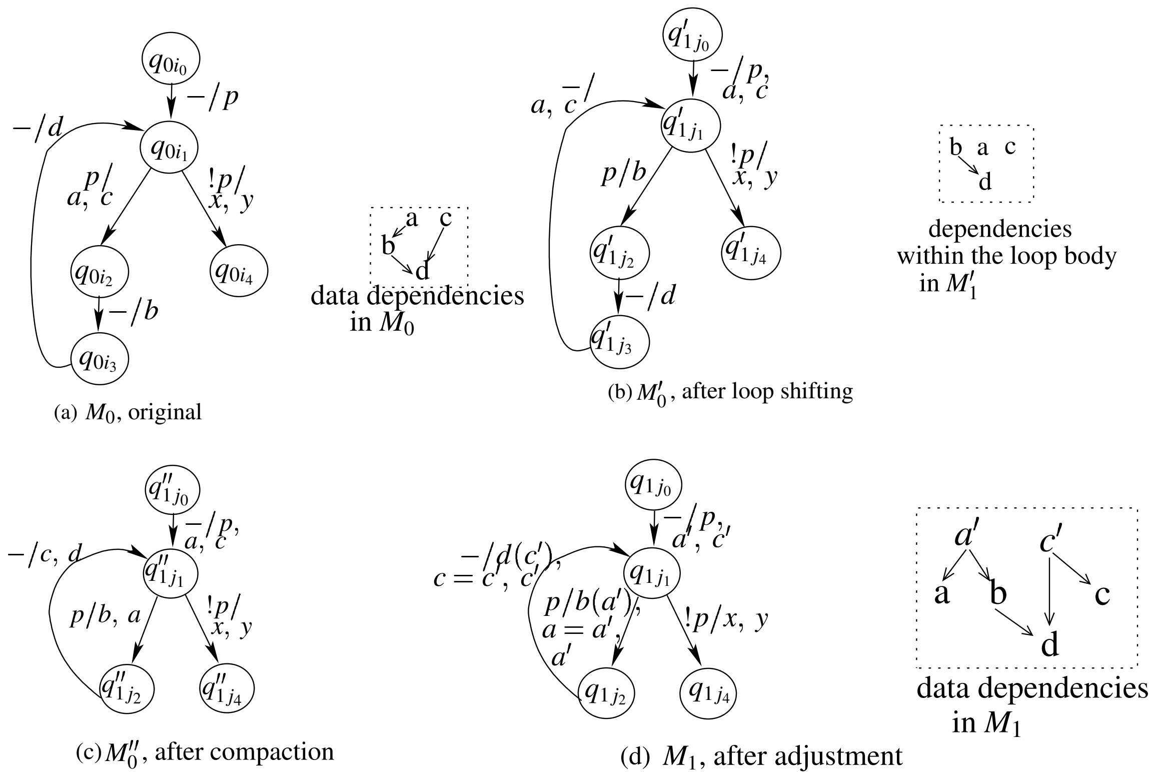


Figure 3.15: Loop shifting and compaction: An example

the original behaviour in the FSMD M_0 of figure 3.15 (a), the value of the variable b depends on a and the value of d depends on c. But, if we consider the ith iteration of the loop in M'_0, then the operations a and c of the loop body represent the operations corresponding to the $(i+1)^{th}$ iteration of M_0. So, the operations b and d in the loop body do not depend on the operations a and c, respectively. The modified data dependencies in the shifted loop is shown in the dotted block in figure 3.15 (b). As a result, the scope of concurrent execution of the operations increases within the loop body. Specifically, it is possible to execute the operations b concurrently with a and the operation c concurrently with d. The compacted FSMD is shown in figure 3.15(c).

Shifting an operation results in execution of the operation one more time than in the original code. For example, if we consider n number of iterations of the loop body, then the operations a and c execute $n+1$ times in the compacted FSMD (figure 3.15 (c)) whereas they would execute n times in the original description (figure 3.15 (a)). It needs to ensure that executing the shifted operation one extra time does not change the behaviour of the program. In order to do so, it is required to perform the following steps. Let the shifted operation be $v \leftarrow f(\ )$. The first step is to create an operation "$w \leftarrow f(\ )$" in place of the shifted operation, where w is a new variable. In the

second step, all the instances of the operation $v \leftarrow f(\,)$ in the loop body in the original behaviour are replaced by two operations"$v \leftarrow w$" and "$w \leftarrow f(\,)$" in parallel. Finally, the operations that use the variable 'v' in the loop body will now use the variable w instead of variable v. It is demonstrated in figure 3.15 (d). The results of the shifted operations a and c are stored in a' and c', respectively. The operation a is replaced by $a \Leftarrow a'$ and a' in the loop body. Thus, by the ith iteration, a' is computed $i+1$ times but a assumes the value of ith iteration. Similarly, the operation c is also replaced. In the original behaviour, the operation b uses the variable a and the operation d uses the variable c. Hence, they will now use the variable a' and c', respectively. In the figure 3.15 (d), $b\ (a')$ indicates that the operation b uses the variable a'. Similarly, $d\ (c')$ indicates that the operation d uses the variable c'.

This situation is similar to the speculation described in the subsection 3.6.8. So, the proposed modification of the algorithm in that subsection can handle this situation. Here, we need to check the equivalence between the FSMDs M_0 in figure 3.15 (a) and M_1 in figure 3.15 (d). For the paths $q_{0i_0} \rightarrow q_{0i_1}$ and $q_{0i_1} \underset{p}{\rightarrow} q_{0i_4}$ in M_0, $q_{1j_0} \rightarrow q_{1j_1}$ and $q_{1j_1} \underset{p}{\rightarrow} q_{1j_4}$, respectively, are ascertained to be the equivalent paths in M_1. Now, while finding the equivalent path of $\beta = q_{0i_1} \underset{p}{\rightarrow} q_{0i_2} \rightarrow q_{0i_3} \rightarrow q_{0i_1}$, it was found that the variables a', c' ($\in V_1 - V_0$) are used before they are defined along $\alpha = q_{1j_1} \underset{p}{\rightarrow} q_{1j_2} \rightarrow q_{1j_1}$. Hence, we need to find the paths that terminate in the state q_{1j_1}. The paths $q_{1j_0} \rightarrow q_{1j_1}$ and $q_{1j_1} \underset{p}{\rightarrow} q_{1j_2} \rightarrow q_{1j_1}$ are such paths. In both the paths, the same operations a' and c' update the variables a' and c', respectively. In fact, if the right hand side of the respective operations is used as the input assertions for a' and c', then α will become the equivalent path of β.

3.7 Conclusions

The verification of the scheduling process is discussed in this chapter. The difficulties of scheduling verification have been identified. The modifications required in our basic equivalence checking method are discussed. A scheduling verification algorithm is proposed. The correctness of the algorithm is proved and the complexity of the algorithm is analyzed. Verification of basic block based and path based scheduling algorithms are discussed next. It is found that our algorithm works for both the cases. Performance of our algorithm for several code motion techniques are discussed. It is shown that our algorithm verifies successfully the code motion techniques like, *renaming, common sub-expression elimination, early condition execution, conditional branch balancing, conditional speculation*. The algorithm, however, fails for the case of *speculation, reverse*

speculation, loop shifting. The modifications of the algorithm are also proposed to handle these code transformation techniques.

Chapter 4

Allocation and Binding Verification

4.1 Introduction

The scheduler assigns time steps to the operations of the input behaviour. The next task is to identify suitable functional units (FUs) and storages units (registers) for the operations and the variables, respectively, of the scheduled behaviour. This task is called allocation and binding. It consists of four subtasks: FU allocation, FU binding, storage allocation and storage binding.

- **FU allocation:** A component library contains multiple types of functional units, each with different characteristics (e.g., functionality, size, delay, etc.) and each implementing one or several different operations. For example, an addition can be carried out by either a simple but slow ripple carry adder or by a more complex but fast carry look-ahead adder. Also, an adder or an adder/subtracter can be used to perform an addition operation. Thus, the functional unit allocation subtask consists in selecting the number and the respective types of different functional units from the component library.

- **FU binding:** After selection of all the functional units, the operations in the behaviour are mapped into the set of selected FUs. This task is called FU binding.

- **Storage allocation:** The minimum number of registers required for storing the variables of the scheduled behaviour is decided by storage allocation. The variables of the scheduled behaviour having non-overlapping lifetimes may share the same register. The lifetime of a variable v is the time interval between its first value assignment (that is, the first appearance

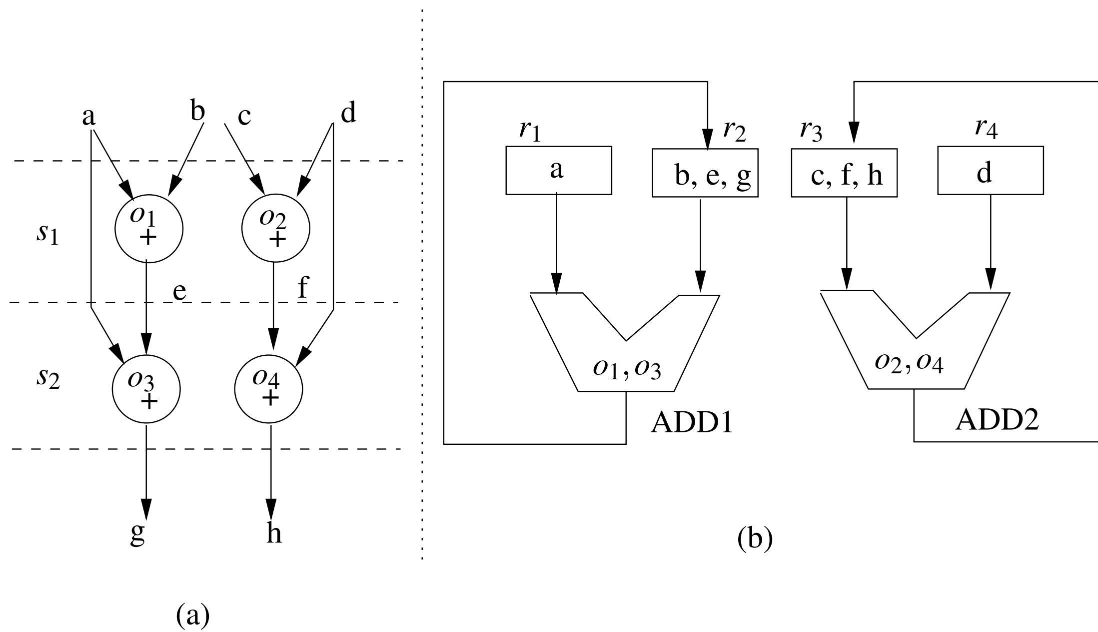


Figure 4.1: An illustration of allocation and binding process: a. Scheduled behaviour b. After allocation and binding

of v on the left-hand side of an assignment statement) and its last use (that is, the last appearance of v on the right-hand side of an assignment statement). The number of registers allocated should be less than or equal to the number of variables in the behaviour.

- **Storage binding:** The variables of the behaviour are mapped to the selected registers by the storage binding subtask.

Example 7 *The allocation and binding process is illustrated in figure 4.1. Let us consider the scheduled behaviour (given as a DFG) of figure 4.1(a). There are four operations* o_1, o_2, o_3, o_4 *in the behaviour and they are scheduled in two time-steps (*s_1 *and* s_2*). Also, there are eight variables* a, b, c, d, e, f, g, h *in the behaviour. Let us assume that two adders* $ADD1$ *and* $ADD2$ *are selected by the FU allocation task. The operations* o_1, o_3 *and* o_2, o_4 *bind to* $ADD1$ *and* $ADD2$*, respectively, as shown in figure 4.1(b). Also, there are only four registers allocated for the eight variables. The variable sets* $\{a\}$, $\{b, e, g\}$, $\{c, f, h\}$ *and* $\{d\}$ *map to registers* r_1, r_2, r_3 *and* r_4*, respectively.* □

4.2 Objectives

The FUs and the registers are shared by the operations and the registers, respectively, of the scheduled behaviour. The objective of this phase of verification is to ensure the correctness of the allocation and the binding process. This is achieved in two steps in this work. In the first step, the correctness of the functional unit allocation and binding is ensured and in the second step, the correctness of register sharing among the behavioural variables is verified.

4.3 Verification of Allocation and Binding of Functional Units

The objective of this step of verification is to ensure that the result of the functional unit allocation and binding task satisfies the following two properties.

- **Enough FUs are allocated:** It needs to be ensured that the number of FUs performing a certain type of operation must be equal to or greater than the maximum number of operations of that type to be performed in any control step. In the example in figure 4.1, at most two addition operations are performed in any control step. So, at least two adders have to be allocated for this example.

- **The operations of each control step are properly mapped to the FUs:** The operations that are scheduled in the same control step cannot be mapped to the same FU. For example, the operations o_1 and o_2 cannot be mapped to the same adder in the example in figure 4.1 because they must be performed in the same control step s_1. Similarly, the operations o_3 and o_4 cannot be mapped to the same adder in this example. It may be noted that the operations o_1 and o_3 can share a single adder because they are carried out in different control steps. Thus, the operations o_1 and o_3 are both mapped to $ADD1$.

These two properties, however, need not be explicitly verified on the FU allocation and binding results. They are automatically accounted for during the *rewriting process* in the data-path and the controller verification phase; this will be discussed in the next chapter.

4.4 Register Sharing Verification

Several variables can be made to share a register if there respective lifetimes do not overlap. Consider the situation in example 7 (figure 4.1). Here, only four registers are used to store eight

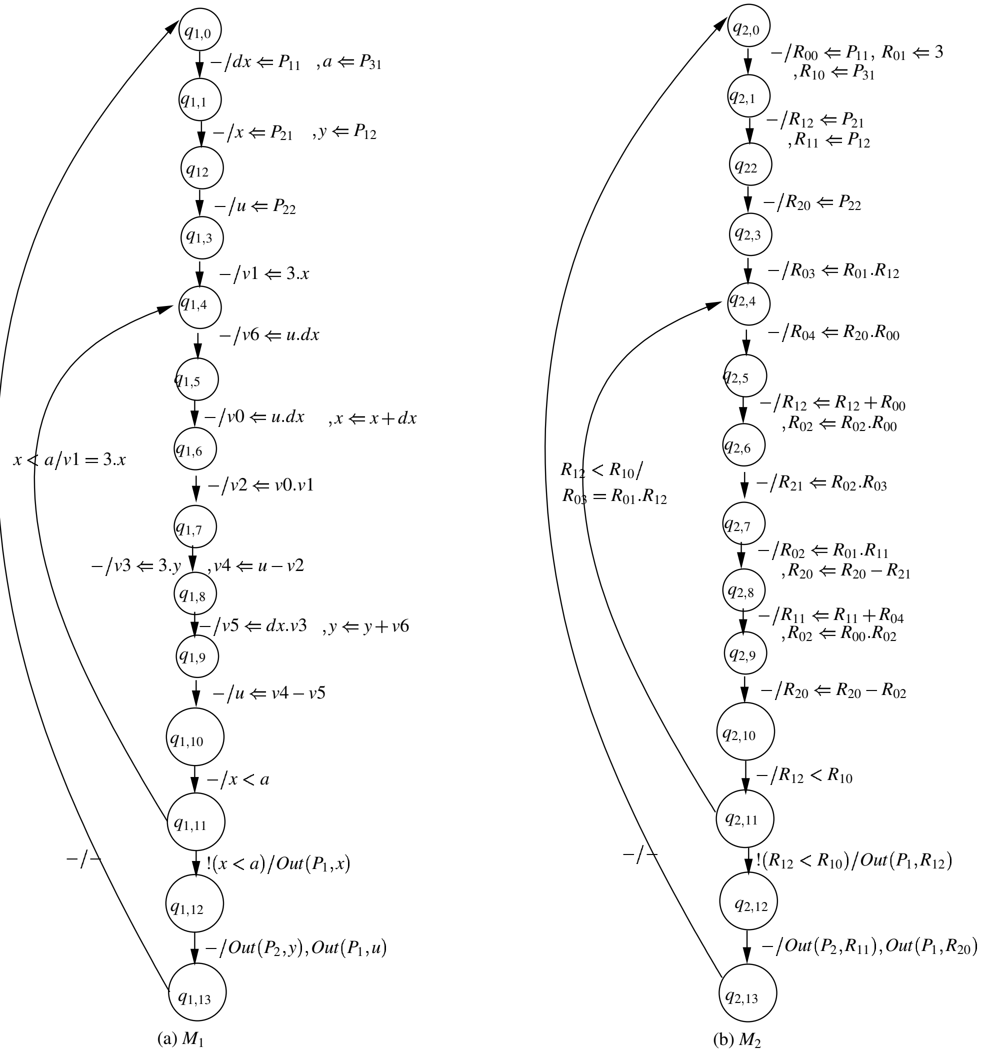


Figure 4.2: DIFFEQ Example: a. FSMD after scheduling b. FSMD after allocation & binding

variables. The objective of this verification step is to ensure that the registers are shared properly among the variables. The equivalence checking method formulated in chapter 2 is useful here. The variable set V_1 of the FSMD M_1 (the scheduled behaviour) and the variable set V_2 of the FSMD M_2 (the behaviour after allocation and binding), however, are different. Hence, the following mapping informations are essential for equivalence checking and are to be provided by the synthesis tool.

4.4.1 The Mapping Functions

It may be recalled that the FSMD corresponding to the scheduled behaviour is denoted as $M_1 = \langle Q_1, q_{10}, I, V_1, O, f_1, h_1 \rangle$ and that corresponding to the output of the allocation and binding phase is

denoted as $M_2 = \langle Q_2, q_{20}, I, V_2, O, f_2, h_2 \rangle$, where V_2 is the set of registers. The following mappings are introduced.

Definition 12 The state mapping function $f_{sm} : Q_1 \leftrightarrow Q_2$.

In the allocation and binding phase, the scheduler output is mapped to the hardware with specific intention of using minimum number of registers, functional units (FUs), muxes, demuxes, etc. Optimization like reduction of total time to execute has already been considered in the scheduling phase. So, the FSMD structure in the output does not change from the input FSMD structure in this phase. Hence, the function f_{sm} is a *bijection*.

Definition 13 Register binding function f_{rb}**:** $Q_2 \times V_2 \rightarrow V_1 \cup \{\perp\}$ *maps the registers at each time step in* M_2 *to variables in* M_1*, i.e., it defines the variable contained in a register at each state of* M_2.

If $f_{rb}(q_{2,i},\ r_j) = v_k \in V_1$, then v_k is said to be *the variable corresponding to the register* r_j and r_j is said to be *the register corresponding to the variable* v_k at the state $q_{2,i}$. The two basic assumptions about the registers considered here are as follows.

1. The registers initially contain some garbage (undefined) values, denoted as $\perp$.

2. Once a value is stored in a register, it continues to hold it until the register has been updated by some other value.

Register	Lifetimes of the variables
R_{00}	$\langle\perp, q_{2,0}, q_{2,0}\rangle, \langle dx, q_{2,1}, q_{2,13}\rangle$
R_{01}	$\langle\perp, q_{2,0}, q_{2,0}\rangle, \langle 3, q_{2,1}, q_{2,13}\rangle$
R_{02}	$\langle\perp, q_{2,0}, q_{2,5}\rangle, \langle v0, q_{2,6}, q_{2,7}\rangle, \langle v3, q_{2,8}, q_{2,8}\rangle, \langle v5, q_{2,9}, q_{2,13}\rangle$
R_{03}	$\langle\perp, q_{2,0}, q_{2,3}\rangle, \langle v1, q_{2,4}, q_{2,13}\rangle$
R_{04}	$\langle\perp, q_{2,0}, q_{2,4}\rangle, \langle v6, q_{2,5}, q_{2,13}\rangle$
R_{10}	$\langle\perp, q_{2,0}, q_{2,0}\rangle, \langle a, q_{2,1}, q_{2,13}\rangle$
R_{11}	$\langle\perp, q_{2,0}, q_{2,1}\rangle, \langle y, q_{2,2}, q_{2,13}\rangle$
R_{12}	$\langle\perp, q_{2,0}, q_{2,1}\rangle, \langle x, q_{2,2}, q_{2,13}\rangle$
R_{20}	$\langle\perp, q_{2,0}, q_{2,2}\rangle, \langle u, q_{2,3}, q_{2,7}\rangle, \langle v4, q_{2,8}, q_{2,9}\rangle, \langle u, q_{2,10}, q_{2,13}\rangle$
R_{21}	$\langle\perp, q_{2,0}, q_{2,6}\rangle, \langle v2, q_{2,7}, q_{2,13}\rangle$

Table 4.1: Mapping of the registers to the variables for DIFFEQ example

The function f_{rb} is total in the sense that any register contains either the value of a variable or the garbage value $\perp$ at each state. The function f_{rb}, however, may not be a bijection as the variables may have non-overlapping lifetimes and accordingly share the same register. Consequently, the number of registers in M_2 is less than or equal to the number of variables in M_1. This mapping function can be constructed from the lifetime information of the variables obtained from the allocation and binding information provided by any high-level synthesis tool. The mapping function f_{rb} produced by our SAST tool for the DIFFEQ example of figure 4.2 is shown in table 4.1. The tuple $\langle v,\ start,\ end \rangle$ for a register R indicates that the value of the variable v is stored in register R from the state 'start' to the state 'end'. For example, the tuple $\langle v0,\ q_{2,6},\ q_{2,7} \rangle$ for the register R_{02} in table 4.1 means that the variable $v0$ remains stored in R_{02} from the state $q_{2,6}$ to the state $q_{2,7}$.

4.4.2 Verification Issues

The control structure of M_2 does not get modified by the allocation and binding process. Hence, the strategy to select cutpoints in an FSMD and the verification method described in chapter 2 can be used in register sharing verification. We, however, need to modify the definition of equivalence of paths as the FSMDs M_1 and M_2 involve different variable sets with different cardinalities. The modification is as follows.

Given two FSMDs $M_1 = \langle Q_1,\ q_{1,0},\ I,\ V_1,\ O,\ f_1,\ h_1 \rangle$ and $M_2 = \langle Q_2,\ q_{2,0},\ I,\ V_2,\ O,\ f_2,\ h_2 \rangle$ and the mapping functions f_{sm} and f_{rb} as defined in the previous section, the equivalence between an expression of M_1 and an expression of M_2 is defined as follows.

Let the expression e_1 over $V_1 \cup I$ at a state $q_{1,i}$ of the FSMD M_1 be denoted as $e_1|_{q_{1,i}}$; similarly, an expression e_2 over $V_2 \cup I$ at a state $q_{2,j}$ of the FSMD M_2 be denoted as $e_2|_{q_{2,j}}$. Let $\{v_i/r_i,\ 1 \leq i \leq l\}$ denote a substitution, where v_i is a variable of the set V_1 and r_i is a register of the set V_2. For any expression e over V_2, let the symbol $e\{v_i/r_i,\ 1 \leq i \leq l\}$ denote the expression obtained by simultaneous substitution of v_i for all occurrences of r_i in e, $1 \leq i \leq l$.

The expression $e_2|_{q_{2,j}}$ having occurrences of $r_1,\ \ldots,\ r_l$ from V_2 is computationally equivalent to $e_1|_{q_{1,i}}$ if $f_{sm}(q_{1,i}) = q_{2,j}$ and $e_2|_{q_{2,j}}\{f(q_{2,j},\ r_k)/r_k,\ 1 \leq k \leq l\} = e_1|_{q_{1.i}}$, where '$=$' stands for syntactic identity. This computational equivalence is denoted as $e_2|_{q_{2,j}} \simeq e_1|_{q_{1,i}}$. [1]

Let, $\alpha_1 = \langle q_{1,p} \Rightarrow q_{1,m} \rangle$ and $\alpha_2 = \langle q_{2,r} \Rightarrow q_{2,s} \rangle$ be two paths in M_1 and M_2, respectively. Let there be n variables in the behavioural specification (M_1) and k registers in the data-path (i.e., in the

[1] Computational equivalence is in general denoted by the symbol $\simeq$ in this work.

FSMD M_2) for the given problem. Let the conditions be $R_{\alpha_1} = c_{11} \wedge c_{12} \wedge \ldots \wedge c_{1x}$ and $R_{\alpha_2} = c_{21} \wedge c_{22} \wedge \ldots \wedge c_{2x}$ and the data transformations be $r_{\alpha_1} = \langle s_{\alpha_1}, O_{\alpha_1} \rangle$ and $r_{\alpha_2} = \langle s_{\alpha_2}, O_{\alpha_2} \rangle$. In particular, note that the ordered tuple $s_{\alpha_1} = \langle e_{11}, e_{12}, \cdots, e_{1n} \rangle$, where each $e_{1i}, 1 \leq i \leq n$, is an expression over $I \cup V_1$ representing the value of the variable v_i after the execution of the path α_1 in M_1 in terms of the initial data state of the path. Similarly, $s_{\alpha_2} = \langle e_{21}, e_{22}, \cdots, e_{2k} \rangle$, where each $e_{2i}, 1 \leq i \leq k$, is an expression over $I \cup V_2$ representing the value of register r_i after the execution of the path α_2 in M_2 in terms of the initial data state of the path. Let the output list of the path α_1 be $O_{\alpha_1} = [OUT(P_{y_1}, e_{11}), OUT(P_{y_2}, e_{12}), \ldots, OUT(P_{y_u}, e_{1u})]$; it is possible that $P_{y_j} = P_{y_k}$ for some $j \neq k$. Let the output list of the path α_2 be $O_{\alpha_2} = [OUT(P_{z_1}, e_{21}), OUT(P_{z_2}, e_{22}), \ldots, OUT(P_{z_v}, e_{2v})]$.

The condition R_{α_1} is equivalent to R_{α_2}, denoted as $R_{\alpha_1} \simeq R_{\alpha_2}$, if $c_{1i}|_{q_{1,m}} \simeq c_{2i}|_{q_{2,s}}, \forall i, 1 \leq i \leq x$. The simple data transformations s_{α_1} and s_{α_2} are equivalent, denoted as $s_{\alpha_1} \simeq s_{\alpha_2}$, if $\forall i, 1 \leq i \leq k, \exists j, 1 \leq j \leq n$ $s.t$ $v_j = f_{rb}(q_{2,s}\ r_i) \wedge e_{1j}|_{q_{1,m}} \simeq e_{2i}|_{q_{2,s}}$. The output lists O_{α_1} and O_{α_2} are equivalent if (i) $u = v$, (ii) $P_{y_i} = P_{z_i}$, $1 \leq i \leq u$ and (iii) $e_{1i}|_{q_{1,m}} \simeq e_{2i}|_{q_{2,s}}$, $1 \leq i \leq u$. Now, the data transformations r_{α_1} and r_{α_2} are equivalent, if $s_{\alpha_1} \simeq s_{\alpha_2}$ and $O_{\alpha_1} \simeq O_{\alpha_2}$. Hence, the path α_1 is said to be equivalent to the path α_2 if $R_{\alpha_1} \simeq R_{\alpha_2}$ and $r_{\alpha_1} \simeq r_{\alpha_2}$.

4.4.3 Verification Algorithm

We have already discussed in subsection 4.4.1 that the structures of both M_1 and M_2 would be the same and have equal number of states. As a result, the number of cutpoints would be equal in M_1 and M_2; also the path covers P_1 and P_2 of M_1 and M_2, respectively, have the same number of paths. For any given path α from P_1 of the form $\langle q_{1,1} \xrightarrow[s_1]{} q_{1,2} \xrightarrow[s_2]{}, \cdots, \xrightarrow[s_{n-1}]{} q_{1,n} \rangle$, there may exist a set P_2' of more than one path from P_2 of the form $\langle q_{2,1} \xrightarrow[s_1']{} q_{2,2} \xrightarrow[s_2']{}, \cdots, \xrightarrow[s_{n-1}']{} q_{2,n} \rangle$ such that $\forall i,\ 1 \leq i \leq n,\ q_{2,i} = f_{sm}(q_{1,i})$ as there may be parallel edges between two states. The path equivalent to α is to be found from the set P_2'. The verification algorithm is given as algorithm 1.

4.4.4 An Example

In this subsection, how the verification of register sharing works is described with the DIFFEQ example. The scheduled FSMD M_1 and the corresponding FSMD M_2 after allocation and binding for the DIFFEQ example are shown in figure 4.2. The mapping function for this example is shown in table 4.1.

According to our definition, the cutpoints in M_1 are $q_{1,0}$ and $q_{1,11}$. Similarly, the cutpoints in

Algorithm 1 Verification algorithm for allocation and binding phase

Input: The FSMDs M_1, M_2 and the mapping functions f_{sm}, f_{rb}.
Output: The 'yes/no' answer for "M_1 is equivalent to M_2".
Method:
Insert cutpoints in M_1 and in M_2.
Find the path covers P_1 and P_2, where P_1 is the set of paths of M_1 and P_2 is the set of paths of M_2 and each path spans from a cutpoint to a cutpoint with no intermediary cutpoint;

```
∀α ∈ P1
do
    P2' = getPath (α, P2);
    /* This function returns a set of paths from P2 corresponding to the path α */
    for each β ∈ P2'
    begin
            match = checkEquivalent(α, β, frb);
            /* This function returns 1, if α ≃ β; 0, otherwise */
            if (match) break;
    end for;
    if (!match) Report:"M1 and M2 are not equivalent"; exit;
end do;
Report: "M1 and M2 are equivalent";
```

Function : $checkEquivalent(\alpha,\ \beta,\ f_{rb})$

```
/* This function checks the equivalence between the path α of M1 and the path β of M2*/
begin
        notEquivalent = 0;
        compute(Rβ, rβ, β);
        compute(Rα, rα, α);
        /* compute functions compute the condition of execution and the data transformations
           of a path */
        if (!Rα ≃ Rβ)
           notEquivalent=1;
        if (!sα ≃ sβ)
           notEquivalent=1;
        if (!Oα ≃ Oβ) notEquivalent = 1;
        if (notEquivalent = 0) return 1;
        else return 0;
end
```

State	dx	x	y	a	u	v0	v1	v2	v3	v4	v5	v6
$q_{1,0}$	–	–	–	–	–	–	–	–	–	–	–	–
$q_{1,1}$	P_{11}	–	–	P_{31}	–	–	–	–	–	–	–	–
$q_{1,2}$	P_{11}	P_{21}	P_{12}	P_{31}	–	–	–	–	–	–	–	–
$q_{1,3}$	P_{11}	P_{21}	P_{12}	P_{31}	P_{22}	–	–	–	–	–	–	–
$q_{1,4}$	P_{11}	P_{21}	P_{12}	P_{31}	P_{22}	–	$3.P_{21}$	–	–	–	–	–
$q_{1,5}$	P_{11}	P_{21}	P_{12}	P_{31}	P_{22}	–	$3.P_{21}$	–	–	–	–	$P_{22}.P_{11}$
$q_{1,6}$	P_{11}	$P_{21}+P_{11}$	P_{12}	P_{31}	P_{22}	$P_{22}.P_{11}$	$3.P_{21}$	–	–	–	–	$P_{22}.P_{11}$
$q_{1,7}$	P_{11}	$P_{21}+P_{11}$	P_{12}	P_{31}	P_{22}	$P_{22}.P_{11}$	$3.P_{21}$	$P_{22}.P_{11}.3.P_{21}$	–	–	–	$P_{22}.P_{11}$
$q_{1,8}$	P_{11}	$P_{21}+P_{11}$	P_{12}	P_{31}	P_{22}	$P_{22}.P_{11}$	$3.P_{21}$	$P_{22}.P_{11}.3.P_{21}$	$3.P_{12}$	$(P_{22}-P_{22}.P_{11}.3.P_{21})$	–	$P_{22}.P_{11}$
$q_{1,9}$	P_{11}	$P_{21}+P_{11}$	$(P_{12}+P_{22}.P_{11})$	P_{31}	P_{22}	$P_{22}.P_{11}$	$3.P_{21}$	$P_{22}.P_{11}.3.P_{21}$	$3.P_{12}$	$(P_{22}-P_{22}.P_{11}.3.P_{21})$	$P_{11}.3.P_{12}$	$P_{22}.P_{11}$
$q_{1,10}$	P_{11}	$P_{21}+P_{11}$	$(P_{12}+P_{22}.P_{11})$	P_{31}	$(P_{22}-P_{22}.P_{11}.3.P_{21}-P_{11}.3.P_{12})$	$P_{22}.P_{11}$	$3.P_{21}$	$P_{22}.P_{11}.3.P_{21}$	$3.P_{12}$	$(P_{22}-P_{22}.P_{11}.3.P_{21})$	$P_{11}.3.P_{12}$	$P_{22}.P_{11}$
$q_{1,11}$	P_{11}	$P_{21}+P_{11}$	$(P_{12}+P_{22}.P_{11})$	P_{31}	$(P_{22}-P_{22}.P_{11}.3.P_{21}-P_{11}.3.P_{12})$	$P_{22}.P_{11}$	$3.P_{21}$	$P_{22}.P_{11}.3.P_{21}$	$3.P_{12}$	$(P_{22}-P_{22}.P_{11}.3.P_{21})$	$P_{11}.3.P_{12}$	$P_{22}.P_{11}$

Table 4.2: Computation of data transformation of the path $q_{1,0} \Rightarrow q_{1,11}$ in M_1

M_2 are $q_{2,0}$ and $q_{2,11}$. The paths from one cutpoint to another without traversing through another cutpoint in M_1 are $\langle q_{1,0} \Rightarrow q_{1,11}\rangle$, $\langle q_{1,11} \Rightarrow q_{1,11}\rangle$ and $\langle q_{1,11} \Rightarrow q_{1,0}\rangle$ and the corresponding paths in M_2 are $\langle q_{2,0} \Rightarrow q_{2,11}\rangle$, $\langle q_{2,11} \Rightarrow q_{2,11}\rangle$ and $\langle q_{2,11} \Rightarrow q_{2,0}\rangle$, respectively. The equivalence of every corresponding pair of paths would be checked one-by-one by our verification algorithm. Let us consider the pair $\alpha = \langle q_{1,0} \Rightarrow q_{1,11}\rangle$ and $\beta = \langle q_{2,0} \Rightarrow q_{2,11}\rangle$. The condition of execution is *true* for both the paths. The data transformation for α is shown in (the last row of table) table 4.2. The same for the path β is shown in (the last row) of table 4.3. The algorithm compares the final value of a register, r say, at the state $q_{2,11}$ with the final value of the variable $f_{rb}(q_{2,11},\ r)$ at the state $f_{sm}^{-1}(q_{2,11}) = q_{1,11}$. For example, the final value of the register R_{20} at $q_{2,11}$ and that of the variable $u = f_{rb}(q_{2,11},\ R_{20})$ at $q_{1,11}$ (obtained from the mapping information in table 4.1) is equal and is $(P_{22} - P_{22}.P_{11}.3.P_{21} - P_{11}.3.P_{12})$. Similarly, the final values of all the registers and those of the corresponding variables can be shown to be equal. Hence, these two corresponding paths are equivalent. The equivalence of other corresponding pairs of paths can be shown in a similar manner. Hence, for the DIFFEQ example, the FSMD M_1 is adjudged to be equivalent to the FSMD M_2 by the algorithm.

4.4.5 Performance of the Algorithm

There are two types of register optimization schemes commonly found in high-level synthesis tools. They are *carrier based* [72] and *value based* [73]. If two or more variables have non-overlapping lifetimes, then they are mapped to the same register in the carrier based register optimization scheme. In the value based approach, the register optimization is modeled as the problem

State	R_{00}	R_{01}	R_{02}	R_{03}	R_{04}	R_{10}	R_{11}	R_{12}	R_{20}	R_{21}
$q_{2,0}$	–	3	–	–	–	–	–	–	–	–
$q_{2,1}$	P_{11}	3	–	–	–	P_{31}	–	–	–	–
$q_{2,2}$	P_{11}	3	–	–	–	P_{31}	P_{12}	P_{21}	–	–
$q_{2,3}$	P_{11}	3	–	–	–	P_{31}	P_{12}	P_{21}	P_{22}	–
$q_{2,4}$	P_{11}	3	–	$3.P_{21}$	–	P_{31}	P_{12}	P_{21}	P_{22}	–
q_{25}	P_{11}	3	–	$3.P_{21}$	$P_{22}.P_{11}$	P_{31}	P_{12}	P_{21}	P_{22}	–
$q_{2,6}$	P_{11}	3	$P_{22}.P_{11}$	$3.P_{21}$	$P_{22}.P_{11}$	P_{31}	P_{12}	$P_{21}+P_{11}$	P_{22}	–
$q_{2,7}$	P_{11}	3	$P_{22}.P_{11}$	$3.P_{21}$	$P_{22}.P_{11}$	P_{31}	P_{12}	$P_{21}+P_{11}$	P_{22}	$P_{22}.P_{11}.3.P_{21}$
$q_{2,8}$	P_{11}	3	$3.P_{12}$	$3.P_{21}$	$P_{22}.P_{11}$	P_{31}	P_{12}	$P_{21}+P_{11}$	$P_{22}-P_{22}.P_{11}.3.P_{21}$	$P_{22}.P_{11}.3.P_{21}$
$q_{2,9}$	P_{11}	3	$P_{11}.3.P_{12}$	$3.P_{21}$	$P_{22}.P_{11}$	P_{31}	$P_{12}+P_{22}.P_{11}$	$P_{21}+P_{11}$	$P_{22}-P_{22}.P_{11}.3.P_{21}$	$P_{22}.P_{11}.3.P_{21}$
$q_{2,10}$	P_{11}	3	$P_{11}.3.P_{12}$	$3.P_{21}$	$P_{22}.P_{11}$	P_{31}	$P_{12}+P_{22}.P_{11}$	$P_{21}+P_{11}$	$(P_{22}-P_{22}.P_{11}.3.P_{21}$ $-P_{11}.3.P_{12})$	$P_{22}.P_{11}.3.P_{21}$
$q_{2,11}$	P_{11}	3	$P_{11}.3.P_{12}$	$3.P_{21}$	$P_{22}.P_{11}$	P_{31}	$P_{12}+P_{22}.P_{11}$	$P_{21}+P_{11}$	$(P_{22}-P_{22}.P_{11}.3.P_{21}$ $-P_{11}.3.P_{12})$	$P_{22}.P_{11}.3.P_{21}$

Table 4.3: Computation of data transformation of the path $q_{2,0} \Rightarrow q_{2,11}$ in M_2

of mapping the data values, produced and used by the operations, into registers. Register optimization is possible when two or more operations use the same data value and the life spans of the two values do not overlap.

The input behaviours are either data-intensive or control-intensive in nature. Symbolic model checking [45] is suitable for formal verification of control-dominated applications. For the control intensive behaviours, the control flow is dependent on the arithmetic bit vector operations; an efficient representation of the transition behaviour under such situations is difficult to obtain due to the state space explosion problem [74]. The data intensive descriptions can be verified by means of symbolic simulation [75]. This method, however, allows only reasoning for a finite number of steps. More specifically, the loops in the description cannot be verified for an arbitrary number of iterations [74].

In this subsection, we analyze these two important issues and show how our algorithm works for these cases.

Register Optimization Schemes

In the carrier based approach, two or more variables share a register if their respective lifetimes do not overlap. One variable always maps to only one register. Therefore, the mapping from the specification variables to the registers is a many-to-one relation. In the value based approach, two or more variables are assigned the same register if they use the same data value or the life span of at least one data value used by each variable is non-overlapping to each other. It is obvious that a variable during its lifetime may assume different values. Also, it is possible that the same value is assigned to different variables. So, the association of specification variables and the registers is a many-to-many relation.

In both these cases, at any state, each register must contain the value of only one variable and this variable is called the variable corresponding to the register, in question, in that state. Also, one variable is mapped to only one register at each state. During equivalence checking of two corresponding paths, our algorithm compares the value of each register at the end state of that path with the value of corresponding variable. Hence, the algorithm is independent of the schemes used for register optimization.

Nature of the Input Specification

Our algorithm is also independent of the nature of the input specification. The control intensive behaviours are decomposed into path segments by inserting cutpoints in all the branch states. This set of path segments constitutes a path cover of the FSMD. Each path segment is then checked for equivalence with the corresponding path segment in M_2. Equivalence of all the corresponding paths of the two FSMDs implies that for any computation of one FSMD, there is an equivalent computation in the other FSMD. It means that for all possible executions, the registers are shared properly. On the other hand, data-intensive specification, in general, have only one path in each FSMD. Equivalence between these two paths proves the correctness of register sharing.

4.5 Conclusions

The verification of allocation and binding phase is discussed in this chapter. The verification task is achieved in two steps. In the first step, the verification of FU allocation and binding is treated and in the second step, correctness of register sharing among the behavioural variable is verified. The correctness of the FU allocation and binding is defined in this work by two properties which are automatically accounted for during the rewriting process used in the subsequent verification phase. The verification of register sharing is done using the equivalence checking method proposed in this chapter. It, however, needs some additional information like mapping between the states of two FSMDs and the mapping between the registers and the variables in each state. Also, definition of equivalence of paths between two FSMDs has been needed to be redefined. It is shown that our register sharing verification method is independent of the nature of the input specification and the register optimization schemes.

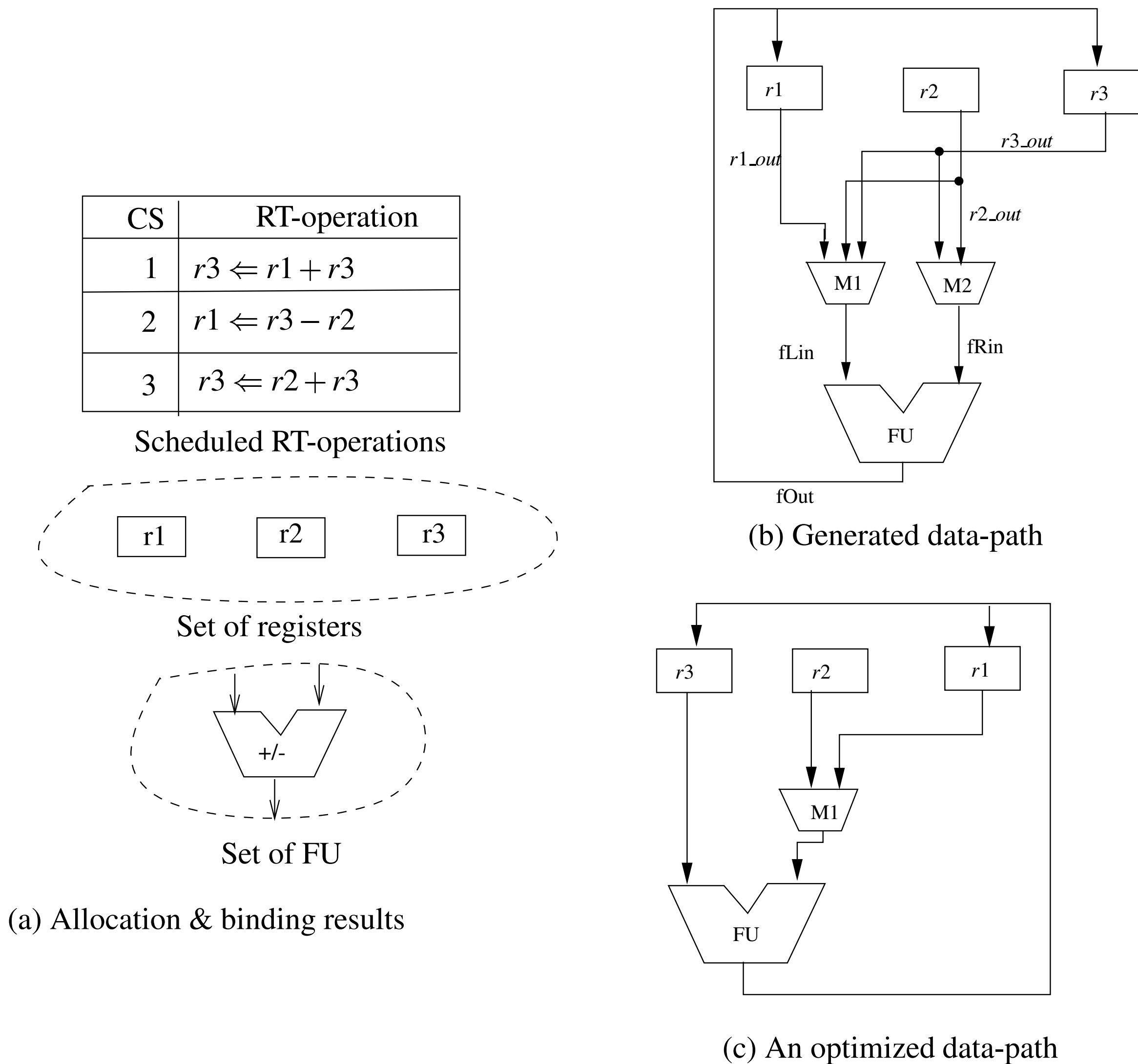
Chapter 5

Data-path and Controller Verification

5.1 Introduction

The allocation and binding process binds the variables to a set of registers and the operations to a set of functional units (FUs) in each control step. The next task is to set the data-path by providing a proper interconnection path from the source(s) to the destination for every register transfer (RT) operation. This process of interconnection generation is called *data-path generation*. The objective of this step is to maximize the sharing of interconnection units and hence minimize the interconnection cost, while supporting the conflict-free data transfers required by the RT-operations. The data-path generation task, in general, consists in identifying the scope of sharing of interconnection paths among data transfer operations which do not take place simultaneously.

Example 8 *Let us consider the situation depicted in figure 5.1. Let there be three RT-operations scheduled in three control steps (CS) involving three registers and one FU (addition/subtraction) with the allocation and binding information as shown in figure 5.1(a). As there is only one FU, all the operations are bound to that. The data-path for the example is shown in 5.1(b). Let the registers* $r1$, $r2$, *and* $r3$ *(the left operands of the three RT-operations) be assigned to the left input of the FU and* $r2$ *and* $r3$ *be assigned to the right input of the FU. Hence, a multiplexer* $M1$ *is used for the left input* $fLin$ *and the multiplexer* $M2$ *is used for the right input* $fRin$ *of the FU. Also, the FU output is stored in the register* $r1$ *in control step* 2 *and in the register* $r3$ *in the control steps* 1 *and* 3. *As such, any data transfer from an FU output to more than one register can share a common bus even if they take place simultaneously. Accordingly, all the three transfers from the*



CS	RT-operation
1	$r3 \Leftarrow r1 + r3$
2	$r1 \Leftarrow r3 - r2$
3	$r3 \Leftarrow r2 + r3$

Figure 5.1: Data-path generation: An example

FU output to the registers are performed using a single bus ($fOut$). It may be noted that the data-path shown in figure 5.1(c) provides for the above three operations incurring less cost. Minimizing such interconnection cost is what is accomplished during data-path generation. The data-path shown in figure 5.1(b) is, however, considered as the running example in the rest of this chapter to show different aspects of data-path and controller verification. □

The minimum number of control signals required to control all the data transfers in each control step is found next. Then, the functionality of each control signal is defined. Finally, the control assertion pattern needed in each control step is found. These processes are collectively called *controller generation*. The controller, represented as an FSM, assigns a value to each control

signal, that is, a control assertion pattern, in each control step to execute all the required data-transfers and proper operations in the FUs. As a result, a set of arithmetic operations as well as a set of relational operations are performed in the data-path. The results of the relational operations are stored in single bit registers whose outputs (status signal) are inputs to the controller. The state transitions in the controller FSM depend on these status signals. Finally, a high-level synthesis (HLS) tool produces an RTL with distinct control-path and data-path (CP-DP). The schematic of the RTL produced by any HLS tool is shown in figure 5.2.

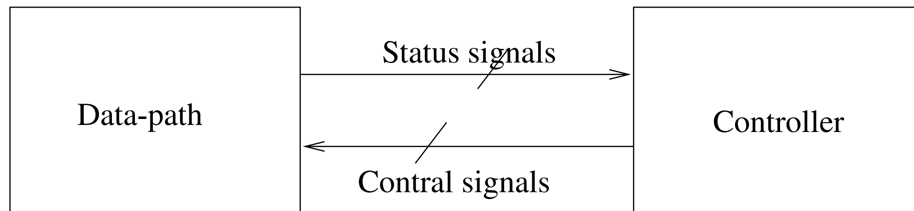


Figure 5.2: The structure of the RTL description produced by any HLS tool

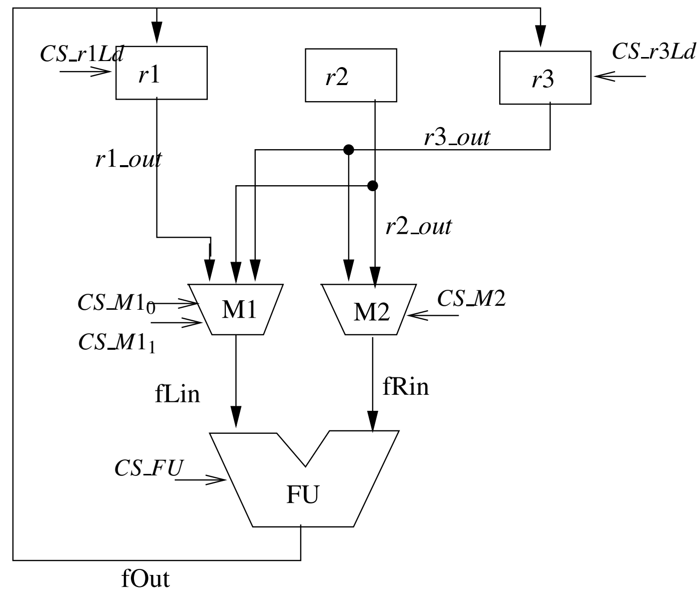


Figure 5.3: Data-path with control signals

Example 9 *The data-path of figure 5.1(b) is redrawn as figure 5.3 with the control signals. Six control signals are required for this data-path. The functionalities of the control signals are as follows:*

$fLin \Leftarrow r1_out \ : CS_M1_1 = 0 \wedge CS_M1_0 = 0$

$fLin \Leftarrow r2_out \ : CS_M1_1 = 0 \wedge CS_M1_0 = 1$

$fLin \Leftarrow r3_out \ : CS_M1_1 = 1$

$fRin \Leftarrow r2_out \ : CS_M2 = 1$

$fRin \Leftarrow r3_out \ : CS_M2 = 0$

$fOut \Leftarrow fLin + fLin \ : CS_FU = 0$

$fOut \Leftarrow fLin - fRin \ : CS_FU = 1$

$r1 \Leftarrow fOut \ : CS_r1Ld = 1$

$r3 \Leftarrow fOut \ : CS_r3Ld = 1.$

The interpretation of the statement $fLin \Leftarrow r1_out \ : CS_M1_1 = 0 \wedge CS_M1_0 = 0$ *is as follows: "if* $CS_M1_1 = 0$ *and* $CS_M1_0 = 0$*, then the micro-operation* $fLin \Leftarrow r1_out$ *occurs in the data-path". Other statements are interpreted likewise.* □

5.2 Verification Goal

The objective of this phase of verification is to ensure the correctness of both the data-path interconnections and the controller FSM. The goals are accomplished in two steps as shown in figure 5.4. First, the FSMD M_3 is constructed from the data-path interconnection informations and the controller FSM. Several inconsistencies, both in the data-path and the controller, are revealed during construction of the FSMD M_3. In the next step, equivalence between the FSMDs M_2 and M_3 is established to verify the correctness of the controller. For this phase of synthesis, a *state based* equivalence checker suffices in contrast to the *path-based* one used for the previous phases [1].

5.3 Construction of the FSMD M_3

The final output of the high-level synthesis is a control-path and a data-path (CP-DP) so that (i) all the register transfer operations and the status condition checking operations in M_2 are indeed provided for by the DP components and their interconnections and (ii) the set of control signal assertions generated and status signals sensed in the states of the controller FSMD M_3 do realize the corresponding register transfers of the FSMD M_2.

[1] In a state based equivalence checker, the equivalence is established between the states of the two FSMDs; whereas, in a path based equivalence checker, the equivalence is established between the paths of the two FSMDs.

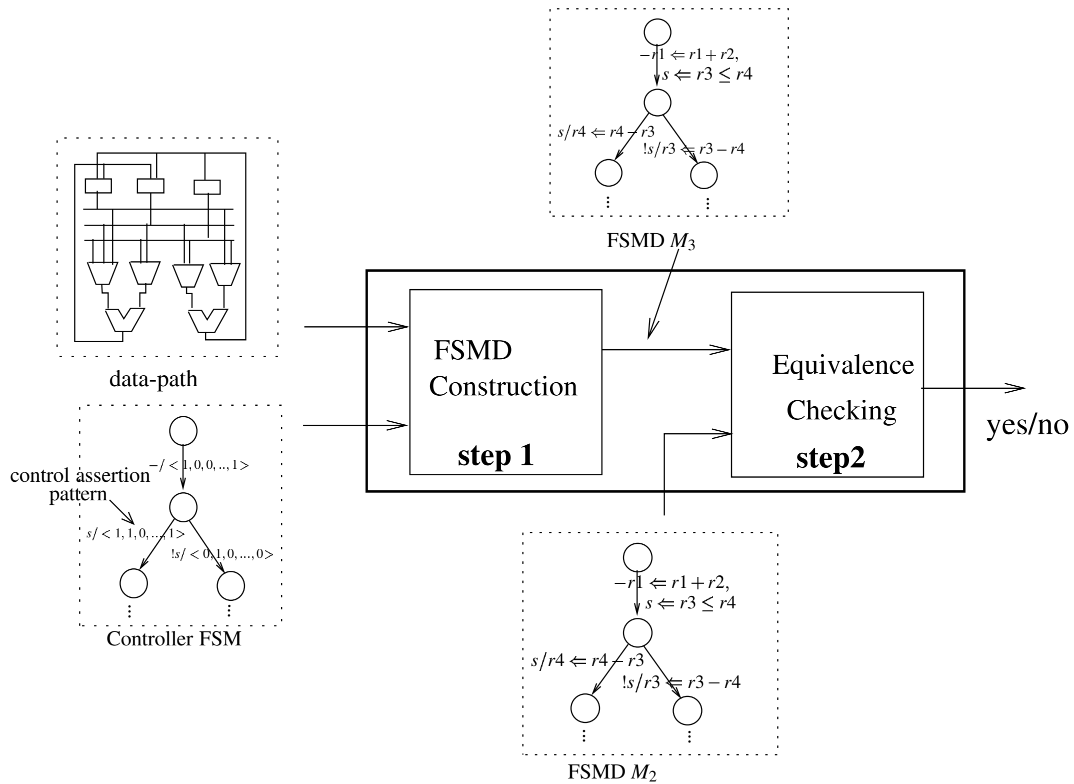


Figure 5.4: The steps of data-path and controller verification

Construction of the FSMD M_3 consists of of following steps:

- Analyze the data-path vis-a-vis the control signal assertion pattern in each state of the controller FSM to construct the RT-operations in that state.
- Replicate the control flow of the controller FSM for the FSMD M_3.

The following two informations have to be extracted from the CP-DP description in order to find the register transfer (RT)-operations in each state of the FSMD M_3.

1. *The set of all possible micro-operations in the data-path.* Let this set be denoted as $\mathscr{M}$. A data movement from a data-path component y to another data-path component x is encoded by the micro-operation $x \Leftarrow y$. The data-path components essentially are storage elements (registers), the functional units and the interconnection components (buses, muxes, de-muxes, switches, etc.).
2. *The control signal assertion pattern for every micro-operation in $\mathscr{M}$.* A control signal assertion pattern needed for any micro-operation is represented as an ordered n-tuple of the form

$\langle u_1,\ u_2,\ldots,\ u_n\rangle$, where u_i represents the value of the control signal c_i and n is the number of control signals; $u_i \in \{0,\ 1,\ X\},\ 1 \leq i \leq n$, is the asserted value of c_i. $u_i = X$ implies that the control signal c_i is not required (relevant) for a particular micro-operation. Let $\mathscr{A}$ be the set of all possible control assertion patterns. So, a function f_{mc} is constructed from the set $\mathscr{M}$ of all micro-operations possible in the given data-path to the set $\mathscr{A}$ of control signal assertion patterns. The DP interconnection is achieved by common signal naming. In order words, if a DP component's output line is connected to another component's input line, then they have the same signal name. Thus, *the data-path structure, in its entirety, is captured by the function f_{mc}*. Obviously, all the elements of the set $\mathscr{A}$ are not involved in the function f_{mc}. Only t number of assertion patterns (one for each micro-operation) constitute the range of f_{mc}, where t is the umber of micro-operations possible in the DP. In other words, the function $f_{mc}\ :\ \mathscr{M}\ \rightarrow\ \mathscr{A}$ is a one-one (injective) mapping but not an onto (surjective) mapping.

In each state of the FSM, the controller generates a control signal assertion pattern to execute a set of micro-operations in the data-path to accomplish a set of register transfer operations concurrently. So, the next task is to obtain the set of micro-operations $\mathscr{M}_A$ ($\subseteq \mathscr{M}$) for a given control assertion pattern A. It is, however, not possible to obtain the set $\mathscr{M}_A$ of micro-operations directly from the control signal assertion pattern A by examining its individual control signals because a micro-operation may be accomplished by a set of control signals rather than an individual control signal. There is no information available in an assertion pattern to group the control signals so that each group defines a micro-operation around a data-path component. The following definition is in order.

Definition 14 Superposition of Assertion patterns:

Let A_1 and A_2 be two arbitrary control signal assertion patterns. Let $\pi_i(A)$ denote the i-th projection of an assertion pattern A which is the asserted value u_i of the control signal c_i. The assertion pattern, $A_1\ \theta\ A_2$, obtained by superposition θ of A_1 and A_2, satisfies the following conditions. For all i,

$$\begin{aligned}\pi_i(A_1\ \theta\ A_2) &= \pi_i(A_1),\ for\ \pi_i(A_1) = \pi_i(A_2)\\ &= \pi_i(A_1),\ for\ \pi_i(A_1) \neq \pi_i(A_2)\ and\ \pi_i(A_1) = X\\ &= undefined\ (U),\ for\ \pi_i(A_1) \neq \pi_i(A_2)\ and\ \pi_i(A_1) \neq X.\end{aligned}$$

Using the above definition and the function f_{mc}, it is possible to construct $\mathscr{M}_A$ from the assertion pattern A by the following definition of $\mathscr{M}_A$: $\mathscr{M}_A = \{\mu_i \mid f_{mc}(\mu_i)\ \theta\ A = f_{mc}(\mu_i)\}$. Let us

consider the superposition of the assertion pattern for a micro-operation μ and a given control assertion pattern A. It may be noted that each bit in A is either '1' or '0' and does not contain any 'X' as it is generated by the controller circuit and the output of any circuit can be either '0' or '1'. On the other hand, the ordered tuple that represents $f_{mc}(\mu)$ may contain 'X' in its ith position if the ith control signal is not involved in μ. If any position i in the ordered tuple $f_{mc}(\mu)$ contains 'X', i.e., $\pi_i(f_{mc}(\mu)) = X$, then the corresponding position in $f_{mc}(\mu)\ \theta\ A$, that is, $\pi_i(f_{mc}(\mu)\ \theta\ A)$, will also contain X. Now, consider a position j in $f_{mc}(\mu)$ which contains 0 or 1. If μ is executed by A, then the corresponding position in the ordered tuple A should contain the same value, that is, $\pi_j(f_{mc}(\mu)) = \pi_j(A)$. As a result, $f_{mc}(\mu)\ \theta\ A$ becomes $f_{mc}(\mu)$ if it is performed by the assertion pattern A. The superposition of the assertion pattern of each micro-operation in $\mathscr{M}$ and A will be checked one by one to select each member of $\mathscr{M}_A$.

Each RT operation that appears in the RTL behaviour is accomplished by a set of concurrent micro-operations. For example, an RT-operation $r_3 \Leftarrow r_1 + r_2$ may be accomplished over the data-path in figure 5.1(b) by the concurrent micro-operations $r1_out \Leftarrow r_1$, $r2_out \Leftarrow r_2$, $fLin \Leftarrow r1_out$, $fRin \Leftarrow r2_out$, $fOut \Leftarrow fLin + fRin$, $r3 \Leftarrow fOut$. It is assumed that the FU performs addition operation on its two input data. So, in order to find the concurrent RT-operations accomplished by a control assertion pattern, it is necessary to find the operations realized by the set $\mathscr{M}$ of concurrent micro-operations.

Finding an RT-operation from a given set of micro-operations is also not trivial because of two reasons. First, there may be more than one RT-operation in that particular state of the FSM. Secondly, there is a *spatial sequence* of concurrent micro-operations needed to accomplish an RT-operation but these are available in an unordered manner in $\mathscr{M}_A$.

The RT-operations accomplished by the set $\mathscr{M}_A$ of micro-operations are identified using a *rewriting method*. The method also reveals the spatial sequence of data flow needed for an RT-operation in a reverse order (from the destination register back to the source registers). The basic rewriting method consists in rewriting terms one after another in an expression. The micro-operations in which a register occurs in the left hand side (lhs) are found first. Such a micro-operation has the form $r \Leftarrow r_in$, where r is a register and r_in is its input terminal. Next, the right hand side (rhs) expression "r_in" is rewritten by looking for a replacement (micro-operation) in $\mathscr{M}_A$ of the form "$r_in \Leftarrow s$" or "$r_in \Leftarrow s_1 < op > s_2$". So, after rewriting "$r_in$", we have the rhs expression, either of the form "s" or of the form "$s_1 < op > s_2$". In the next step, s (or s_1 *and* s_2 for the latter case) are rewritten *provided they are not registers*. When the expression in hand is

of the form "$s_1 < op > s_2$" (and s_1, s_2 are not registers), then rewriting takes place from left to right in a *breadth-first manner*. Thus, at any point of time, the expression in hand can be of the form "$((s_1 < op_1 > s_2) < op_2 > s3) < op_3 >_\uparrow \ldots$", where the pointer indicates the signal to be rewritten next. The process terminates successfully when all s_i's in the expression in hand are registers. The rewriting method is given in the subsequent subsection.

The control structure of the FSMD M_3 can be obtained from the controller FSM and the RT-operations of each control state can be constructed by the mechanism described above.

5.3.1 Construction of the FSMD M_3: An Example

Let us consider the data-path shown in the figure 5.3. In this figure, $r1, r2, r3$ are registers, $M1, M2$ are multiplexers, FU is a functional unit and $r1_out$, $r2_out$, $r3_out$, $fLin$, $fRin$, $fOut$ are interconnection wires. The control signal names start with CS. The functionalities of the control signals have been described earlier in example 9. The set of all micro-operations $\mathscr{M}$ possible in the data-path of figure 5.3 is as follows:

$$\mathscr{M} = \{r1_out \Leftarrow r1,\ r2_out \Leftarrow r2,\ r3_out \Leftarrow r3,$$
$$fLin \Leftarrow r1_out,\ fLin \Leftarrow r2_out,\ fLin \Leftarrow r3_out,$$
$$fRin \Leftarrow r2_out,\ fRin \Leftarrow r3_out$$
$$fOut \Leftarrow fLin + fRin,\ fOut \Leftarrow fLin - fRin,$$
$$r1 \Leftarrow fOut,\ r3 \Leftarrow fOut\}.$$

Let the order of the control signals in a control signal assertion pattern be

$CS_M1_1 \prec CS_M1_0 \prec CS_M2 \prec CS_FU \prec CS_r1Ld \prec CS_r3Ld$. It may be noted that the micro-operations $fLin \Leftarrow r1_out$ and $fLin \Leftarrow r2_out$ depend on more than one control signal (on both CS_M1_1 and CS_M1_0).

The function f_{mc} from the set of micro-operations $\mathscr{M}$ to the set of all possible control assertion patterns $\mathscr{A}$ is given in table 5.1. This function can be obtained from the output of any HLS tool containing the RTL behaviour of each component used in the data-path.

Let $A = \langle 1,\ 0,\ 1,\ 1,\ 1,\ 0 \rangle$ be the control assertion pattern in a particular state of the controller FSM. The set of micro-operations $\mathscr{M}_A$ for this control assertion pattern A is determined as follows. The superposition of the control assertion pattern of each micro-operation and the pattern A is checked one by one to decide whether to include that particular micro-operation in $\mathscr{M}_A$ or not. This process is tabulated in table 5.2. In the table, U denotes the undefined value. So, $\mathscr{M}_A = \{\ r1_out \Leftarrow$

Micro-operations	Corresponding control assertion pattern $\langle CS_M1_1, CS_M1_0,\ CS_M2,\ CS_FU,\ CS_r1Ld,\ CS_r3Ld\rangle$
$r1_out \Leftarrow r1$	$\langle X,\ X,\ X,\ X,\ X,\ X\rangle$
$r2_out \Leftarrow r2$	$\langle X,\ X,\ X,\ X,\ X,\ X\rangle$
$r3_out \Leftarrow r3$	$\langle X,\ X,\ X,\ X,\ X,\ X\rangle$
$fLin \Leftarrow r1_out$	$\langle 0,\ 0,\ X,\ X,\ X,\ X\rangle$
$fLin \Leftarrow r2_out$	$\langle 0,\ 1,\ X,\ X,\ X,\ X\rangle$
$fLin \Leftarrow r3_out$	$\langle 1,\ X,\ X,\ X,\ X,\ X\rangle$
$fRin \Leftarrow r2_out$	$\langle X,\ X,\ 1,\ X,\ X,\ X\rangle$
$fRin \Leftarrow r3_out$	$\langle X,\ X,\ 0,\ X,\ X,\ X\rangle$
$fOut \Leftarrow fLin + fRin$	$\langle X,\ X,\ X,\ 0,\ X,\ X\rangle$
$fOut \Leftarrow fLin - fRin$	$\langle X,\ X,\ X,\ 1,\ X,\ X\rangle$
$r1 \Leftarrow fOut$	$\langle X,\ X,\ X,\ X,\ 1,\ X\rangle$
$r3 \Leftarrow fOut$	$\langle X,\ X,\ X,\ X,\ X,\ 1\rangle$

Table 5.1: The function f_{mc} from the set $\mathscr{M}$ to the set $\mathscr{A}$

Micro-operation (μ)	Control assertion pattern of μ ($f_{mc}(\mu)$)	$f_{mc}(\mu)\ \theta\ A$	in $\mathscr{M}_A$? (yes/no)
r1_out ⇐ r1	$\langle X,\ X,\ X,\ X,\ X,\ X\rangle$	$\langle X,\ X,\ X,\ X,\ X,\ X\rangle$	**yes**
r2_out ⇐ r2	$\langle X,\ X,\ X,\ X,\ X,\ X\rangle$	$\langle X,\ X,\ X,\ X,\ X,\ X\rangle$	**yes**
r3_out ⇐ r3	$\langle X,\ X,\ X,\ X,\ X,\ X\rangle$	$\langle X,\ X,\ X,\ X,\ X,\ X\rangle$	**yes**
$fLin \Leftarrow r1_out$	$\langle 0,\ 0,\ X,\ X,\ X,\ X\rangle$	$\langle U,\ 0,\ X,\ X,\ X,\ X\rangle$	no
$fLin \Leftarrow r2_out$	$\langle 0,\ 1,\ X,\ X,\ X,\ X\rangle$	$\langle U,\ U,\ X,\ X,\ X,\ X\rangle$	no
fLin ⇐ r3_out	$\langle 1,\ X,\ X,\ X,\ X,\ X\rangle$	$\langle 1,\ X,\ X,\ X,\ X,\ X\rangle$	**yes**
fRin ⇐ r2_out	$\langle X,\ X,\ 1,\ X,\ X,\ X\rangle$	$\langle X,\ X,\ 1,\ X,\ X,\ X\rangle$	**yes**
$fRin \Leftarrow r3_out$	$\langle X,\ X,\ 0,\ X,\ X,\ X\rangle$	$\langle X,\ X,\ U,\ X,\ X,\ X\rangle$	no
$fOut \Leftarrow fLin + fRin$	$\langle X,\ X,\ X,\ 0,\ X,\ X\rangle$	$\langle X,\ X,\ X,\ U,\ X,\ X\rangle$	no
fOut ⇐ fLin − fRin	$\langle X,\ X,\ X,\ 1,\ X,\ X\rangle$	$\langle X,\ X,\ X,\ 1,\ X,\ X\rangle$	**yes**
r1 ⇐ fOut	$\langle X,\ X,\ X,\ X,\ 1,\ X\rangle$	$\langle X,\ X,\ X,\ X,\ 1,\ X\rangle$	**yes**
$r3 \Leftarrow fOut$	$\langle X,\ X,\ X,\ X,\ X,\ 1\rangle$	$\langle X,\ X,\ X,\ X,\ X,\ U\rangle$	no

Table 5.2: Construction of the set $\mathscr{M}_A$ from the function f_{mc} for the control assertion pattern $A = \langle 1,\ 0,\ 1,\ 1,\ 1,\ 0\rangle$

$r1,\ r2_out \Leftarrow r2,\ r3_out \Leftarrow r3,\ fLin \Leftarrow r3_out,\ fRin \Leftarrow r2_out,\ fOut \Leftarrow fLin - fRin,\ r1 \Leftarrow fOut\}$.

The micro-operation in which a register occurs in the left hand side is $r1 \Leftarrow fOut$. The sequence of rewriting steps for this micro-operation is as follows:

$r1 \Leftarrow fOut$

$\Leftarrow fLin - fRin$ [by the micro-opn. $fOut \Leftarrow fLin - fRin$] ($step$ 1)

$\Leftarrow r3_out - fRin$ [by the micro-opn. $fLin \Leftarrow r3_out$] $(step\ 2)$

$\Leftarrow r3_out - r2_out$ [by the micro-opn. $fRin \Leftarrow r2_out$] $(step\ 3)$

$\Leftarrow r3 - r2_out$ [by the micro-opn. $r3_out \Leftarrow r3$] $(step\ 4)$

$\Leftarrow r3 - r2$ [by the micro-opn. $r2_out \Leftarrow r2$] $(step\ 5)$

So, the RT-operation $r1 \Leftarrow r3 - r2$ is executed by the given control assertion pattern A of a state of the FSM and the forward spatial sequence of the micro-operations for this RT-operation is the reverse order in which they are used in the above rewriting steps; more specifically, therefore, the forward sequence is $r2_out \Leftarrow r2$, $r3_out \Leftarrow r3$, $fRin \Leftarrow r2_out$, $fLin \Leftarrow r3_out$, $fOut \Leftarrow fLin - fRin$, $r1 \Leftarrow fOut$.

The RT-operations for all other states of the FSM can be found out in a similar manner.

5.3.2 A Rewriting Method

Algorithm 2, given below, depicts the method of finding the set of RT-operations from a set $\mathscr{M}_A$ of micro-operations. This algorithm uses a function namely $findRewriteSeq$ to find an RT-operation over the data-path using $\mathscr{M}_A$ starting from a micro-operation having a register as its lhs term. This function is given as algorithm 3.

Algorithm 2 Algorithm to find a set of RT-operations accomplished by a set of micro-operations

---

Input: The set of micro-operations $\mathscr{M}_A$ for a given control assertion pattern A.
Output: A set of RT-operations RT_A accomplished by $\mathscr{M}_A$.
Method:
Let $\mathscr{M}'_A$ be $\{\mu | \mu \in \mathscr{M}_A)$ and μ has a register as its lhs term $\}$;

for each μ in $\mathscr{M}'_A$
{
  $replaced = \phi$;
  $Seq[0] \Leftarrow \mu$;
  $\mu \leftarrow findRewriteSeq\ (\mu,\ \mathscr{M}_A,\ replaced,\ Seq,\ 1)$;
  /* "μ" - initially a micro-operation which is finally transformed to an RT-operation by the function
  "*replaced*" - used by the function to detect if a data flow loop is set up by the control assertion.
  "*Seq*" contains the final sequence of micro-operations used in rewriting - depicts the data flow
  in reverse, which obtains the RT-operation. */
  $RT_A = RT_A \cup \{\mu\}$;
}

Algorithm 3 The findRewriteSeq function

Function: $findRewriteSeq(\mu,\ \mathscr{M}_A,\ replaced,\ Seq,\ i)$

/* Starting with the micro-operation μ, this function finds an RT-operation from the set of micro-operations $\mathscr{M}_A$. At the end of successful execution of the function, μ stores the computed RT-operation. This function also stores the rewriting sequence in the array *Seq* in an ordered manner. The variable *replaced* is used to store the terms which have been replaced during the rewrite process. */

```
{
  Let s be the leftmost non-register signal in the rhs expression of μ.
  if (No non-register signal in the rhs expression of μ)
  {
    Report (“the RT operation found is μ”); /* terminates successfully */
    return μ;
  }
  else if (s ∈ replaced)
  {
    Report (“loop set up in the data-path by the control assertion”);
    return empty RT-operation;
  }
  else
  {
    Let M_s ⊂ M_A be the set of micro-operations s.t. each member of M_s has s as its lhs signal.
    if (M_s == ϕ) /* No micro-operation found in M_A which has s as its lhs signal */
    {
      Report(”Inadequate set of micro-operations”);
      return empty RT-operation;
    }
    else if (|M_s| > 1) /* M_s contains more than one micro-operation */
    {
      Report (”data conflict”); /* more than one driver activated for a signal */
      return empty RT-operation;
    }
    else /* M_s contains a single micro-operation. */
    {
      Let M_s = {m};
      Seq[i] = m;
      replace the leftmost occurrence of s in the rhs expression of μ with the rhs expression of m;
      replaced = replaced ∪ {s};
      return findRewriteSeq(μ, M_A, replaced, Seq, i+1);
    }
  }
} /* End of the function */
```

5.3.3 Correctness and Complexity of Algorithm 2

Correctness of the algorithm

The correctness of the rewriting algorithm depends directly on the correctness the function *findRewriteSeq* which is given by the following theorems.

Theorem 4 *(Termination) The function $findRewriteSeq$ always terminates.*

Proof 4 *There are only a finite number of signals in the data-path. The function may terminate in one of the four ways namely, (i) it returns after finding an RT-operation successfully, (ii) it detects "loop set up in the data-path by the control assertion" and returns, (iii) it detect "Inadequate set of micro-operations" and returns or (iv) it detects "data-conflict" and returns. If a recursive invocation does not detect one of the error situations (ii), (iii) or (iv) above, then it must replace a term in the rhs expression of μ and enhance the set "replaced". Hence, if the function invokes itself more number of times than the number of signals in the data-path, then the set "replaced" will finally contain all the signals of the data-path and there will be no more terms in the rhs expression of μ which is replaceable; the next invocation of the function should, therefore, terminate on situation (i) or (ii).*

Definition 15 Forward rewriting by a micro-operation*:*

An expression e is said to be obtained from an expression e^- by forward rewriting by a micro-operation $s \Leftarrow e_r$, if e can be obtained by replacing one or more occurrences of e_r in e^- by s.

It may be noted that in the algorithm, the rewriting of an expression e_1, at hand, by a micro-operation μ of the form $s \Leftarrow e_2$ is carried out by replacing the occurrence of the lhs signal s of μ in the expression e_1 by the rhs expression e_2 of μ. In contrast, the forward rewriting does the opposite, in keeping with the direction of data flow represented by the micro-operation (hence the name).

Lemma 2 *(*Realizability of an RT operation*): An RT operation $r \Leftarrow e$ is realizable over the data path if there exists a sequence σ of micro-operations (over the data path) such that the expression "r" is obtained from the expression e by* forward rewriting *of e by the members of, and according to, the sequence σ.*

Proof: *Every micro-operation over a data path is realizable over it. Thus, if an expression e is obtainable from an expression e^- by forward rewriting of e^- by some micro-operation, then an RT*

of the form $r \Leftarrow e^-$ is realizable over the data path, if $r \Leftarrow e$ is realizable over the same. The proof follows by repeated applications of the above argument over the sequence σ.

Theorem 5 *(Soundness): If the function* findRewriteSeq *terminates successfully, then the algorithm finds an RT operation which is realizable over the data path by the control assertion pattern A.*

Proof 5 *Let the function terminate successfully and the algorithm obtain "Seq" as $\langle \mu_0, \mu_1, \cdots, \mu_k \rangle$ and an RT operation $r \Leftarrow e$, p say. Let us consider the reverse of "Seq" = $\langle \mu_k, \cdots, \mu_1, \mu_0 \rangle = \sigma$, say. Since the algorithm changes only the right hand side (rhs) of μ_0, the last member μ_0 in σ must have r as the lhs register. Let μ_0 be $r \Leftarrow e_0$. Let the (rhs) expression obtained after application of μ_i in "Seq" be e_i. Clearly, $e_k = e$, the rhs of p, the RT operation returned by the algorithm. In the following, we prove that the RT operation $r \Leftarrow e_i$ is realizable by the sequence $\langle \mu_i, \cdots, \mu_1, \mu_0 \rangle, 0 \leq i \leq k$, by induction on i. Hence, in particular, $r \Leftarrow e_k$ (= p) is realizable by σ.*
(Basis $i = 0$)*: $r \Leftarrow e_0$ $(= \mu_0)$ is realizable since all micro-operations are realizable.*
(Induction step)*: Let $r \Leftarrow e_i$ be realizable by $\langle \mu_i, \cdots, \mu_1, \mu_0 \rangle$. Let e_i be of the form $e' s e''$, where s is the leftmost non-register signal in e_i, that is, e' contains only register signals. So, from the step(s) of the algorithm, μ_{i+1} must be of the form $s \Leftarrow e_r$ and the algorithm obtains e_{i+1} as $e' e_r e''$. From induction hypothesis it follows that the RT operation $r \Leftarrow e_{i+1} (= r \Leftarrow e' e_r e'')$ is realizable by the sequence $\langle \mu_{i+1}, \mu_i, \cdots, \mu_1, \mu_0 \rangle$.*

Theorem 6 *(Completeness) If there is an RT-operation p which is realizable using the micro-operation in $\mathscr{M}_A$, then the algorithm returns the sequence of micro-operations corresponding to the in-order traversal of the parse tree of p.*

Proof 6 *Let an RT operation p be realizable using the micro-operations in $\mathcal{M}_a$; let p be of the form: $r \Leftarrow e$. From lemma 2, there exists a sequence of micro-operations such that the expression "r" is obtained by forward-rewriting of e by applying the sequence. Without loss of generality, let this sequence correspond to the traversal of the parse tree of p in the order $\langle (realize)\ rightsubtree,\ left\ subtree,\ root \rangle$ which is a variation of the postorder traversal (with an interchange of order between the subtrees). We refer to this traversal order as postorder (ignoring the variation). Let the sequence σ be $\langle \mu_0, \mu_1, \cdots, \mu_k \rangle$. We now prove that the algorithm realizes the in-order traversal of the parse tree, that is, $\langle root,\ left\ subtree,\ right\ subtree \rangle$ (because of the "replace-leftmost-signal-first" approach) and accordingly returns the sequence $\mu = \langle \mu_k, \mu_{k-1}, \cdots, \mu_0 \rangle$ which is the reverse of*

σ. *We accomplish these two steps by induction on the depth i of the parse tree of the RT operation p.*

(Basis $i = 1$): The RT operation p is just a micro-operation, ν say. The parse tree comprises the root and a leaf and the link corresponds to ν. In this case, $\mu = \sigma = \langle \nu \rangle$.

(Induction step): Suppose that the algorithm can find the sequence of micro-operations corresponding to the in-order traversal of the parse tree of any realizable RT-operation of depth i. In particular, therefore, this sequence minus its first element realizes the in-order of the parse tree of the rhs expression of the RT operation.

Now, let the depth of the parse tree of the RT operation p be $i+1$. We have the following two cases:

case 1*: p is of the form $r \Leftarrow e$. Let the sequence σ corresponding to the postorder traversal of the parse tree of p be $\langle \mu_0, \mu_1, \cdots, \mu_k \rangle$. Obviously, the sequence $\sigma^- = \langle \mu_0, \cdots, \mu_{k-1} \rangle$ realizes (the postorder traversal of the subtree of) e. The parse tree of e has depth i and hence, by induction hypothesis,the algorithm finds the reverse of σ^-, that is, the sequence $\mu^+ = \langle \mu_{k-1}, \cdots, \mu_0 \rangle$. The last member $\mu_k \in \sigma$ pertains to the realization of the root (according to postorder); also, from the definition of realizability of RT operation it follows that μ_k must be of the form $r \Leftarrow e$ so that its application as the last one in the sequence σ renders e to r. Thus, it is found first by the algorithm before finding μ^+. So the algorithm realizes the the sequence $\mu = \langle \mu_k, \mu_{k-1}, \cdots, \mu_0 \rangle$ which corresponds to the in order traversal of the parse tree of p and is the reverse of σ.*

case 2*: p is of the form $r \Leftarrow e_1 \langle op \rangle e_2$. The sequence $\sigma = \langle \mu_0, \mu_1, \cdots, \mu_k \rangle$ corresponding to the postorder traversal of the parse tree of p can be split into three subsequences $\sigma_2 = \langle \mu_0, \mu_1, \cdots, \mu_{i-1} \rangle$, $\sigma_1 = \langle \mu_i, \mu_{i+1}, \cdots, \mu_{k-1} \rangle$ and $\sigma_3 = \langle \mu_k \rangle$, where σ_2 corresponds to the postorder traversal of the right subtree of p (i.e., the parse tree of e_2), σ_1 corresponds to the left subtree of p (i.e., the parse tree of e_1) and σ_3 corresponds to the root. Also, from realizability of p it follows that μ_k is of the form $r \Leftarrow e_1 \langle op \rangle e_2$. The algorithm identifies this as μ_0. The parse trees of e_1 and e_2 are of depths $\leq i$. So, from induction hypothesis, the algorithm realizes the in order traversal of e_1 followed by the in order traversal of e_2 obtaining the sequence ν_1 as the reverse of σ_1 followed by $\nu_2 = reverse(\sigma_2)$, that is, the sequence $\langle \mu_{k-1}, \cdots, \mu_{i+1}, \mu_i, \mu_{i-1}, \cdots, \mu_1, \mu_0 \rangle$. So the algorithm returns the sequence $\langle \mu_k, \mu_{k-1}, \cdots, \mu_{i+1}, \mu_i, \mu_{i-1}, \cdots, \mu_1, \mu_0 \rangle$ which corresponds to the in order traversal of the parse tree of p and is the reverse of σ.*

Complexity of the Rewriting Method

Let the number of FUs, registers and the interconnect components (like, mux, demux, switch etc.,) altogether be c. Let the number wires and the registers in the data-path be w and r, respectively. The number of micro-operations possible in the data-path is $O(wc)$. So, the number of micro-operations in $\mathscr{M}_A$ is $O(wc)$. Let us consider the function $findRewriteSeq$. In each invocation of the function, one term (wire) is rewritten and no term is rewritten more than once. Hence, number of invocations of the function is $O(w)$. The complexity to the function $termToRewrite$ is $O(w)$. The complexity to determine whether $s \in replaced$ is $O(w)$. The complexity of the function findMicroOpn is $O(wc)$. So, the complexity of the function $findRewriteSeq$ is $O(w(w+w+wc)) = O(w^2c)$. The number of micro-operations in $\mathscr{M}_A'$ is $O(r)$. Hence, the complexity of the rewriting method is $O(w^2cr)$.

5.4 Verification During Construction of FSMD M_3

Several inconsistencies in the data-path interconnections and in the control signal assertion patterns can be detected during construction of the FSMD M_3. They are discussed as follows.

- *Inadequate set of micro-operations performed by a control assertion pattern:* It occurs when no micro-operation is selected by the rewriting rule in a certain step of the rewriting process before the terminating condition (that is, all the terms in the rhs expression are registers) is reached. This situation arises due to either of following two reasons: (i) interconnection between two data-path components is not actually set by the control pattern but is required to complete an RT-operation and (ii) the control signals are asserted in a wrong manner which leads to a situation where the required data transfer is not possible in the data-path.

- *Data conflict:* It occurs when more than one replacement are found for a non-register term in any step of the rewriting process. It means more than one data from different data-path components try to pass through a single data-path component which obviously causes a data conflict. It arises due to wrong control assertion pattern.

- One non-register data-path component can be assigned by only one value in a particular control step. If a non-register term is rewritten twice during the rewriting process, then it implies an improper control assertion pattern or a loop in the data-path structure without having any register.

The correctness of the functional unit (FU) allocation and binding tasks are already defined by in section 4.3 by the following two properties.

1. Enough FUs are allocated.
2. The operations in each control step are properly mapped to the FUs

The verification of these two properties is also performed during the rewriting process. If one or both of these two properties do not hold, then the rewriting method will report some erroneous situations. Let the 1st property does not hold. As a result of this, the following two situations might occur

1. Some operations are not bounded to any FU,
2. Two or more operations are mapped to the same FU.

The rewriting method will fail to find any replacement for some non-register term in course of rewriting process and reports "Inadequate set of micro-operations" in the situation 1. It will find more than replacements for some non-register term in the situation 2; hence reports "data conflict". If the 2nd property does not hold, then two or more operations scheduled in the same time step are mapped to the same FU. As a result of it, the situation 2 will occurs. In this case also, the rewriting process reports "data conflict".

5.4.1 Redundancy Optimization in the Data-path and in the Controller

It might happen that some micro-operations in the set $\mathscr{M}_A$ are never selected by the rewriting rule for a control assertion pattern A. It means that these micro-operations are not required for the RT-operations performed by the assertion pattern A and one may find more proper assertion pattern A' so that $\mathscr{M}_{A'}$ does not contain these extra micro-operations. One may note that some micro-operations happen always in the data-path because they are not controlled by any control signals. For example, the micro-operations $r1_out \Leftarrow r1$, $r2_out \Leftarrow r2$, $r3_out \Leftarrow r3$ in figure 5.3 are not controlled by any control signal and they happen in every control step of controller FSM. This type of micro-operations are ignored during redundancy optimization.

If some micro-operations of the set $\mathscr{M}$ are not present in any $\mathscr{M}_A$ for all assertion patterns A that occur in the different states of the controller FSM, then it means that these micro-operations are not at all required to execute the specified input behaviour in the data-path. So, one may optimize

the DP further to restrict these micro-operations from occurring in the DP without affecting other micro-operations. Again, the micro-operations which do not involve any control signal are ignored during redundancy optimization.

5.5 Verification by Equivalence Checking

In the data-path and controller generation phase, the behaviour represented by the FSMD M_2 is mapped to hardware. The number of states and the control structure of the behaviour are not modified in this phase. Hence, there is a one-to-one correspondence between the states of FSMDs M_2 and M_3. Let the mapping between the states of M_2 and those of M_3 be represented by a function $f_{23} : Q_2 \leftrightarrow Q_3$. The state q_{3i} ($\in Q_3$) of the FSMD M_3 is said to be the corresponding state of q_{2i} ($\in Q_2$) if $f_{23}(q_{2i}) = q_{3i}$.

A set of RT-operations are formed for each state transition of the FSMD M_3 from the corresponding control assertion pattern. Now, the question is whether all the RT-operations corresponding to each state transition of the FSMD M_2 is captured by the controller or not. In other words, it is required to verify that all the RT-operations in each state transition in FSMD M_2 are also present in the corresponding state transition in FSMD M_3 and no extra RT-operation occurs in the transition of the FSMD M_3. A transition $q_{3k} \xrightarrow[c']{} q_{3l}$ of the FSMD M_3 is said to be the corresponding transition of a transition $q_{2k} \xrightarrow[c]{} q_{2l}$ of the FSMD M_2 if $f_{23}(q_{2k}) = q_{3k}$, $f_{23}(q_{2l}) = q_{3l}$ and the condition c is equivalent to the condition c'. In the following, a *state based equivalence checking algorithm* is given.

The successful completion of the algorithm ensures that *any RT-operation occurs in any state transition of M_3 iff it also occurs in the corresponding transition in M_2*. It assures that the control signal generation in each state is correct. The number of iterations of the algorithm mainly depends on the number of transitions in the FSMD M_2. If the number of transitions in M_2 is e and the maximum possible RT-operations in any transition is k, then the complexity of the algorithm is $O(ek)$.

5.6 Conclusions

The verification of data-path interconnection and the controller behaviour is discussed in this chapter. The verification task is performed in two steps. In the first step, an FSMD M_3 is constructed

Algorithm 4 State based equivalence checking algorithm

```
Input:    FSMD M2, M3 and the function f23.
Output:   'yes/no' answer for "M2 is equivalent to M3".
for each state q2i of M2
{
  Let q3i be f23(q2i); // q3i is the corresponding state of q2i
  for each state transition t from q2i
  {
    find a transition t' from q3i which has an equivalent condition to that of t;
    for each RT-operation opn in t
    {
      if opn does not occur in t'
      {
        report "opn is not present in transition t'; hence not equivalent";
        exit;
      }
    }
    if there is some RT-operation which occurs in t' but not in t
    {
      report "extra RT-operation occurs in t'; hence not equivalent"
      exit;
    }
  }
}
```

from the data-path information and the controller FSM. In the second step, a state based equivalence checking methodology is used to verify the correctness of the controller behaviour. A *rewriting method* is proposed which is used during the construction of the FSMD M_3; the method finds the RT-operations performed by a given control assertion pattern in each state of the controller FSM. The correctness of this method has been proved and the complexity of the method is analyzed. The construction process is described with an example. Several inconsistencies and redundancies, both in the data-path and the controller, are revealed during construction of the FSMD M_3. The state based equivalence checking method ensures that any RT-operation occurs in an state transition of M_3 iff it also occurs in the corresponding transition in the FSMD M_2.

Chapter 6

Development of a High-level Synthesis Tool (SAST)

6.1 Introduction

A high-level synthesis tool, called *structured architecture synthesis tool (SAST)*, has been developed in this work to support hand in hand synthesis and verification. SAST takes the behavioural description and produces a synthesizable register transfer level (RTL) code in Verilog. It is an interconnection aware HLS tool as it produces a structured data-path by avoiding random interconnections among the data-path elements. This tool automatically generates the FSMDs from its input, intermediary results and the output so that a phase-wise verification of HLS can be carried out as proposed in this work. This chapter covers the following aspects:

- Target architecture
- Synthesis flow of SAST
- Generation of FSMDs from the intermediate synthesis results of SAST

6.2 Target Architecture

A *structure architecture* (SA) has been considered for the data-path in the SAST tool. The generated data path is organized as *architectural blocks* (A-blocks). Each A-block has a local functional

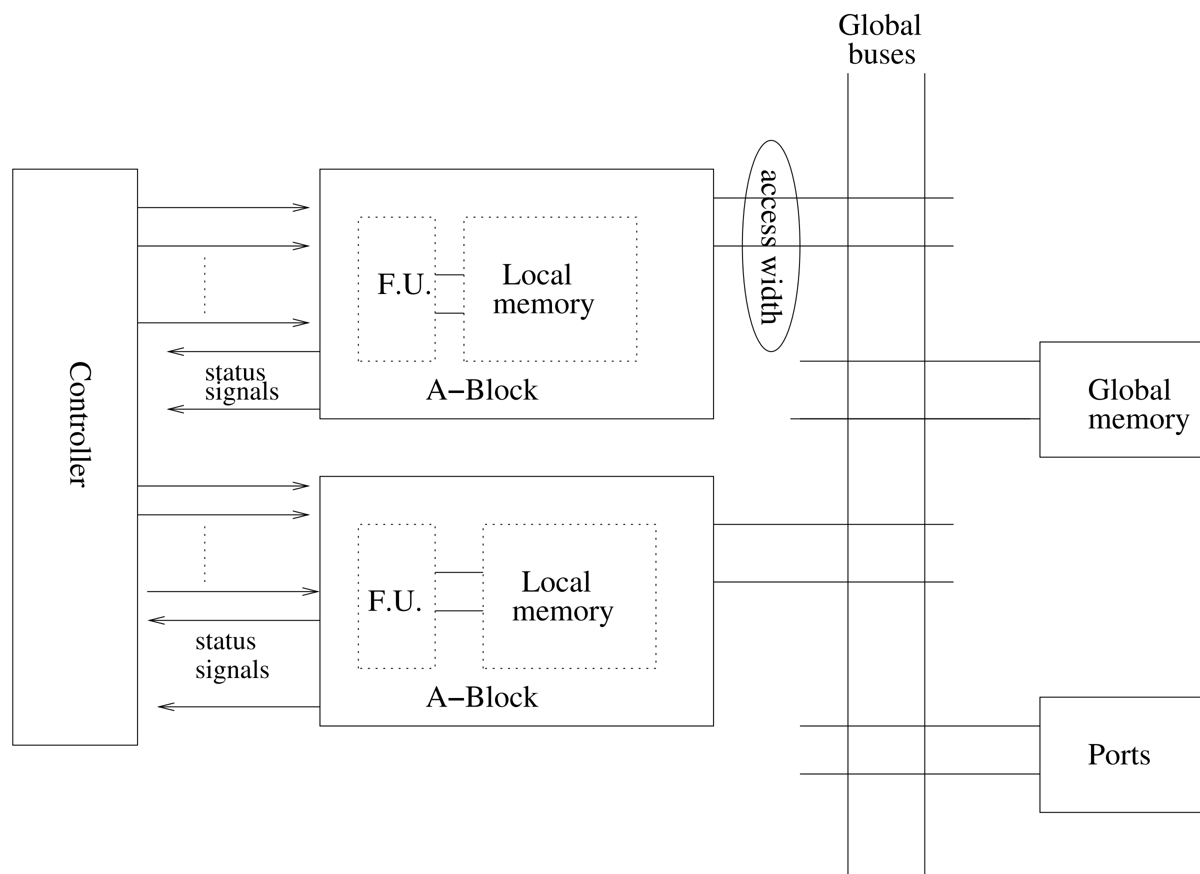


Figure 6.1: Schematic of structured architecture.

unit (FU), local storage and local buses. All the A-blocks in a design are interconnected by a number of global buses. In addition to the local memories in all the A-blocks, SAST also permits the use of global memories as architectural components. These memories are similar to an A-block, except that they do not have any functional unit associated with itself. These memories can be accessed globally by all the A-blocks using global buses. The schematic diagram of the Structured Architecture (SA) is shown in Figure 6.1. All the data path components such as, the local buses, storage units, functional units in the A-blocks and the global buses, are of the same width.

The SA is characterized by a set of architectural constraints such as the number of A-blocks, the number of global memories, the number of global buses interconnecting the A-blocks, the number of access links or access width connecting an A-block to the global buses and the maximum number of concurrent writes per time step to the storage locations in an A-block. The architectural parameters which are internal to an A-block (e.g. the number of access links and the number of write ports to the internal memory, etc.) are the same for each A-block. This structured data paths

avoid random interconnects between data path elements. Each A-block has a regular implementation. As a result the reusability of data-paths are expected to have simple physical design. Figure 6.2 illustrates an A-block.

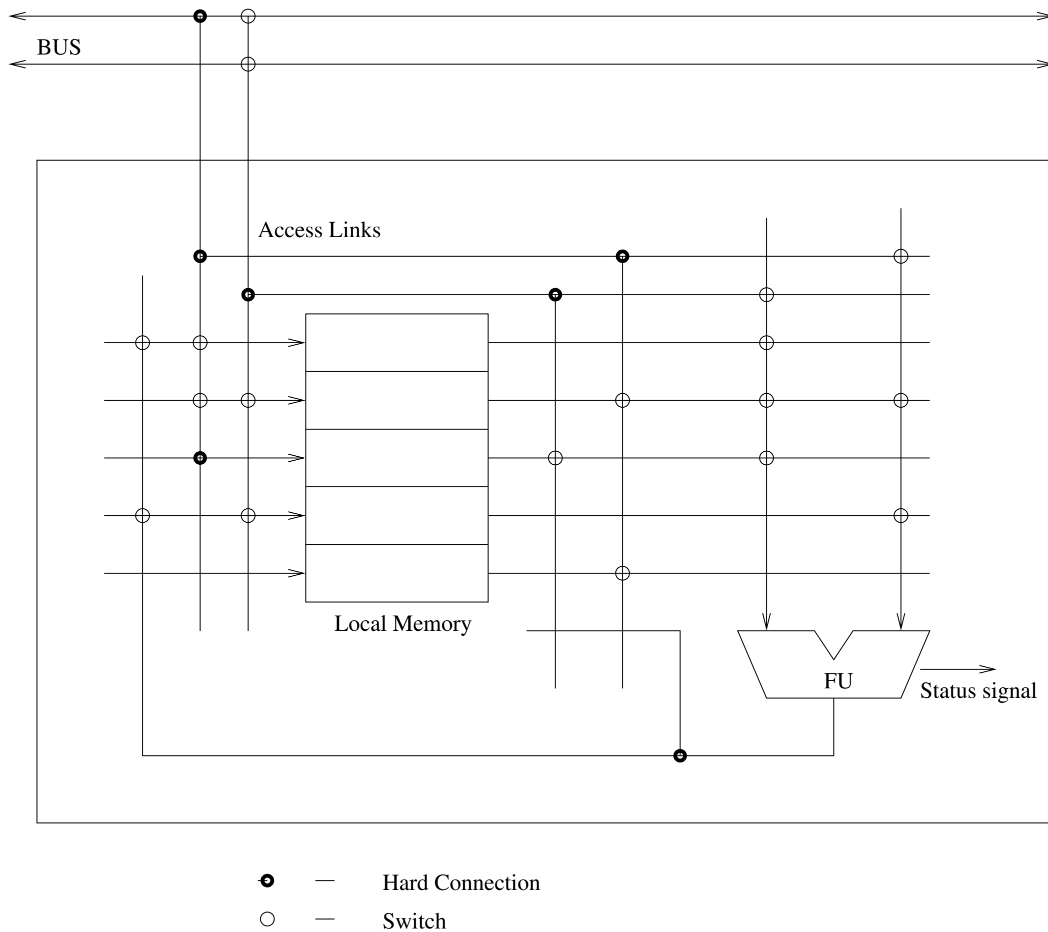


Figure 6.2: An Architecture Block

Input/Output ports of the system are connected to the global buses so that all the A-blocks can access any of the ports. Each A-block has local memory as register bank, which are connected to the global buses through internal buses (access links). Each A-block has one functional unit (FU), which takes input from either the local memory or the internal buses. The output from the functional unit connects back to either the register bank or the internal buses. There are switches to enable/disable the connection between any two components in the A-block. Switches, which connect the internal buses and the output of FU to the input ports of the registers, are called *in-switches*. Switches, which connect the output of registers (to the inputs of the FUs) and output of the FU to internal buses, are called *out-switches*. Global buses are connected to input ports of FU

through internal buses and out-switches. The output port of the FU is also connected to the global buses through internal buses and out-switches.

6.3 SAST Synthesis Steps

The synthesis tool SAST takes a behavioural description and the architectural parameters of the structured data-path as input and produces a synthesizable RTL code in Verilog. SAST consists of the following sequence of phases:

1. *Control and data flow graph (CDFG) generation:* The input behaviour is translated into a CDFG in this phase.

2. *Preprocessing:* This step converts the CDFG into an intermediate representation which is required for the scheduling process. The main task of the preprocessor module is to find the dependency information within each basic block and the incoming and outgoing variables of each basic block.

3. *Scheduling:* The *scheduler* schedules the operations in minimum number of time steps. The scheduler of SAST also schedules all the transfers over the global buses. It also gives the composition of the functional units within each A-block and binds the operators of the input behaviour to the functional units.

4. *Register allocation and binding:* The minimum number of registers required to store the variables is found and their binding is done in this phase.

5. Data-path and control-path generation: The interconnection of the data-path is found out based on the scheduling, allocation and binding informations and the controller is constructed in this phase.

6. Verilog code generation: Finally, the data-path and control-path informations are encoded in Verilog in this phase.

The synthesis steps of SAST's are shown in figure 6.3 and they are discussed in detail in the subsequent subsections.

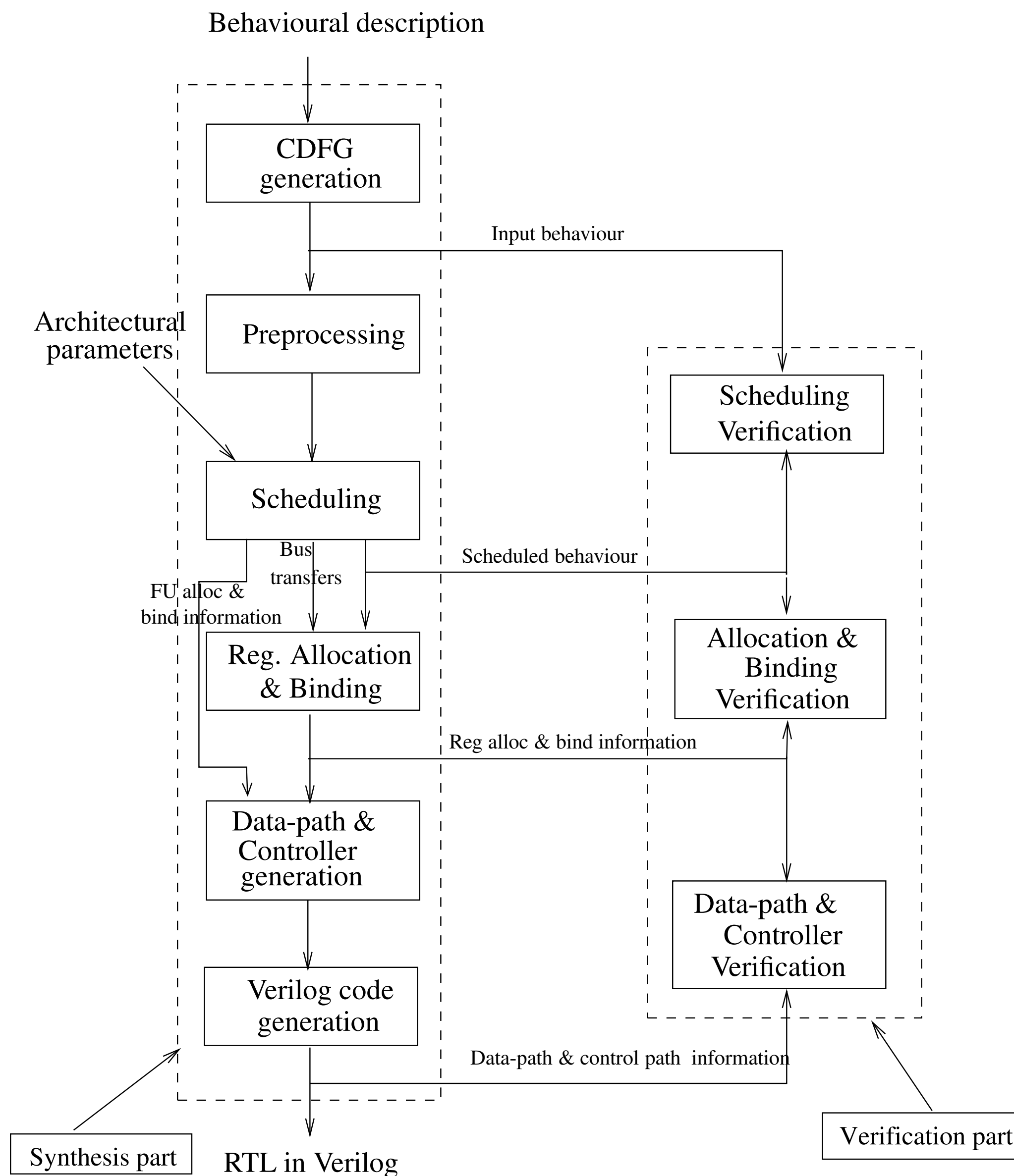


Figure 6.3: SAST synthesis steps

6.3.1 CDFG Generation

Most of the HLS systems require the input in the form of a CDFG to be used for all the phases. Instead, the behavioral design specification at a high level of abstraction is coded in languages like C or some hardware design language such as VHDL, Verilog, etc and provided as input to an HLS system. The methodologies to translate this high-level specification into CDFG are similar to the ones used in the front-end of a typical compiler flow. The CDFG representation used, behavioural input used for SAST and the methodology for the translation scheme are discussed in

this subsection.

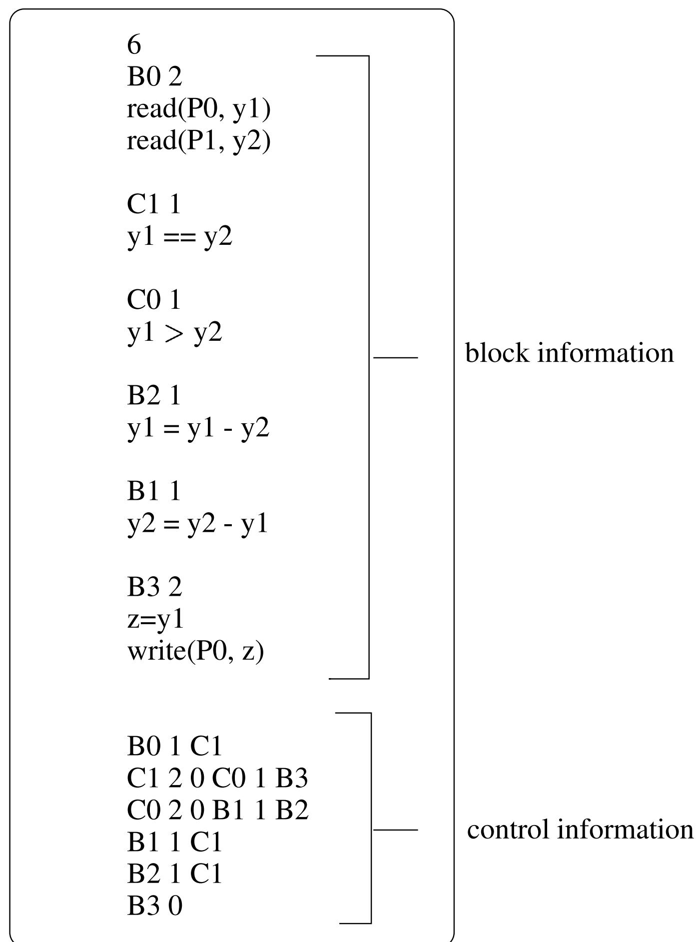


Figure 6.4: CDFG representation of GCD behaviour

CDFG Representation

The CDFG representation used by *SAST* is block based. The CDFG is a directed graph that can be represented as $B = (V, E)$. A node $v \in V$ represents either a basic block or a conditional block. An edge $e \in E$ maintains the control flow among these blocks. Each basic block consists of a set of there-address operations, maintaining the data dependency among the operations.The control blocks consist of conditional statements representing constructs like *IF*, *CASE* or *LOOP*. There is a directed edge from the block b_i to the block b_j if

- there is a conditional jump from the last statement of b_i to the first statement of b_j, or
- b_j immediately follows b_i in the order of the program.

The block b_i is said to be the *predecessor* of b_j, and b_j is said to the *successor* of b_i.

An example of CDFG representation of the GCD behaviour is shown in figure 6.4. The value 6, written at the start of the CDFG, denotes the number of blocks in the CDFG. It is followed by the expression $B0$ 2, where $B0$ indicates a basic block and 2 denotes the number of operations in $B0$. The operations in the basic block $B0$ is given next. We introduce the notions for reading and writing data from ports namely as *read* and *write* operations, respectively, in the CDFG representation. The statement $read(P0, y1)$ in the figure means the input value read from port $P0$ are assigned to the variable $y1$. The order of operations within a basic block maintains the data dependency among the operations. There are six blocks in this CDFG, where $B0$, $B1$, $B2$, $B3$ are the basic blocks and $C0$, $C1$ are the conditional blocks in figure 6.4. The control structure of the CDFG follows next. There is only one successor for every basic block. For example, $B0$ 1 $C1$ in figure 6.4 indicates that $B0$ has only one successor block namely $C1$. A conditional block has two successor blocks. For example, the conditional block $C1$ has two successors as given by $C1$ 2 0 $C0$ 1 $B3$ in figure 6.4. It means that the control goes to the block $C0$ if the conditional statement $y1 == y2$ of $C1$ becomes false; else control goes to $B3$. The successor field need not be explicit - for a basic block, it is always 1, for a conditional block, it is always 2.

Behavioural input for SAST

For *SAST*, a language, similar to C but with facilities to specify interfaces that will finally appear in the synthesized circuit, is used for behavioral specification. Common programming constructs like assignment, conditional, loops, case statements, are available.

Translation Methodology

The formation of CDFG requires parsing of the input behavioural code. The various steps involved for this task are shown in figure 6.5. These phases resemble the front-end of any compiler. First two phases were generated by standard compiler construction tools, Lex and YACC.

The output of the syntactic analysis phase is a tree structure commonly known as *parse tree* or *Abstract Syntax Tree* . This tree represents the syntactic structure of the program according to formal grammar associated with the parser. Explicit rules are required to extract such a parse tree from the YACC code. These redefine rules were written corresponding to the grammar productions to extract the CDFG representation. This methodology of translating behavioural description to its equivalent CDFG is also known as *Syntax-directed translation*.

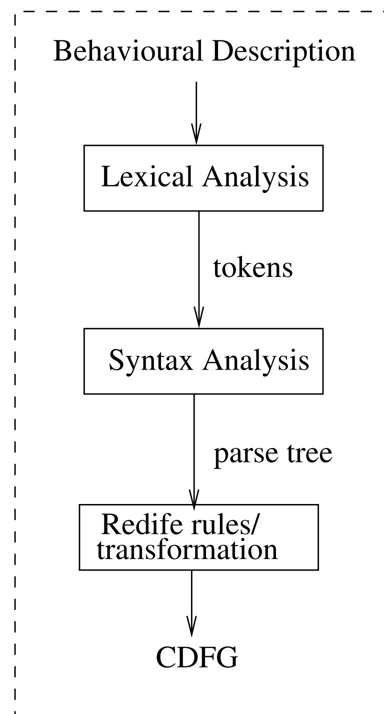


Figure 6.5: Steps involved in CDFG generation.

6.3.2 Preprocessing

The preprocessor converts the CDFG into an intermediate representation (IR) which consists of the precedence constraints or partial order between the operations in each basic block, along with the incoming and outgoing variable sets for each basic block. The preprocessing task is explained in figure 6.6. The main sub-tasks are as follows:

- *Sanitization*: constructs the symbol table of variables in the CDFG, the operation list for each basic block and the successor blocks for each block of the CDFG.
- *construct live sets*: computes input and output variable list for each basic block, from the list of operations in the basic block and the flow of control information,
- *construct dependency*: computes the precedence constraints between the operations in the basic block, from the live sets and operations of the basic block, and
- *generate intermediate form*: puts the basic block information in a manner which is suitable for scheduling basic blocks with the existing scheduling algorithm of SAST, from the live sets and partial order of the basic blocks.

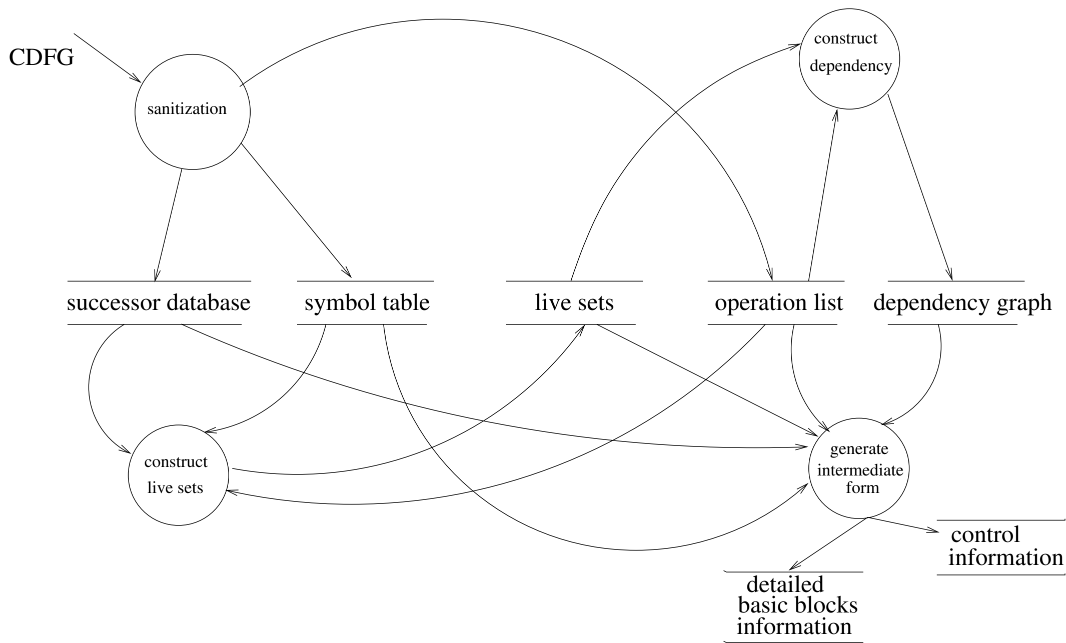


Figure 6.6: Data flow diagram for preprocessor before scheduling

Live variable analysis

It computes two set, one of the incoming variables and the other of the outgoing variables for each basic block in the CDFG, using the control flow among the basic blocks and the list of operations in each basic block, built from the sanitization. Data flow analysis is performed over the CDFG to find out the incoming and the outgoing variables of each basic block.
Four different sets need to be maintained for each basic block *i*,
use_i : set of variables whose values may be used in *i* prior to any definition of the variable in *i*,
def_i : set of variables being defined in the *i* prior to any use of that variable in *i*,
in_i : set of variables live at the entry point of *i*,
out_i : set of variables live at the exit point of *i*.
These sets are used in computing the incoming and the outgoing variable sets for *i*, from the flow of control information and operation list in *i*.

Computation of *use*, *def* sets: For each basic block (bb) *i*, use_i is the set of variables used before defining in the bb *i* and def_i is the set of all left hand side (lhs) variables of the operations in the bb *i*. The following data flow equations compute the sets use_i and def_i from the set of operations

in the basic block i:

$$use_i := \bigcup_{operations \in i} source\ operands\ of\ operation \tag{6.1}$$

$$def_i := \bigcup_{operations \in i} destination\ operand\ of\ operation \tag{6.2}$$

Computation of the sets *in* and *out*: The following data flow equations compute the sets in_i and out_i from the sets use_i, and def_i and the flow of control information:

$$out_i := \bigcup_{j \in succ(i)} in_j \tag{6.3}$$

$$in_i := use_i \cup (out_i - def_i) \tag{6.4}$$

From equation 6.3, we say that a variable v is live at the exit point of a basic block iff it is live coming into one of its successors. Similarly, using equation 6.4, a variable v is live coming into a basic block i if either it is used before redefinition in i or it is live coming out of i and is not redefined in i.

Dependency graph extraction

After construction of the live sets for each basic block i, the partial order or the dependency graph between the operations in each basic block i is to be constructed. A *dependency graph* of a basic block consists of nodes representing functional operators and read/write operators corresponding to the I/O interface (port). Nodes are connected by arcs that represent either the communication of values or the ordering of I/O operations by dependencies. If a node N_1 computes a value that is used by node N_2, then there is an edge from N_1 to N_2. The communication between nodes along the path represents whether the value computed in N_1 is actually used in N_2. However, in addition to the read-before-write and write-before-read dependencies that exist between normal operations, there exist read-before-read dependencies between operations to an I/O port, since the values present at a port is changed by the execution sequence of port operations. Let us consider the behavioural description given in figure 6.7 [6]. The partial order constructed for the set of three address statements of this behavioural description is shown in figure 6.8, with the incoming

variables in= {3, dx, u, x, y}.

```
v0 = dx *  u (0)  v3 =  3 *  y (4)  u = v4 - v5 (8)
v1 =  3 *  x (1)  v4 =  u - v2 (5)  y =  y + v6 (9)
 x =  x + dx (2)  v5 = dx * v3 (6)
v2 = v0 * v1 (3)  v6 =  u * dx (7)
```

Figure 6.7: A sample behavioural description

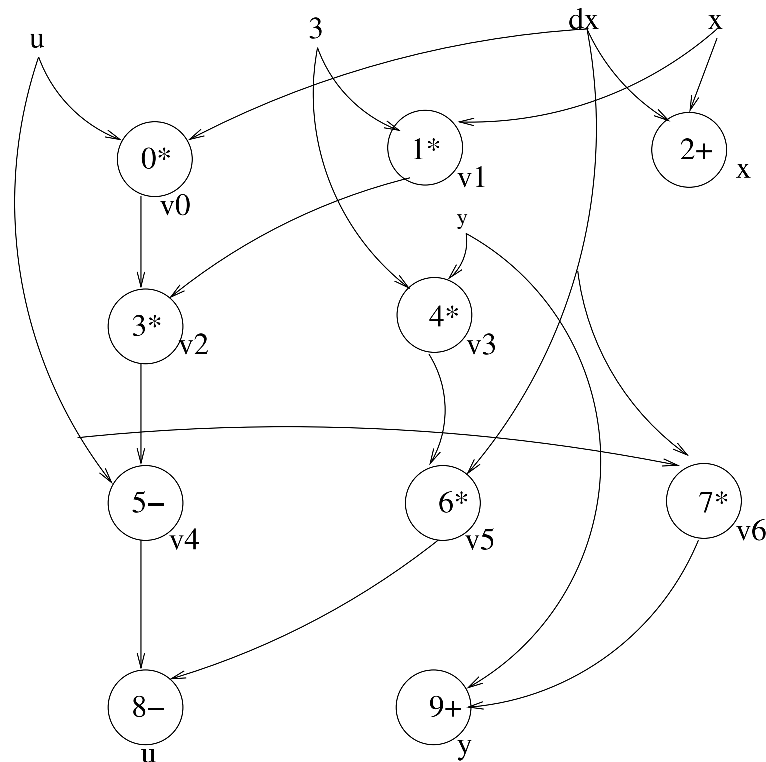


Figure 6.8: Partial order for the behavioural description given in figure 6.7.

Intermediate representation

The live variable *in* and *out* sets for each basic block and the partial order among the operations in each basic block is represented as an intermediate representation which is used for scheduling by SAST.

6.3.3 Scheduling

Scheduler takes the live variable sets, the dependency graph of each basic block and the architectural parameters as inputs and schedules the list of operations satisfying the architectural constraints. The scheduler of SAST is genetic algorithm (GA) based. The GA is designed to support synthesis of structured data paths. The scheduling is guided by user specified architectural parameters such as, the numbers of A-blocks, global buses, access links and concurrent writes in an A-block. The last parameter determines the number of write ports for the local memory in each A-block. There is no restriction on the number of read ports for the local memory.

Scheduling algorithm of SAST delivers the following:

- a schedule of operations over time and over the A-blocks,
- the schedule of all transfers over the global buses satisfying the architectural constraints, and
- the composition of the FU in each A-block,

The last item suggests that the SAST scheduler also takes care of the functional unit (FU) allocation and binding task. It is because of the structured nature of the architecture. Whenever an operation is scheduled in any A-block, the FU of that A-block is allocated to that operation. The composition of the FU of an A-block is the set of all the operations which gets scheduled to the A-block.

Time	Bus transfer	program variable definition	Schedule of operations	
			in ablk A_0	in ablk A_1
1.	$dx(1 \to 0)$		4. $v3 = 3 * y$	0. $v0 = dx * u$
2.	$x(1 \to 0)$		6. $v5 = dx * v3$	1. $v1 = 3 * x$
3.		$2(0) \to x(1)$	2. $x = x + dx$	3. $v2 = v0 * v1$
4.	$u(1 \to 0)$		7. $v6 = u * dx$	5. $v4 = u - v2$
5.	$v5(0 \to 1)$	$9(0) \to y(0),\ 8(1) \to u(1)$	9. $y = y + v6$	8. $u = v4 - v5$

Table 6.1: Schedule of operations of the code for DIFFEQ given in figure 6.7.

Let us consider the schedule in table 6.1 for the operations given in figure 6.7. This schedule has been constructed for a structured architecture with two A-blocks, one global bus and one access link per A-block. The operations are scheduled in 5 time steps as shown in table 6.1. It is assumed that the value generated by an operation in an A-block gets stored in that A-block after completion of the operation. If an operand of an operation is not available in the A-block where the operation is scheduled, then it needs to be brought from the A-block where it is available using global bus. In

the schedule of table 6.1, the transfers of relevant data among the A-blocks is shown; the numbers in round brackets identify the source and destination A-blocks. For example, $v5$ $(0 \rightarrow 1)$ indicates that the variable $v5$ is transfered from the A-block 0 to the A-block 1. The necessity of transferring the variable $v5$ from the A-block 0 to 1 is as follows. The operation $v5 \Leftarrow dx * v3$ is scheduled in the second time step in A-block 0. As, it is assumed that the value generated by an operation in an A-block gets stored in that A-block after the completion of the operation, the value of $v5$ is stored in A-block 0. Similarly, the value of $v4$ is stored in A-block 1. Now, the operation $u \Leftarrow v4 - v5$ is scheduled in the A-block 1 in the time step 5. So, the variable $v5$ needs to be transfered from the A-block 0 to the A-block 1. The scheduler schedules this bus transfer in the time step 5 as the bus is available in that time step. If no bus were available in that time step, then the scheduler would schedule this bus transfer in time step 4 or 3 (searching in descending order) as the value of the variable $v5$ is available from the time step 3.

Sometimes there are variables defined in a basic block and required after all the operations in that basic block are performed. This is common in loop based computation where a variable is updated in the body of the loop and used in the next iteration. We refer to such variables as *program variables*. It is required to know the A-block in which a program variable is available initially. This is taken as input in the current implementation of SAST. In this example, 3, dx, x, y and u are initially located in the A-blocks 0, 1, 1, 0 and 1, respectively. Variable definition column represents the definition of the program variables, that is, a variable whose values have been used in the successor blocks. For example, 2 (0) $\rightarrow$ x (1) indicates that the result of the second operation scheduled in A_0 defines the outgoing variable x. Updated value of x will be stored in A-block 1.

GA Based Scheduling

A brief overview of the GA is as follows. The detailed description follows in the succeeding paragraphs. In view of the complex nature of the problem a structured solution representation has been used, as against a conventional simple bit string. An initial population of solutions is generated at random. New solutions are obtained by inheriting values of the decision variables from parent solutions, picked up from the population. It was noted in the earlier section that for many problems the attributes are not independent and so the resulting solution representation could correspond to an infeasible solution. This is true in the case of scheduling problem and so a completion algorithm has to be used to obtain a feasible solution from the available solution representation, obtained through crossover or by setting the attributes at random while generating

the initial population of solutions. A scheduling heuristic has also been used with the completion algorithm and this has been found to improve the performance of the GA. A population control mechanism had to be employed to sustain diversity in the population, while at the same time retaining solutions with good overall and also partial fitness. The GA is run up to a fixed number of iterations and this serves as the stopping criterion. The last improvement in solution cost (i.e. when the best solution is obtained) usually occurs well before all the iterations are completed.

In the rest of this section we explain the solution representation, the cost function, the parent selection scheme, the crossover scheme, the completion algorithm, the replacement scheme and the heuristic to enhance the performance of the GA.

Solution representation: Each solution contains several decisions which are required for the proper implementation of the design. For each operation, the time when it is scheduled and the A-block where it is to be scheduled are stored. For each input operand of an operation, the A-block from where this value is to be obtained and the transfer time are given. If the operand is present in the same A-block, then the time of transfer is redundant. For each program variable, the time of assignment and the A-block from where the value of it is available are indicated. In order to represent the composition of an FU, it is necessary to indicate which operations an FU can implement.

Thus, there are three types of information to be represented, namely i) Information directly related to the scheduling of operations, ii) information indicating the scheduling of variable transfers and iii) information regarding the composition of FUs. A structured representation is used for storing the above information. This is unlike the simple binary bit based coding commonly used with GAs. Usual crossover techniques with a bit based representation for such a problem would often produce meaningless or infeasible solution representations. As a result, computing time and resources would be wasted in producing such infeasible offsprings. The structured representation used here is suitable for performing an algorithmic crossover (described shortly in the paragraph on solution completion), which leads to a feasible solution representation.

Cost function: The scheduling algorithm tries to find a schedule of operations and transfers within a specified number of time steps. The solution cost is formulated to indicate the cost of the

hardware and the extra time steps used in the schedule. It is of the form

$$C = (penalty)(extra\ \ time\ \ steps) + (cost\ \ of\ \ FUs).$$

The penalty is chosen to accord priority to finding a solution within the specified number of time steps. It is set to be a constant which is an order of magnitude higher than the maximum possible cost of the FUs. In addition, the cost of the FUs is also separately accessible for performing population control, explained later.

Parent selection: The parents are selected on the basis of their costs using the roulette wheel technique [76]. This being a minimization problem, the selection probability of a parent is computed taking into account the maximum cost of solutions in the population as follows: $p_{s_i} = \frac{C_{max} + \delta - C_i}{N_{sols}(C_{max} + \delta) - \sum_i C_i}$, where p_{s_i} is selection probability for solution i, $\delta \geq 0$, C_i is the cost of the solution, C_{max} is the maximum solution cost in the current population and N_{sols} is the number of solutions in the population. Solutions with higher cost are selected with lower probability. If $\delta = 0$ then the solutions with cost C_{max} will never be selected. Selection is done with replacement so that a solution may participate more than once in crossovers.

```
procedure crossover()
1.  chose two parents from the population of solutions.
2.  mutate each parent according to the mutation probability.
3.  for each operation to schedule do
4.      inherit the various scheduling information of the operation
             (such as, the A-block where it is to be scheduled,
              the time when the operation is to be initiated,
              for each input operand, the source A-block and the
              transfer time) from the two parents.
5.  for each of the program variables do
6.      inherit the time of assignment and the source A-block from
              the two parents.
7.  for each of the A-blocks
8.      inherit a library module to implement operations to be realized
              in the FU of this A-block from the two parents.
```

Figure 6.9: Generating initial attributes of offspring by crossover.

Crossover: New solutions are generated through crossover (refer to figure 6.9). The example 10 treats the formation of the operation scheduling attributes through crossover. First two parent solutions are selected. These go through a mutation and then the actual crossover takes place to generate a raw offspring. The crossover proceeds with inheritance of attributes of the solution from both the parents. These attributes include schedule times and A-block bindings of operations, transfer times for operation inputs and the defined program variables. The FU configuration of the solution is also formed by inheritance from the parents. Inheritance of the attributes from either of the two parents proceeds in the (inverse) ratio of their solution costs. The solution at this stage is, in general, not feasible. The completion algorithm, explained next, is applied to this raw solution to generate a feasible solution.

Example 10 *We consider the formation of scheduling attributes of operations. Consider operation '$3: v2 = v0 * v1$' in figure 6.7 which is a multiplication. Operation '3' has the variables $v0$ and $v1$ as the left and the right operands, respectively.*

Table 6.2 shows some hypothetical scheduling attributes of the two operations in the parent solutions. It also shows the attributes resulting in the offspring solution as a result of the crossover. Several interesting aspects are to be noted. The offspring inherits the A-block of the first parent. The source A-block for the left and the right operands are inherited from the second and the first parent, respectively. This has resulted in a possible inconsistency because now, both the operands are preferably obtained from A-block '0', which may not be possible. Another inconsistency is evident in the transfer time step of the left operand, which has taken the value 4, while the operation is scheduled in time step 3. These inconsistencies come about because the scheduling attributes are not independent. They are resolved during solution completion. □

Solution completion: It was indicated in example 10 that crossover produces a solution representation that may not correspond to a feasible solution. A procedure for 'solution completion' is applied to the raw solution resulting from attribute inheritance during crossover. Solution completion is also applied while generating new solutions because the randomly generated attributes used to construct the initial solutions may not correspond to feasible solutions either. The procedure is essentially a list scheduling algorithm with some programming intricacies to support the various features. A simplified version is shown in figure 6.10. A simple example of application of solution completion, while scheduling the operation whose scheduling attribute formation is illustrated in

Attribute	Value in parent 1	Value in parent 2	Value in off-spring	Sourcing parent
Scheduled in time step	3	4	3	1
A-blk. where scheduled	1	0	1	1
Source A-blk. of left operand	1	0	0	2
Time step of transfer of left operand	3	4	4	2
Source A-blk. of right operand	0	1	0	1
Time step of transfer of right operand	3	3	3	1 or 2

Table 6.2: Crossover of scheduling attributes of operation ‘3’ of figure 6.7

example 10, is given in example 11. The main data structures is a pair of lists, the ready list and the active list. A pair of these lists is used for scheduling operations and another pair for scheduling assignments. Operations or assignments in both types of lists are ready for scheduling in the current time step. However, it is only attempted to schedule operations or assignments from the corresponding active list. In each iteration the ready lists are processed to transfer some operations or transfers to the corresponding active lists. It is first attempted to schedule operations in the active list on the unit indicated in the solution representation for that operation. If this attempt to schedule the operation fails, then it is attempted to schedule these operations on other available FUs. This is done to utilize the FUs which may otherwise go utilized in the current time step and is done only after it has been attempted to schedule all the operations on the active list on the designated FU. If any operation gets scheduled, then the process of transferring operations to the active list from the ready list and then scheduling them is repeated. The intention of maintaining an active list of operations is to give priority to the operations in this list over the operations in the ready list for scheduling in the current time step. Assignments are normally handled after all the operations in the current time step have been scheduled. To avoid any excessive adverse effect of such a bias, assignments are sometimes attempted before trying the second round of scheduling operations, as indicated above, on other available FUs. When no more scheduling is possible, the data structures are updated to close the current time step and scheduling proceeds from the next time step.

Example 11 *Consider the application of solution completion to the offspring considered in example 10. Assume that operation ‘3’ occurs in the active list while scheduling for time step ‘3’. Let us assume that A-block ‘1’ is available and the operation can be placed there as indicated by the corresponding attribute in the offspring. The algorithm would find that it is not feasible to transfer*

the left operand into the A-block in time step 4. The algorithm would consider all feasible time steps for transferring the operand. It would consider them so as to monotonically recede from the time step indicated in the offspring. Thus, if the feasible transfer times for the left attribute were time steps 2 and 3, then the algorithm would first consider time step 3 and then time step 2. Let us assume that it is feasible to transfer the operand in time step 3 from A-block '1'. Now while considering the second operand, suppose it is not feasible to transfer it from A-block '0', as indicated in the offspring attribute. The algorithm would then try to source the operand from other A-blocks. Let us assume that it succeeds in sourcing the operand from A-block '1'. This operation is now scheduled in time step 3. □

A scheduling heuristic is also used intermittently with the intention of improving the quality of the solutions in the population. The heuristic may be used while transferring operations from the ready list to the active list (line 4, in figure 6.10). Normally operations are selected from the active list for scheduling at random (line 5, in figure 6.10). However, if the heuristic is being used, then the operations are chosen from the list on the basis of the scheduling heuristic. The application of the heuristic is explained in the paragraph below.

While trying to schedule an operation in an A-block at a specific time, it is first checked whether the FU can be used without input-conflict, output-conflict or execution-conflict. The availability of operands is checked next. If an operand is not present in the current A-block, then it needs to be transferred from another A-block, in the current time step or a preceding one. For an operand or variable to be transferred at a particular time, a free transfer path from the source to the destination needs to be identified. Thus, a free bus and a free access link at the source and destination A-blocks have to be found. An operation can be scheduled in an A-block only if the FU can be used without conflict and the operands are available or can be made available.

The in ward transfer of a variable currently unavailable is made as follows. The variable can be transferred any time between the first time step and the current time step. It can be transferred from any A-block where the variable is available at the time the transfer is being attempted. The transfer is first attempted at the time and from the A-block indicated for that value in the solution. If the transfer cannot be satisfied this way then other time and A-blocks are considered in the following order: $t_s - 1, t_s - 2, \dots,$ t_d and $(b_s + 1)$ mod tot_b, $(b_s + 2)$ mod tot_b, ..., respectively (i.e. as late as possible), where t_s is the desired time of transfer, t_d is the time from where the variable is defined, b_s is the desired source A-block and tot_b is the total number of A-blocks. The order of scanning is block major (i.e. the block index changes slower).

```
procedure complete_solution()
1.  prepare initial ready lists of operations and variable assignments.
2.  while (operations and assignments remain to be scheduled)
3.  {   decide whether heuristic scheduling is to be used
             or priority will be given to transfers.
4.      transfer some operations to active list from ready list.
5.      try to schedule active operations on units indicated in the chromosome.
6.      if (priority_trn_flag)   try to schedule active assignments.
7.      try to schedule remaining operations on other units.
8.      if (an operation has been scheduled) then redo iteration.
9.      if (not priority_trn_flag)   try to schedule active assignments.
10.     update ready list of operations.
11.     update status of FUs.
12.     bring in ready transfer candidates to active transfer list.
13.     move some transfers from ready list to active list.
14.     update data structures and flags.
15.     increment the time step.
16. }
```

Figure 6.10: Completion algorithm.

Application of Heuristic: It has been noticed while designing the system that optimization obtained only by applying the genetic operators of mutation and crossover, with small enough population sizes, do not perform very well in practice. It was therefore decided to use a heuristic, to be applied stochastically, in the completion algorithm to schedule operations. The heuristic is based on a weight computed for each operation, which is defined as $w_i = \sum_{o_j \succ o_i}(d_j + W)$, where o_i and o_j are operations, o_j is a successor of o_i and W is a fixed positive value. While selecting an operation to schedule using the heuristic, it is chosen at random in proportion to its computed weight. A stochastic choice is made to avoid excessive bias to a particular decision.

The heuristic is applied at two places, while selecting operations from the active list and while transferring operations from the ready list to the active list. While completing a solution it is applied with a certain probability that is taken as a parameter. Even when it is being applied it is turned on and off at random as scheduling progresses through the time steps to avoid excessive bias from the heuristic which might undo the evolutionary process.

Replacement: The replacement policy is designed to ensure that all solutions generated stay in the population for at least one iteration. This is done by introducing all the new solutions generated

through crossover during one generation of the GA into the population and replacing an equal number of existing solutions. The offsprings are stored in an adjoint pool, to be introduced into the main population once all the offsprings from the current generation are produced. The solutions to be replaced are mostly chosen at random. This could lead to removal of apparently good solutions, with low cost, from the population. To counter this, a scheme has been used to retain the solutions with better costs and at the same time maintain a diversity of FU configurations in the population.

6.3.4 Register Allocation and Binding

The task accomplished in this phase can be stated as follows. Given a set V of variables and an A-block A, identify the subset of variables such that each one can be bound to the same register in A. For this, a compatible relation R over V is defined as follows. $v_1 \; R \; v_2$ iff their lifetimes in A do not overlap.

The subtasks involved in *register allocation and binding* are:

- Lifetime analysis of variables in an A-block from the schedule of operations and the bus transfers of variables over the global buses,
- Determination of the compatibility relation and constructing the compatibility graph from the lifetimes of variables in an A-block, and
- Register optimization by computing the minimum number of registers required in the data path of an A-block from the lifetimes of variables and the compatibility graph in an A-block.

Lifetime analysis

Since a variable may reside in more than one A-block at the same time, it is required to speak of its lifetime corresponding to an A-block. Let the lifetime of the variable v with respect to an A-block A_i be denoted as $\langle \mathrm{v}, A_i \rangle$. Also, the lifetime of a variable may span across the basic blocks. Let the of lifetime of $\langle \mathrm{v}, A_i \rangle$ in a basic block b be represented by a 3-tuple $\langle b, s, e \rangle$, where

- b denotes a basic block in the scheduled CDFG,
- s denotes the control step of the basic block b, where the current lifetime has started consequent to a new definition of the variable v, and
- e denotes the last use of the variable v in the basic block b.

Variables are of two types, program variables that span across basic blocks and temporaries whose lifetimes remain confined within a basic block. So, the lifetimes of a program variable v_i with respect to an A-block A_j, i.e. $\langle v_i,\ A_j\rangle$, can be represented as an ordered list of lifetimes. Let $\langle v_i, A_j\rangle := \langle\langle b_1, s_1, e_1\rangle, \langle b_1, s_2, e_2\rangle,\ , \ldots, \langle b_k, s_k, e_k\rangle\rangle$, where, for any basic block b_l, if $\langle b_l,\ s_1,\ e_1\rangle$, $\langle b_l, s_2, e_2\rangle \in (v_i, A_j)$, and $\langle b_l, s_1, e_1\rangle$ precedes $\langle b_1, s_2, e_2\rangle$, then $e_1 < s_2$. The list $\langle v_i, A_j\rangle$ contains at most one "open" lifetime which is the last one in the list and all the preceding ones are closed. On the other hand, the lifetime of a temporary variable v_i with respect to an A-block A_j, i.e., $\langle v_i, A_j\rangle$, is of the form $\langle b_k, s_1, e_1\rangle$, which is either *open* or *closed*. Closed lifetimes are those whose last use of value defined at control step s of basic block b has been found. Open lifetime is such a value defined at control step s of the basic block b, its last use to be found. Open lifetime switches to closed in 2 cases,

1. new definition of the $\langle v_i, A_j\rangle$ in basic block b in A-block A_j,

2. execution reaches end of the basic block b.

While Computing lifetimes for the variables the following two pieces of information need to be considered.

- bus transfers of a variable over the global buses between A-blocks and the I/O ports, and

- Schedule of the operations.

Bus transfers: Bus transfer of a variable v from A-block A_i to A-block A_j over the global buses at control step k defines an open lifetime of v in A_j. Also, the last use of v in A_i is changed to k.
Let us consider a bus transfer $v(A_i \rightarrow A_j)$ at control step k of basic block b. If $\langle v,\ A_j\rangle$ is added to list of program variable lifetimes, then close the currently open program variable lifetime of $\langle v,\ A_j\rangle$ if any exists and open a lifetime for $\langle v,\ A_j\rangle$ with $\langle b,\ k, -\rangle$; last use of $\langle v,\ A_j\rangle$ is to be found out. Upgrade last use of $\langle v,\ A_i\rangle$ till k, i.e., $\langle b, s, k\rangle$.
If it is found that the last use of variable v in A_j is also k, i.e., lifetime is $\langle b,\ k, k\rangle$, then this lifetime can be removed from $\langle v,\ A_j\rangle$ because the variable v is not required to be stored in any register in A_j and can be directly fed into the input of the FU of A_j in time step k.

Schedule of the operations: Let an operation $v_d \Leftarrow v_{s1} \ <op> \ v_{s2}$ of basic block b be scheduled in the ith time step in the A-block A_k. The schedule of this operation effects the lifetimes of the variables involved as follows. If v_d is a program variable and it has an open lifetime, then that lifetime needs to be closed. Also, a lifetime for $\langle v_d, A_k\rangle$ with $\langle b, i+1, -\rangle$ is opened. The last use of $\langle v_{s1},\ A_k\rangle$ and $\langle v_{s2},\ A_k\rangle$ are upgraded till i, i.e., $\langle b,\ s,\ i\rangle$.

Algorithm 5 constructs the lifetimes of both program and temporary variables in all the A-blocks. The top level module computing the lifetimes of variables in the A-blocks is *find_Lifetimes*, which takes as input parameters the scheduled CDFG and the bus transfers, the set of live variables along with the location availability in the A-blocks for each basic block and returns the lifetimes of the program and the temporary variables.

Determination of the compatibility relation

Computation of minimal number of registers in an A-block A_i from the list of program and temporary variable lifetimes in A_i requires the construction of compatibility graph in which nodes represent the variables appearing in A_i and edges between two variables exists if they have non-overlapping lifetimes. Compatibility relation exists between lifetimes of variables in A_i. It is not necessary to examine the lifetimes in $\langle v_1, A_i\rangle,\ \langle v_2, A_j\rangle$ where $A_i \neq A_j$.
Three different cases arise while constructing the compatibility graph between the variables. They are as follows.

1. Two temporary variables $\langle v_1, A_i\rangle, \langle v_2, A_i\rangle$ with lifetimes $\langle b_1, s_1, e_1\rangle, \langle b_2, s_2, e_2\rangle$ can be mapped to the same register, iff
$$b_1 \neq b_2 \ \vee \ ([s_1, e_1] \cap [s_2, e_2] = \phi) \tag{6.5}$$

2. A program variable $\langle v_1, A_i\rangle := \langle\langle b_1, s_1, e_1\rangle, \langle b_1, s_2, e_2\rangle, \cdots, \langle b_k, s_k, e_k\rangle\rangle$ and another temporary variable $\langle v_2, A_i\rangle$ with the lifetime $\langle b_l, s_l, e_l\rangle$ can be mapped to the same register, iff
$$b_j \neq b_l \ \vee \ ([s_j, e_j] \cap [s_l, e_l] = \phi),\ 1 \leq j \leq k \tag{6.6}$$

3. Two program variables $\langle v_1, A_i\rangle,\ \langle v_2, A_i\rangle$ with respective ordered lifetimes,
$\langle v_1, A_i\rangle := \langle\langle b_1, s_1, e_1\rangle, \langle b_1, s_2, e_2\rangle, \cdots, \langle b_k, s_k, e_k\rangle\rangle$, and
$\langle v_2, A_i\rangle := \langle\langle b_1, s_1, e_1\rangle, \langle b_1, s_2, e_2\rangle, \cdots, \langle b_l, s_l, e_l\rangle\rangle$

Algorithm 5 Computation of lifetimes of variables from the scheduled CDFG.

```
procedure find_Lifetimes()
Input: scheduled CDFG and the bus transfers, live variables set along with location availability
    in A-block for each basic block,
Output: list of lifetimes for both program and temporary variables in the A-blocks
for each basic block b in the scheduled CDFG do,
{
    for each variable i, ⟨v_i,A_k⟩ ∈ in_b do,
    {
        create open lifetime of ⟨v_i,A_k⟩ := ⟨b,0,−⟩;
        append new open lifetime of ⟨v_i,A_k⟩ to program variable list in A_k;
    }
    for each control step i of b do,
    {
        if any bus transfers in i with v(A_j → A_k)
        {
            update the lifetime of ⟨v,A_j⟩;
            close currently open lifetime of ⟨v,A_k⟩ if exists;
            create a new open lifetime of ⟨v,A_k⟩ := ⟨b,i,−⟩;
        }
        for each operation j of b, O_bj : v_d ⇐ v_s1 < op > v_s2 scheduled at i do,
        {
            A_k := A-block on which O_bj scheduled;
            put i as the last use of v_s1 and v_s2 in their respective life time (v_s1,A_k) and (v_s1,A_k);
            close currently open lifetime of ⟨v_d,A_k⟩ if exists;
            create a new life time of ⟨v_d,A_k⟩ := ⟨b,i+1,−⟩;
        }
    }
    steps := number of control steps of the basic block;
    for each variable i, ⟨v_i,A_k⟩ ∈ out_b do,
        update lifetimes of the output program variables ⟨v_i,A_k⟩ to steps+1;
    close the currently open lifetimes of variables;
    for each program variable v_i in each A-block A_j
        if any lifetime in ⟨v_i, A_j⟩ is created through bus transfer and has same s and e, then
        remove that lifetime from ⟨v_i, A_j⟩;
    for each temporary variable v_i of A-block A_j
        if ⟨v_i, A_j⟩ is created through bus transfer and has same s and e, then remove ⟨v_i, A_j⟩;
}
```

can be mapped to the same register, iff

$$b_m \neq b_n \;\vee\; ([s_m, e_m] \cap [s_n, e_n] = \phi),\ 1 \leq m \leq k,\ 1 \leq n \leq l \tag{6.7}$$

Register Optimization by Clique partitioning

From the compatibility graph in each A-block A_i, the minimal subset of maximum complete subgraphs is computed. The variables or nodes in these complete subgraphs have been mapped to a single register in the data path, due to their non-overlapping lifetimes in A_i. Maximum complete subgraphs are computed using the *clique partitioning algorithm* which finds the minimal number of cliques (complete subgraphs), given a graph. Due to the exponential complexity of the clique partitioning algorithm, the heuristic given in [77] has been employed to find the minimal number of cliques from the compatibility graph. So, the total number of registers needed in A_i is the number of cliques found from the clique partitioning method.

6.3.5 Data-path and Controller Generation

Data-path Generation

The interconnection topology that supports data transfers between the storage and the functional units is one of the factors that has a significant influence on the data path performance. The complexity of the interconnection topology is defined by the number of interconnection units between any two ports of functional and storage units. Each interconnection unit can be implemented with a multiplexer or a bus, the latter has been used in our data path.

Data path in each A-block is a collection of a functional unit, a register file and interconnection wires and switches, which connect/disconnect two wires depending on the control signal. A switch is a hardware (implemented using FETs) placed between two wires; depending on the control signal that it receives, it connects/disconnects the two wires. There are two types of switches used in our design, unidirectional switch and bidirectional switch, A unidirectional switch, when closed, can transmit data in one direction; a bidirectional switch, when closed, transmit in both directions.

Registers in an A-block are organized as a register file (register bank). The number of writes to a register file during a control step is limited by the number of write ports (given as an architectural constraint). All register transfers, except the local assignments, go through functional units in an A-block; direct interconnection of two functional units are not allowed. We, therefore, need

interconnection units to connect the output ports of storage units to the input ports of functional units (i.e the input interconnection network) and the output ports of functional units to the input ports of storage units (i.e, the output interconnection network).

Registers are connected to both left port and right port of the functional unit through unidirectional switches. Only those registers that hold the first operand in any of the operation connect to the left port of FU and those registers that hold the second operand in any of the operation connect to the right port of FU. Output port of the FU is connected to those registers, which hold the result variable of any operation. If both the operands of an operation are available in the same A-block, then the registers holding the operands are connected to the FU. If any of the operand is not available in the same A-block, it needs to be transfered from another A-block where it resides. If the transferred variable is not used further in the destination A-block, then it is not needed to be stored in this A-block. In this case, it is better to provide the operand to the FU directly from the global buses. In some cases, it is needed to transfer the result of any operation to another A-block; so the output port of the FU is also connected to global buses.

Controller Generation

The basic functionalities of the controller unit is shown in figure 6.11. It consists of mainly two functional blocks namely *next state generator* and *control signals generator*. The control signal generator block has two parts. The part 1 of it involves the generation of control signals to the out-switches, bidirectional switches and the FUs and the part 2 involves generation of the control signals for the in-switches and the write enable signals to the registers. In the following, this two functional blocks of the consists are discussed.

Next-state Generation

Finding the next state in the finite sate machine (FSM) generated from the data-path is done by the sequence generator which is implemented as a counter. The inputs to the sequence generator are the status signals generated from the functional units and the present state of execution and the outputs of the sequence generator is the next state of execution.

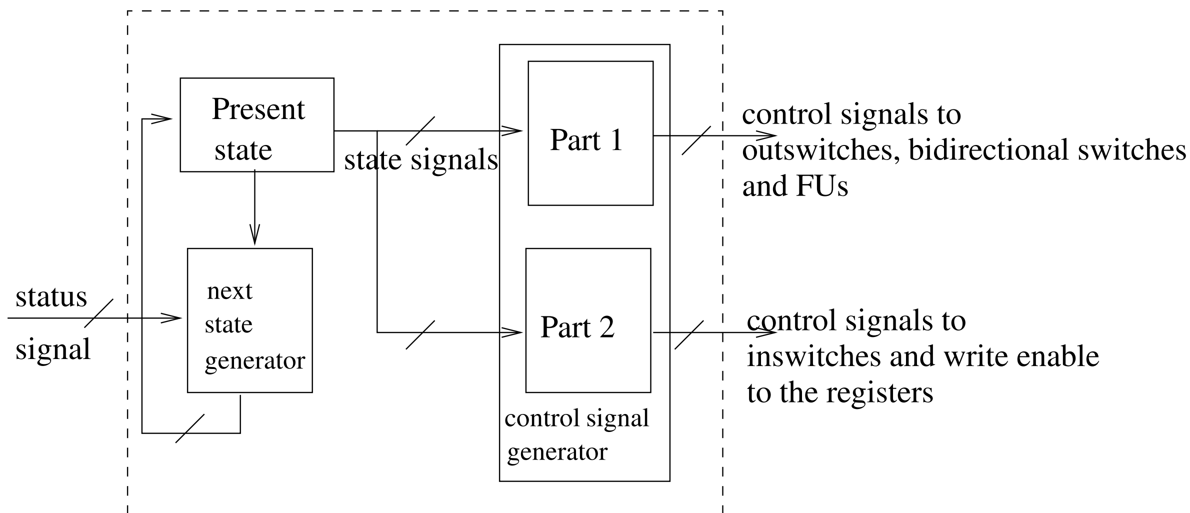


Figure 6.11: The block-diagram of the controller

Control Signal Generation

The controller generates control signals for the functional units in all the A-blocks, for the switches inside each A-block, and for those outside the A-blocks and the write enable signals for the registers. Depending on the state of the FSM, the controller assigns '1' 0r '0' to each of its control signal. Except the write enable signals for the registers, all the remaining control signals are generated at the start of the control step. The write enable signals for the registers are generated at the end of the control step, that is, at the start of the next control step. The generation of write enable signals to all registers in the data-path are explained in figure 6.12.

The timing diagram of control, data and register write enable signals are shown in figure 6.13. As shown in figure 6.13, the state signals are generated by the counter at the raising edge of the clock. State signals are decoded by an X-decoder to produce control signals for the functional units and the switches in the data-path. The control signals for the in-switches are phase shifted by 180 degree's using D-latches. Therefore, the phase shifted control signals are started at the falling edge of the clock. Performing an *and* operation of the write select signals with clock produces the write enable signals. Thus, a write enable signal is a pulse produced at the start of the next control step. The bus lines contain the data to be written into registers or to be sent to the input ports of the functional units. The output of the functional unit and the internal buses are connected to the input ports of the registers. Data from the functional unit is delayed by the time it takes to execute an operation. Therefore, the data present at the input ports of a register needs some time to stabilize. As the data present at the inputs of the registers are stable at the end of the control step, all writes

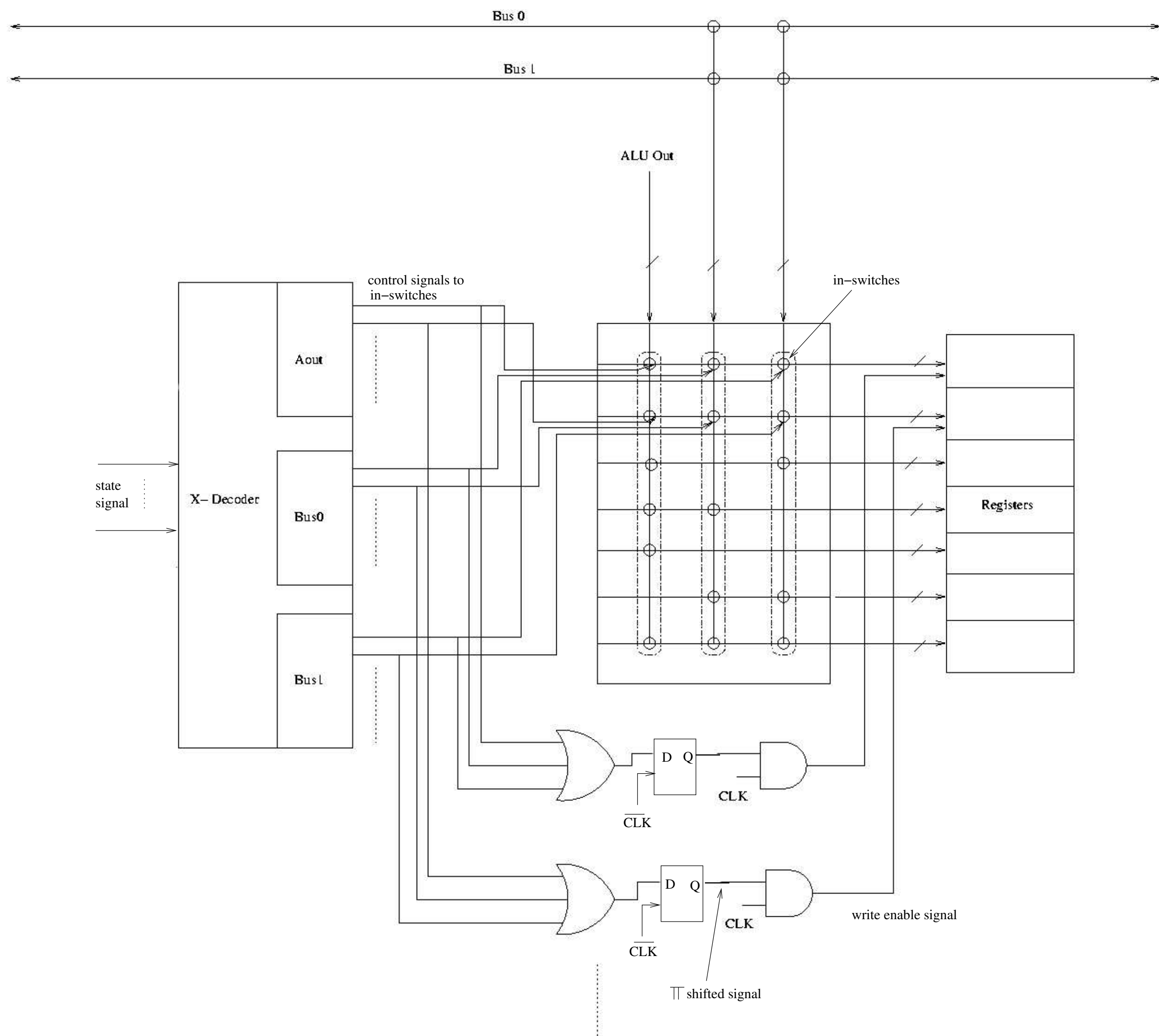


Figure 6.12: Generating Write Enable signals.

into registers are done at the raising edge of the write enable signal, which is at the starting of the next control step.

6.3.6 Verilog Code Generation

The final step in the high level synthesis process is to generate the HDL code for the resultant RTL description. There are many hardware description languages like Verilog, VHDL, SystemC, etc., to describe the hardware. Verilog has been used to describe the RTL generated from the synthesis process. There are two main functional modules in the process of generating the Verilog HDL, the data path generator and the controller generator. The RTL Verilog code generated by SAST

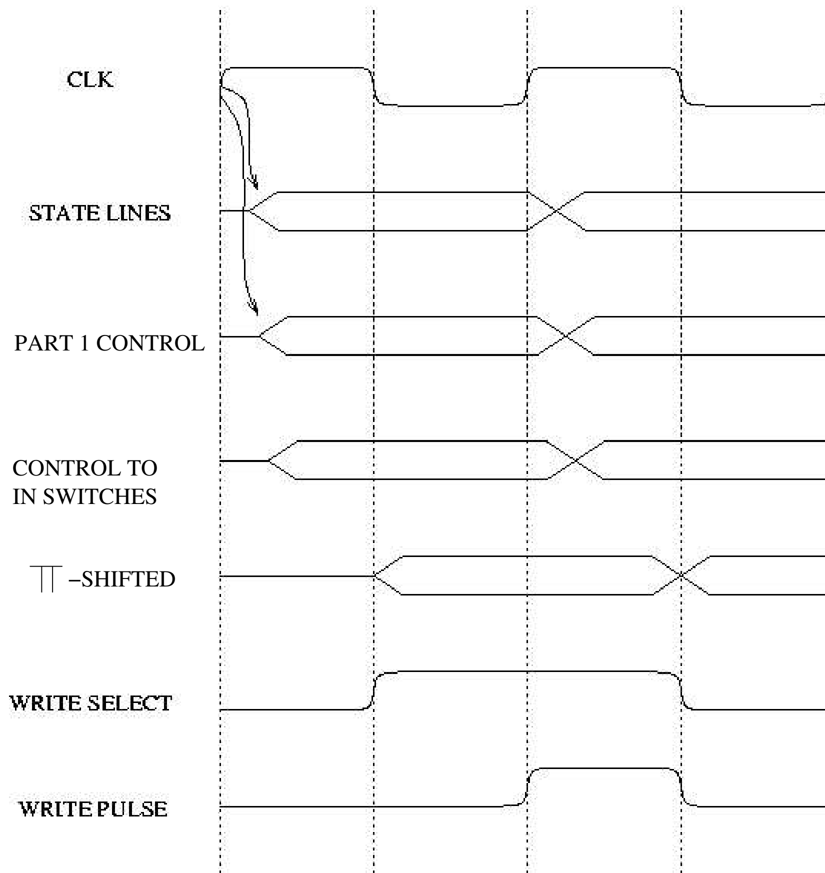


Figure 6.13: Timing diagram for control and write enable signals.

can directly be used as input for any commercial logic synthesis tool such as Synopsys Design Compiler for further synthesis.

6.4 Generation of the FSMDs for Verification

Our proposed verification methodologies are incorporated with SAST. Therefore, SAST produces the FSMDs from the input behaviour and the output of each synthesis phase. Accordingly, four FSMDs are constructed in SAST. They are M_0 from the input behaviour, M_1 from the scheduled behaviour, M_2 from the allocation and binding result and finally M_3 from the output RTL behaviour produced by SAST. In the following, the FSMD construction mechanisms are discussed.

6.4.1 Construction of FSMD from the CDFG

The algorithm 6 constructs the FSMD M_0. The top level module of the algorithm is ConstructM_0. The input to this procedure is a CDFG with list of operations in each block and output is an FSMD M_0. The procedure starts with the start node of the CDFG as *cdfgNode* and the reset state of FSMD M_0 as the $fsmdState$. In each of its invocation, the function considers a block of the CDFG and produces the corresponding portion in M_0 starting from the state $fsmdState$. If the current processing block is a basic block, it constructs the DFG of the operations in that basic block first. Then all the operations in one level of DFG are mapped to one transition of the FSMD. There would be one state in M_0 for each level of the DFG. The control flow is also properly maintained. If the current processing block is a conditional block, then a divergence in flow is created in the FSMD M_0 with the proper transition condition.

Example 12 *The CDFG of the differential equation solver (DIFFEQ) is given in figure 6.14(a) and the control flow of the CDFG is explicitly shown in figure 6.14(b). The FSMD M_0 of the DIFFEQ behaviour constructed by the algorithm 6 is shown in figure 6.14(d). Let us consider the invocation of the function for the basic block $B1$. So, the function invokes with constructM_0 (B1, q_{02}). The DFG of the operations in B1 is formed first and it is shown in figure 6.14(c). Now, the function considers the operations in the level 1 of the DFG. A new state q_{03} and a transition $q_{02} \rightarrow q_{03}$ are added in M_0 and shown in figure 6.14(d). All other levels of the DFG are treated in a similar way. The successor node C1 of B1 is not visited. So, the function is called as constructM_0 (C1, q_{06}). Here, successorT node is B1 which is already visited. So, the function adds a transition $q_{06} \rightarrow q_{02}$ in M_0 with condition $x < a$, where q_{02} is the start state in M_0 for the basic block B1. The successorF node of C1 is B2 which is not visited. So, the function adds a new state q_{07} and a transition $q_{06} \rightarrow q_{07}$ with transition condition $!x < a$ in M_0.*

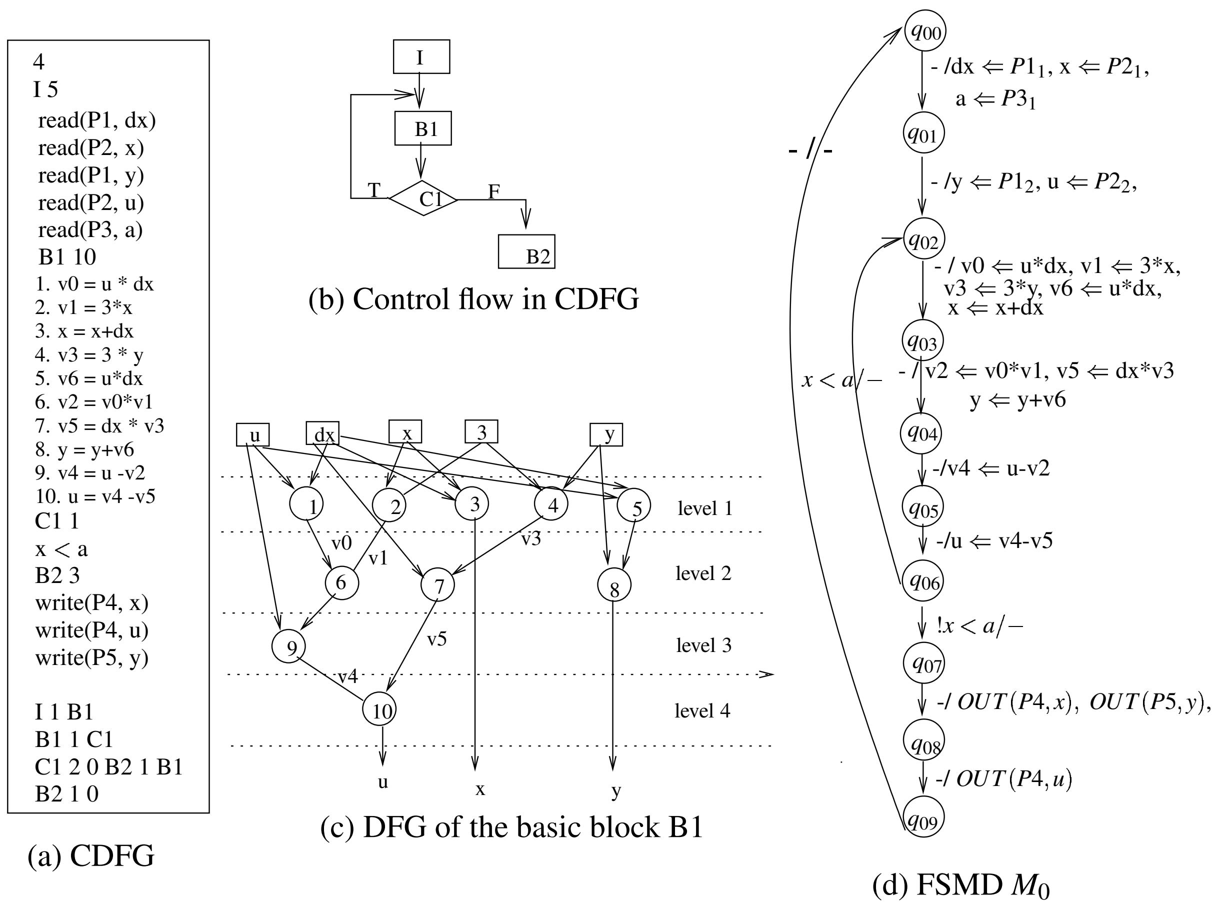
□

6.4.2 Construction of FSMD from the Scheduled Behaviour

The scheduled behaviour is also represented as a CDFG in SAST. In the scheduled CDFG, each basic block consists of a set of operations with time step assigned to each operation. The construction procedure of the FSMD M_1 from the scheduled behaviour is almost the same as the construction procedure of the FSMD M_0 from the CDFG. The only difference is that instead of considering

Algorithm 6 Construction of the FSMD M_0 from CDFG

```
procedure: ConstructM0 (cdfgNode, fsmdState)
begin
  if (cdfgNode is a basic block)
  { Mark cdfgNode as visited;
    Construct the DFG of the operations in cdfgNode; /* Let there be l levels in DFG */
    Let the successor node of cdfgNode be succNode;
    for level i = 1 to l of DFG
    { if (level is l)
      { if (succNode is not NULL and visited)
        { Let representation of the succNode start with the state succState in M0;
          nextState = succState;
        } else /* succNode is not NULL and not visited */
              Add a new state newState in M0;
      } else /* level is not l */
            Add a new state newState in M0;
      Add a transition fsmdState → newState in M0 with TRUE as condition and
        the operations in level i as transformation;
    } /* end for */
    if (succNode is not NULL and not visited)
      constructM0 (succNode, nextState);
  } /* end -if (cdfgNode is a basic block) */
  else /* cdfgNode is a control block */
  { Let the cdfgNode contains the relational expression exp;
    Let succNodeT be the successor node of cdfgNode when the condition becomes true and
      succNodeF be the same when the condition becomes false;
    if (succNodeT is not visited)
    { Add a new state newState in M0; Add a transition fsmdState → newState with
        condition exp and no transformation;
      constructM0 (succNodeT, newState);
    } else /* succNodeT is already visited */
        Add a transition in M0 of the form fsmdState → succState with condition exp and no tran-
        sformation; /* Let representation of the succNodeT start with the state succState in M0 */
    if (succNodeF is not visited)
    { Add a new state newState in M0; Add a transition fsmdState → newState with
        condition !exp and no transformation;
      constructM0 (succNodeF, newState);
    } else /* succNodeF is already visited */
        Add a transition in M0 of the form fsmdState → succState with condition exp and no tran-
        sformation; /* Let representation of the succNodeF start with the state succState in M0 */
  }
  if (all the nodes of CDFG are visited) Draw a transition with no condition and transformation
     from each state of with no outward transition to the reset state of M1;
end
```



(a) CDFG

(b) Control flow in CDFG

(c) DFG of the basic block B1

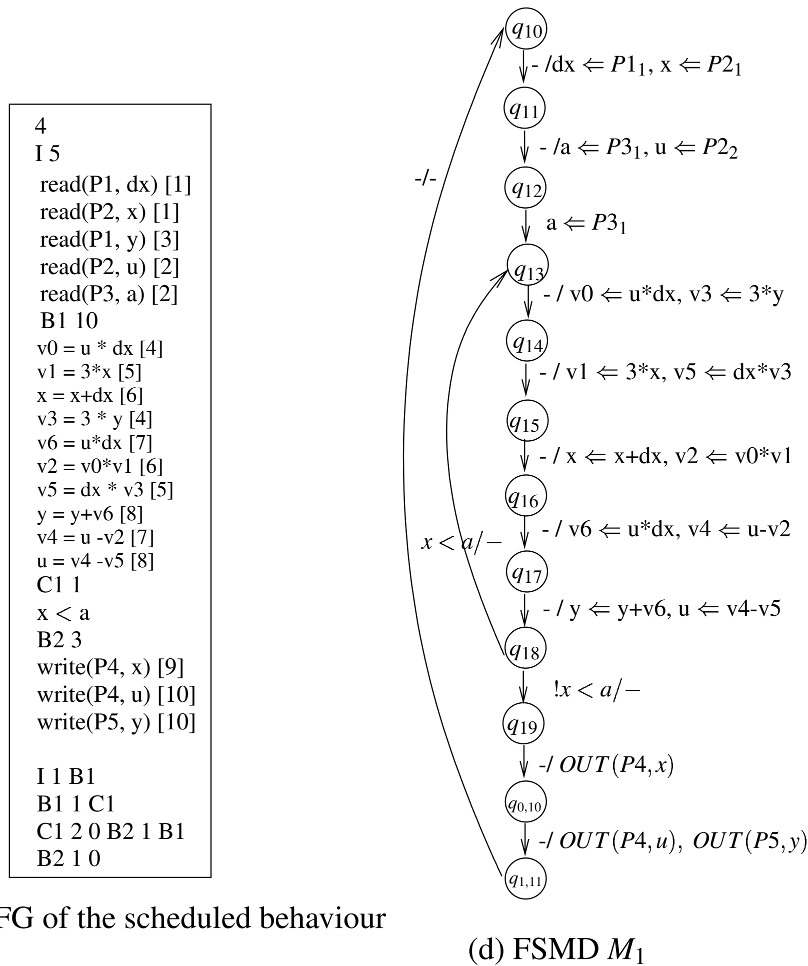
(d) FSMD M_0

Figure 6.14: DIFFEQ example to show how the FSMD M_0 is constructed from CDFG

each level of the DFG of a basic block, all the operations scheduled in one time step are mapped to the same transition. The conditional blocks are treated in a similar manner. The scheduled behaviour of the DIFFEQ example is given in figure 6.15 (a). The value within the brace for each operation represents the time step in which an operation is scheduled. For example, the operation $v0 = u * dx$ is scheduled in 4th time step. The FSMD M_1, constructed from the scheduled CDFG for this example, is given in figure 6.15 (b).

6.4.3 Construction of FSMD from the Allocation and Binding Results

The algorithm 7 constructs the FSMD M_2. The top level module of this algorithm is constructM_2. The inputs to this module are the FSMD M_1, the scheduled operations and bus transfers and the register mapping information of the variables computed during allocation and binding phase. The control structure of the FSMD M_2 is the same as that of the FSMD M_1. The variables in the condition and the operations of each state transition of M_1 are replaced by the appropriate registers



(a) CDFG of the scheduled behaviour

(d) FSMD M_1

Figure 6.15: DIFFEQ example: the scheduled behaviour and the FSMD M_1

in M_2. Also, for each bus transfer scheduled, one register transfer operation is added in the corresponding state transition of M_2. It is important to note that bus transfers do not have any effect in the FSMD M_1 because a bus transfer $v(x \rightarrow y)$ can be represented as $v \Leftarrow v$ which is true and no need to add such operation in M_1. The variable v, however, needs to be stored in one register, say R_{xm}, in A-block x. When this variable is transfered from the A-block x to the A-block y, it is necessary to store this variable in a different register, R_{yn} when v is not used in the same time step in y. In this case, it is essential to add a register transfer operation $R_{yn} \Leftarrow R_{xm}$ in the FSMD M_2.

6.4.4 Construction of FSMD from RTL Design

The process of generating the FSMD M_3 from the CP-DP informations of the generated RTL circuits is already discussed in section 5.3. This mechanism is implemented in SAST.

Algorithm 7 Construction of the FSMD M_2

procedure: **construct**M_2**( )**

begin

create a state q_{2i} in M_2 for each state q_{1i} of M_1;

for each transition $q_{1i} \rightarrow q_{1j}$ of M_1

create a transition $q_{2i} \rightarrow q_{2j}$ in M_2;

replace the variables with appropriate registers (found from life time information of the variable) in the relational expression representing the condition of the state transition of $q_{1i} \rightarrow q_{1j}$ and put the relational expression over the registers as the condition of the state transition $q_{2i} \rightarrow q_{2j}$;

for each operation $v_m \Leftarrow v_k < op > v_l$ in $q_{1i} \rightarrow q_{1j}$

{

Let this operation is scheduled in A-block A_o;

replace v_m with r_m, where r_m is a register in A_o that stores v_m in q_{1j};

If any register in A_o stores v_k at q_{1i}

replace v_k with r_k;

else if bus transfer $v_k(p \rightarrow o)$ in $q_{1i} \rightarrow q_{1j}$ and register r_k of A-block A_p stores v_k in q_{1i}

replace v_k with r_k;

else report("Lifetime of v_k is not defined in q_{1i}"); exit;

do the same for the variable v_l;

}

for each bus transfer $v(x \rightarrow y)$ in the state $q_{1i} \rightarrow q_{1j}$, if v has a lifetime in y

put a register transfer operation $R_{xm} \Leftarrow R_{yn}$ in $q_{2i} \rightarrow q_{2j}$, where v is stored in the register R_{xm} of the A-block x in the state q_{1i} and in R_{yn} of the A-block y in the state q_{1j};

end

6.5 Conclusions

The synthesis flow of the high-level synthesis tool SAST, developed in this work, is discussed in this chapter. SAST takes behavioural description in VHDL and produces a synthesizable RTL design in Verilog. It is an interconnection aware high-level synthesis tool as it produces a structured data-path. Our proposed target data-path structure is discussed. The synthesis of SAST starts with producing the CDFG from the input VHDL behaviour. In the preprocessing step, it converts the CDFG into an intermediate representation required for the subsequent synthesis phases. The next step is scheduling. The scheduler of SAST is GA-based which supports the structured data-path. The GA-based scheduler is discussed in detail. The minimum number of registers, required to store the variables, are found next. In the next step, the data-path and the controller are generated. Finally, the RTL design is encoded in Verilog. Each of these steps of SAST has been discussed in this chapter. SAST is a hand-in-hand synthesis and verification platform. It produces the FSMDs M_0, M_1. M_2 and M_3 required for the verification phases. The construction of the FSMDs have also been discussed.

Chapter 7

Experimental Results

7.1 Introduction

The SAST tool is implemented in 'C' language using the methodologies discussed in the previous chapter. The verification method for each phase of high-level synthesis (HLS) is implemented and integrated with SAST. The normalizer is also implemented and incorporated with the equivalence checkers. The tool has been run successfully on several high-level synthesis benchmarks. A case study depicting the synthesis flow of SAST is given as appendix A. A graphical user interface is created for this tool as shown in figure 7.1.

7.2 Synthesis and Verification Results

7.2.1 Effects of the Architectural Parameters on Synthesis Results

The tool has been implemented on an Intel Pentium 4, 1.70 GHz, 256MB RAM machine and run on several HLS benchmarks with different architectural parameters to examine the effects of the architectural parameters on the synthesis results. The results are tabulated in table 7.1. For each benchmark, the number of basic blocks in the CDFG (column 2), the number of operations in the behaviour (column 3), the number of control steps required to schedule the operations (column 8) and the number of registers (column 9) used for different combinations of the architectural parameters are shown in this table.

The following observations can be made on the results shown in table 7.1.

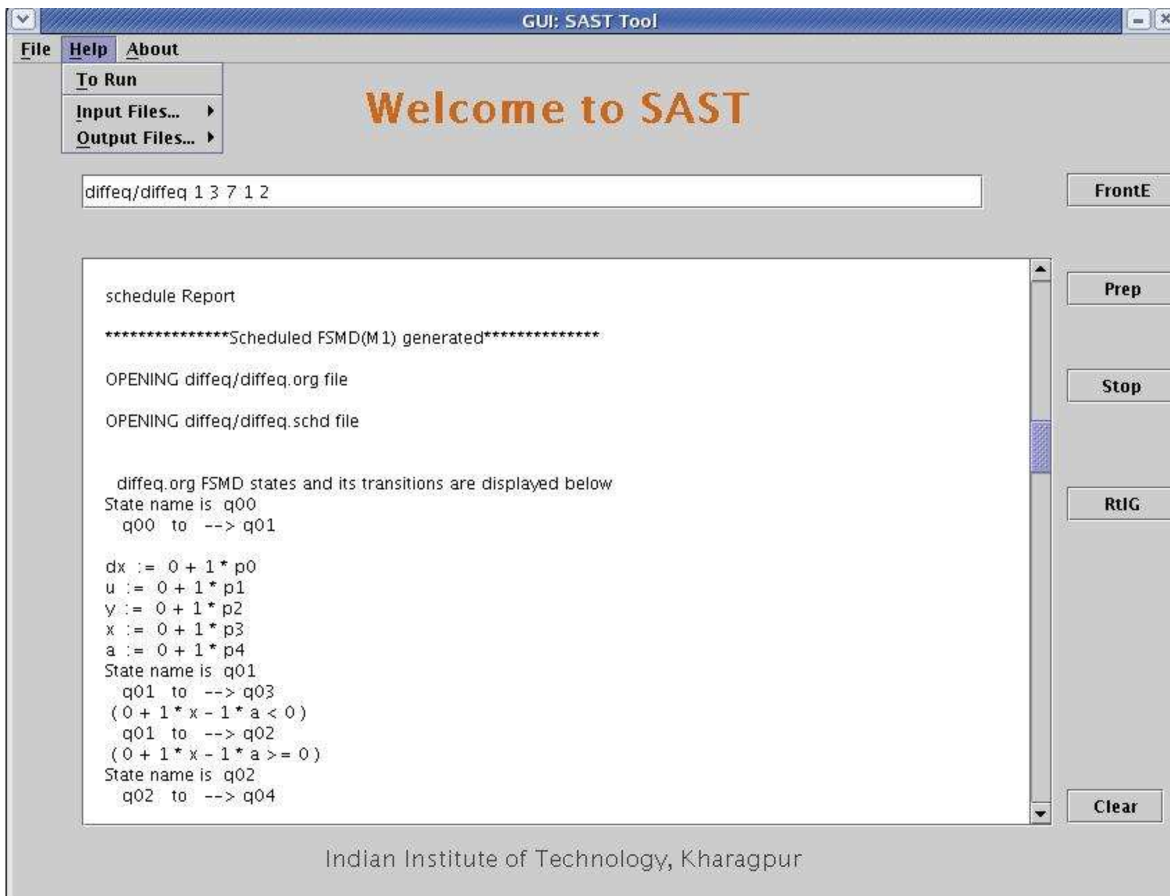


Figure 7.1: Graphical User Interface for SAST

- Use of more resources naturally reduces the number of time steps required to schedule the operations.

- A proper balance among the number of A-blocks, the number of buses and the number of access links needs to be maintained. Increasing the number of A-blocks without increasing the other two does not produce better scheduling results. For example, SAST takes 38 time steps to schedule the operations of the DCT example when the architectural parameters are fed as 2 A-blocks, 1 global bus, 1 access Link, 2 writes per time step (row 3 for DCT). Increasing only one A-block decreases the number of scheduling steps by two (row 2 for DCT) whereas, increasing one A-block and one global bus reduces the number of scheduling steps by 9 (row 1 for DCT).

- There is no well defined relationship between the number of registers required to store the variables and that of A-blocks.

Benchmark			*Arch. Parameters*				*Synthesis result*	
Name	#basic blk	#operation	#A-blk	#bus	#Acc link	#Write	#time step	#Reg
DIFFEQ	4	19	2	1	1	1	17	13
			2	2	1	1	13	11
			2	2	2	2	14	12
			3	2	1	1	12	10
			3	2	2	2	12	12
DCT	3	40	3	2	1	2	29	27
			3	1	1	2	36	29
			2	1	1	2	38	26
EWF	3	53	3	2	1	2	29	17
			3	2	2	2	28	19
			4	3	3	3	24	19
FIR	3	20	2	2	1	1	17	20
			2	2	2	2	16	19
			2	1	1	1	19	21
FP_ADD	9	16	2	2	1	1	14	9
			2	2	2	1	14	11
			2	1	1	1	15	9
MODN	10	16	2	2	1	1	15	12
			2	1	1	1	15	12
			2	2	2	1	13	9

Table 7.1: Synthesis results for some HLS benchmarks for different architectural parameters

7.2.2 Comparison with other Synthesis Tools

The output of SAST is compared with those of several existing HLS tools. In particular, the fifth order elliptic wave filter (EWF) [78] was worked out here and the results have been given in table 7.2. Synthesis systems SAM [9], STAR [8] and HAL [6] also try to minimize the interconnection cost. The greater predictability of the layout structure is ensured in CASS and CORBA synthesis systems. Results have been compared with these systems too and tabulated in table 7.2. It can be observed that the solution given by SAST is comparable to other tools but it produces a more regular data-path by avoiding random interconnections among the data-path elements.

Further, the RTL generated by SAST has been logic synthesized using Synopsis Design Analyzer with 0.18 micron CMOS 9 library of National Semiconductor Corporation, USA. Result has been illustrated for the EWF benchmark in figure 7.2. The ratio of interconnection overhead to that of the cell area is 1.1 percent (as reported by Synopsis Design Analyzer) for EWF. Similar results

System	#Time step	#Adder	#Multiplier	#Bus	#A-blk	#Acc-link	#Reg
SAST	28	3	2	1	3	1	16
COBRA	28	3	2	3	3	-	14
CASS	28	3	2	5	4	-	18
SAM	29	2	1	-	-	-	14
STAR	29	2	1	-	-	-	13
HAL	28	3	2	-	-	-	14
PSGA_SYN	28	3	2	-	-	-	12

Table 7.2: Comparison of results with a few other synthesis tools.

are also obtained for the other HLS benchmarks.

7.2.3 Verification vs. Synthesis

The outputs generated by SAST for several HLS benchmarks as shown in table 7.3. The number of states in the pre-scheduled FSMD M_0 and in the post-scheduled FSMD M_1, the CPU time and the memory usage are tabulated for each HLS benchmark. The number of states in M_2 and M_3 are the same as the number of states of M_1. It may be noted that the CPU time and the memory usage for overall verification process are much lower than those of the overall synthesis process because of hand-in-hand progress of synthesis and verification.

Name	*#state*		*CPU time in ms*		*memory used in kb*	
	M_0	M_1	verification	synthesis	verification	synthesis
DIFFEQ	9	12	6.372	54 x10^3	32.3	4910
EWF	17	35	4.740	127 x10^3	56.5	4436
GCD	7	4	9.340	21 x10^3	23.5	6318
DCT	10	29	3.546	163 x10^3	52.1	4631
TLC	7	8	10.660	101 x10^3	31.5	7375
MODN	6	7	12.128	90 x10^3	26.5	9654
PERFECT	9	6	11.128	53 x10^3	23.5	9987

Table 7.3: Results for different high-level synthesis benchmarks

The number of paths in the initial path covers and in the computed path covers, the observed number of iterations and the worst case number of iterations of the scheduling verification algorithm for different HLS benchmarks are shown in table 7.4. The number of paths in the computed path cover P_0 differs from that in the initial path cover P_0' when path-based scheduling approach is

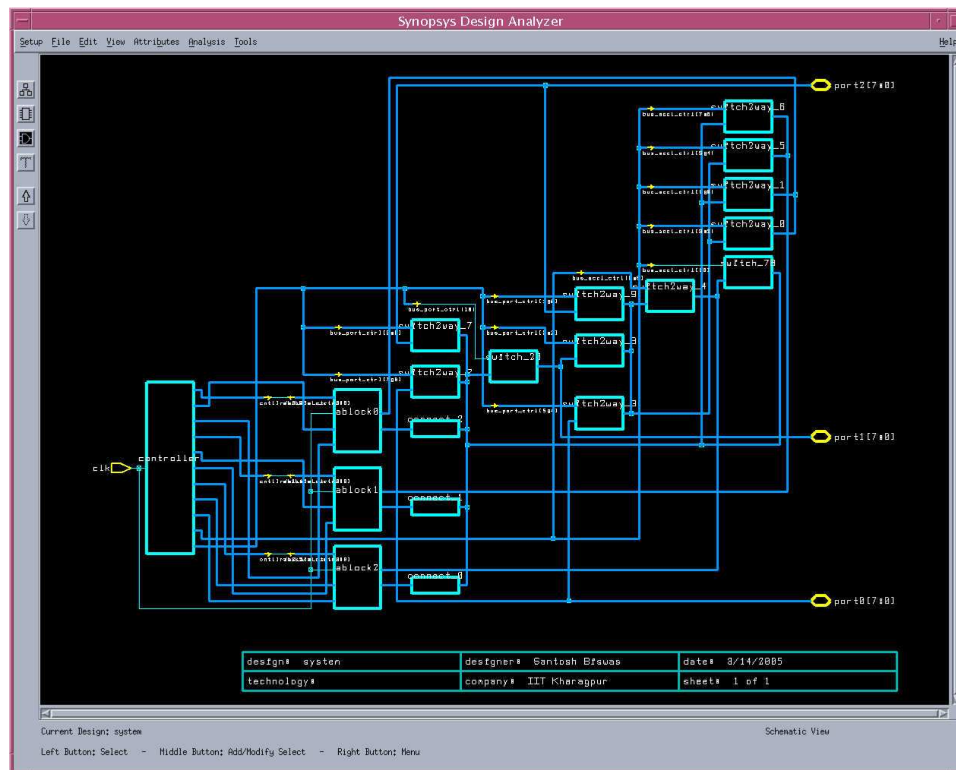


Figure 7.2: Synopsys DA output for EWF.

applied. This situation arises for the GCD, MODN, PERFECT benchmarks as shown in table 7.4. Our scheduling verification algorithm can successfully verify such situations. It is clear from the observed number of iterations of the scheduling verification algorithm for different HLS benchmarks that the worst case situation does not occur in practice. The bars in figure 7.3 represent the number of variables in the input behaviour (first bar) and the number of registers in the data-path generated by SAST (second bar) for each HLS benchmark. It is evident from this figure that SAST optimizes the number of registers and our verification works well in this case.

7.3 Conclusions

This chapter reports some experimental results on SAST over several HLS benchmarks. A comparative study is done on the HLS benchmarks by varying the architectural parameters to examine the effects of the architectural parameters on the synthesis result. Comparison with other existing HLS tools is also made. It has been observed that the results produced by SAST are comparable with the existing synthesis tools but it produces a better interconnection in the data-path. The memory

Name	*#paths*		*#path extension* required	*#iterations*	
	P'_0	P_0		actual	worst case
DIFFEQ	3	3	0	3	12^{13}
EWF	1	1	0	1	35^{36}
GCD	11	7	3	11	$5^4 4^5$
DCT	1	1	0	1	29^{30}
TLC	14	13	2	16	8^9
MODN	8	12	2	14	$2^7 7^8$
PERFECT	7	5	2	7	$2^6 6^7$

Table 7.4: Scheduling verification results for different high-level synthesis benchmarks

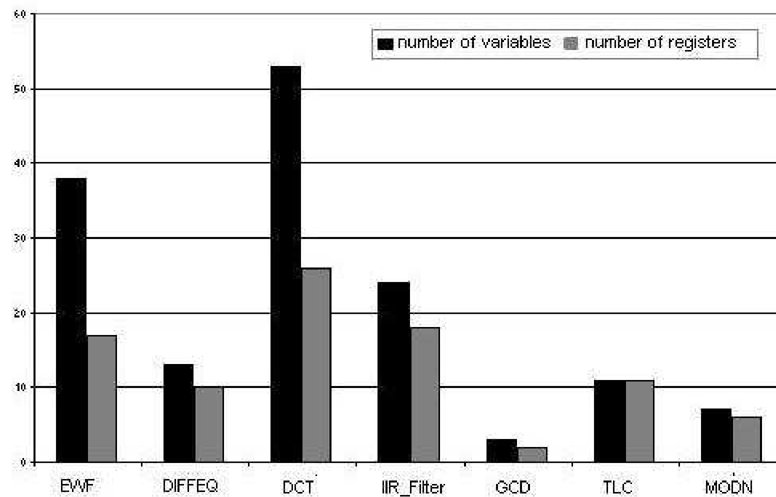


Figure 7.3: The typical number of variables and the number of registers required for different HLS benchmarks

usage and the CPU time of the verification process are much lower than those of synthesis process of SAST because the verification activity is planned to proceed phase-wise, hand-in-hand with synthesis. Experimental results on scheduling verification reveal that the worst case bound of complexity usually does not occur in practical situations. The results also show that SAST optimizes the registers and the control signals significantly and register sharing verification and data-path and controller verification work successfully in the respective phases for each benchmark.

Chapter 8

Conclusions and Future Scope of Work

8.1 Summary of the Work

In this thesis, a hand-in-hand verification and (high-level) synthesis of digital circuits has been proposed. We have underlined that a phase-wise verification technique with scope to handle the specialities of each synthesis sub-task separately is necessary for high-level synthesis (HLS). In this work, the correctness of HLS process is verified in three phases. Phase-I verifies the scheduling process. In phase-II, the allocation and binding process is verified against the scheduled behaviour. Phase-III ensures the correctness of the data-path interconnections and the controller.

A formal method for checking the equivalence between two FSMDs is formulated in chapter 2. This approach is applied to the first two phases of HLS verification. The equivalence of two FSMDs is defined. An equivalence checking method has been devised and validated. A normal form is used to represent the arithmetic expressions over integers in this work. Several simplifications that are carried out during normalization process are also proposed.

The verification of the scheduling process is discussed in chapter 3. The modifications required in our basic equivalence checking method are identified. A scheduling verification algorithm is presented. The correctness of the algorithm is proved and the complexity of the algorithm is analyzed. It is found that our algorithm works for both the basic block based scheduling algorithm and path based ones. It is also shown that our algorithm verifies successfully the code motion techniques like, *renaming, common sub-expression elimination, early condition execution, conditional branch balancing, conditional speculation*. The algorithm, however, fails for *speculation, reverse speculation and loop shifting*. The modifications of the algorithm are also proposed to handle these code

transformation techniques.

The verification of allocation and binding phase is discussed in chapter 4. The verification task is achieved in two steps. In the first step, the verification of FU allocation and binding is treated and in the second step, correctness of register sharing among the behavioural variables is verified. The verification of register sharing is done using the proposed equivalence checking method. It, however, needs some additional information like mapping between the states of two FSMDs and the mapping between the registers and the variables in each state. It is shown that our register sharing verification method is independent of the nature of the input specification and the register optimization schemes.

The verification of data-path interconnection and the controller behaviour is discussed in chapter 5. The verification task is performed in two steps. In the first step, an FSMD is constructed from the data-path information and the controller FSM. In the second step, a state based equivalence checking methodology is used to verify the correctness of the controller behaviour. A *rewriting method* is proposed which is used during the construction of the FSMD; the method finds the set of RT-operations performed by a given control assertion pattern in each state of the controller FSM. The correctness of this method has been proved and the complexity of the method is analyzed. Several inconsistencies and redundancies, both in the data-path and the controller, are revealed during construction of the FSMD. The state based equivalence checking method ensures the correctness of the controller FSM.

A high-level synthesis tool called *structured architecture synthesis tool (SAST)* to support hand in hand synthesis and verification has been developed through this work. SAST takes the behavioural description and produces a synthesizable register transfer level (RTL) code in Verilog. It is an interconnection aware high-level synthesis tool as it produces a structure data-path. The tool has been run on several HLS benchmarks to examine the effectiveness of the tool.

8.2 Future Scope of Work

- Multi-cycle execution of a slow operation means that the operation is scheduled across two or more control steps with faster operations scheduled concurrently over the cycles. This increases the utilization of the faster functional units since two or more operations can be scheduled on them during a single operation in the multicycle unit [1]. Pipelined execution

of the operations increases the utilization of the functional units. This method allows concurrent operation on several pairs of operands, each getting partially computed in one stage of the pipelined unit. The verification methodology proposed in this work can handle only the single-cycle and non-pipelined operations. Representation of multi-cycle and pipelined operations in an FSMD, distinguishing them from singe-cycled and non-pipelined ones and expanding the steps for computaion of the condition of exectution and data transformation of a path when the path contains such operations will be an interesting extension of this work.

- The proposed equivalence checker cannot process behavioural descriptions which contain arrays. Many real-life examples such as, lift controllers, digital signal processing circiuits and image processing circuits, involve arrays. Global memory has been provided for in the structured architecture used in this work for synthesis which are meant to store the arrays. Both the synthesis phases and the corresponding verification phases, however, need to be non-trivially extended to handle such circuits.

- Application of the proposed equivalence checking approach in other areas like embedded system design process [79, 80], automatic code generation schemes [81, 82, 83, 84, 85], etc., may be taken up as future research areas.

Appendix A

Synthesis with SAST: A Case Study

The synthesis flow of SAST is explained with the differential equation solver (DIFFEQ) [1] benchmark which is used to solves a system of first order differential equations.

The SAST takes four input files as input for each design. They are as follows:

1. $< designname >$*.beh*: The behavioural description of design.

2. $< designname >$*.arch*: Contains the architectural parameters.

3. $< designname >$*.opr*: Contains the operator library.

4. *ga_param*: Contains the genetic algorithm (GA) parameters.

Input behaviour: The behaviour design of DIFFEQ problem, encoded in SAST input format, is given below. We have taken 2 A-blocks, 2 global buses, 2 write ports and 1 access link as architectural parameters for this case study.

```
entity diffeq is
port (dx, u, y, x, a: in integer
  x1, u1, y1: out integer;
end diffeq;

architecture behav of diffeq is
begin
process(x, y, u)
variable v0, v1, v2, v3, v4, v5, v6 : int ;
begin
       while( x < a ) loop
        v0 := u*dx ;
v1 := 3*x ;
x  := x+dx ;
v2 := v0*v1 ;
v3 := 3*y ;
v4 := u-v2 ;
```

```
v5 := dx*v3 ;
v6 := u*dx ;
u  := v4-v5 ;
y  := y+v6 ;

end loop;

        x1 <= x;
        u1 <= u;
        y1 <= y;
end process;
end behav;
```

The CDFG generation: The behavioural complier (or the CDFG generator) of SAST produces the CDFG from the given input behaviour. The CDFG produced by the SAST for the DIFFEQ benchmark is given below.

```
4
B0 5
read (p0 , dx )
read (p1 , u )
read (p2 , y )
read (p3 , x )
read (p4 , a )

C0 1
x < a

B1 10
v0 = u * dx
v1 = 3 * x
x = x + dx
v2 = v0 * v1
v3 = 3 * y
v4 = u - v2
v5 = dx * v3
v6 = u * dx
u = v4 - v5
y = y + v6

B2 6
x1 = x
write (p7 , x1 )
u1 = u
write (p6 , u1 )
y1 = y
write (p5 , y1 )

4
B0 1 C0
C0 2 0 B1 1 B2
B1 1 C0
B2 0
```

Preprocessing: The next step of SAST is preprocessing. In this process, SAST converts the CDFG into an intermediate representation (IR). This IR actually consists of two files. They are as given below.

1. $<$ *designname* $>$ *.bb*: for storing IR of the basic blocks information.
2. $<$ *designname* $>$ *.po*: for storing IR of the partial order of the operations.

Scheduling: The output of the GA based scheduler of SAST is given below.

```
Block I
etime: 3;
************* schedule ******** idx=43 ********
 1: btrn:   ^5000( 0)  ^5001( 1)  xasg:   0( 0)-> 0( 0)   1( 1)-> 1( 1) opn:  < 0, 0>    < 1, 0>
 2: btrn:   ^5002( 4)  ^5001( 3)  xasg:   3( 0)-> 3( 0)   4( 1)-> 4( 1) opn:  < 3, 0>    < 4, 0>
 3: btrn:   ^5000( 2)             xasg:                   2( 1)-> 2( 1) opn:             < 2, 0>


Block B1
etime: 5;
************* schedule ******** idx=96 ********

 1: btrn:     1( 1)          xasg:                         opn:  < 0, 4>    < 1, 4>
 2: btrn:     4( 1)          xasg:                                    opn:  < 3, 4>    < 4, 4>
 3: btrn:   ^ 0( 0)          xasg:                2( 0)-> 1( 1)     opn:  < 6, 4>    < 2, 2>
 4: btrn:   ^ 3( 0)          xasg:                                    opn:  < 5, 3>    < 7, 4>
 5: btrn:                    xasg:   8( 0)-> 3( 0)   9( 0)-> 2( 1)     opn:  < 8, 3>    < 9, 2>


Block C1
etime: 1;
************* schedule ******** idx=1 ********

 1: btrn:                    xasg: ^ 4( 0)-> 4( 1) ^ 3( 1)-> 3( 0)     opn:             < 0, 1>


Block B2
etime: 2;
************* schedule ******** idx=0 ********

 1: btrn:   ^ 0( 1)          xasg:                                    opn:             < 0, 0>
 2: btrn:   ^ 1( 1)  ^ 2( 0)  xasg:                                    opn:  < 1, 0>    < 2, 0>
```

The above output is explained as follows. Scheduling information of each basic block is printed one after another. For each basic block, *etime* says the number of control steps needed to execute all the operations in that basic block. *btrn* describes bus transfers present in the scheduled output. This example is scheduled by using 2 global buses; thus there are two columns to represent the bus transfers, one for each global bus. Bus transfers are represented in the way `^a(b)` or `a(b)`, where *a* is the index of the variable to be transferred and *b* is some operation number or the index of the A-block from which the transfer is taking place. If the value of *a* is greater than or equal to 5000, the transfer is from input ports to A-blocks and the port number is calculated as *a* - 5000. and the value of *b* represents the operation number, which requires the transfer. If *a* is preceded by the symbol ˆ, then it means the variable in transfer is a result of an operation. *xasg* describes the definition of the program variable whose value has been used in the successor blocks. For example, 2 (0) $\rightarrow$ x (1) indicates that the result of the 2nd operation scheduled in the A-block 0 defines the

outgoing variable *x*. The updated value of *x* will be stored in A-block 1. The *opn* represents the scheduling of the operation. There are two columns following *opn* corresponding to the 2 A-blocks (provided as an architectural parameter) in which operations are scheduled. The column following *opn* represents the operations scheduled in the A-block 0 and the next row represents the same for A-block 1. For example, $< 7,4 >$ in the 4th row of basic block $B1$ represents the operation number 7 of operation type 4 (which is multiplication) is scheduled in A-block 1.

Lifetimes of the Variable: From the scheduler output, the lifetimes for the program variables and the temporary variables in each A-block are computed. A program variable's life spans across more than one basic block and a temporary variable's life spans only for one basic block. Each life span of a variable is described as $< s,e,b >$, where s is the control step in which the variable got assigned a new value in the basic block b, e is the control step in which the variable is last used in the basic block b. Output produced by SAST for DIFFEQ problem after the lifespans of each variable is calculated is shown below.

```
Program Variables in A Block 0
dx 0   2 4 I    0 6 B1  0 2 C1
u 1   3 4 I     0 4 B1  6 6 B1  0 2 C1  0 2 B2
3 1   0 0 B1

Temporary Variables in A Block 0
v1 2  1 2 B1
v0 3  2 2 B1
v3 4  2 3 B1
v2 5  3 4 B1
v5 6  4 5 B1
v4 7  5 5 B1


Program Variables in A Block 1
x 0   2 4 I     0 3 B1  4 6 B1  0 2 C1  0 1 B2
a 1   3 4 I     0 6 B1  0 2 C1
y 2   4 4 I     0 5 B1  6 6 B1  0 2 C1  0 2 B2
3 3   0 2 B1

Temporary Variables in A Block 1
dx 4  3 4 B1
v6 5  5 5 B1
```

Register Allocation and Binding: By constructing the compatibility graph from the lifetimes of all the variables in each A-block and by applying the clique partitioning algorithm, SAST finds the minimum number of registers required to store the variables in an A-block. The output produced by SAST after the register binding phase is shown below.

```
A-block 0
```

```
Register 0 has 1 elements : dx
Register 1 has 1 elements : u
Register 2 has 3 elements : v1 v2 v4
Register 3 has 2 elements : v0 v5
Register 4 has 1 elements : v3

A-block 1
Register 0 has 1 elements : x
Register 1 has 1 elements : a
Register 2 has 1 elements : y
Register 3 has 1 elements : 3
Register 4 has 2 elements : dx v6
```

The data-path and the controller Generation: Interconnections between the functional unit and registers in each A-block is done by internal buses and switches. SAST also minimizes the number of switches in the data-path. The final output produced by the SAST is the synthesizable Verilog RTL of both the data path and the controller. Verilog RTL produced by the SAST is given below.

```
/* --- Definition of Control Steps --- */

`define   CS0   4'b0000
`define   CS1   4'b0001
`define   CS2   4'b0010
`define   CS3   4'b0011
`define   CS4   4'b0100
`define   CS5   4'b0101
`define   CS6   4'b0110
`define   CS7   4'b0111
`define   CS8   4'b1000
`define   CS9   4'b1001
`define   CS10  4'b1010
`define   RESET 4'b1111

/* Defining Symbolicconstants to all operations in the design*/
`define  A0_ALUMUL  1'd0
`define  A0_ALUSUB  1'd1
`define  A1_ALUMUL  2'd0
`define  A1_ALUADD  2'd1
`define  A1_ALULT   2'd2

/* Defining Symboilc constants for each bit in all control signals */
`define  a0_accl0_r_dx_0 1'b0
`define  a0_accl0_r_dx_1 1'b1
`define  a0_accl0_r_u_0 1'b0
`define  a0_accl0_r_u_1 1'b1
`define  a0_alu_r_v0_v5_0 1'b0
`define  a0_alu_r_v0_v5_1 1'b1
`define  a0_r_u_alult_0 1'b0
`define  a0_r_u_alult_1 1'b1
`define  a0_r_dx_alurt_0 1'b0
`define  a0_r_dx_alurt_1 1'b1
`define  a0_accl0_r_v1_v2_v4_0 1'b0
`define  a0_accl0_r_v1_v2_v4_1 1'b1
`define  a0_alu_r_v1_v2_v4_0 1'b0
`define  a0_alu_r_v1_v2_v4_1 1'b1
`define  a0_r_v0_v5_alult_0 1'b0
`define  a0_r_v0_v5_alult_1 1'b1
`define  a0_r_v1_v2_v4_alurt_0 1'b0
`define  a0_r_v1_v2_v4_alurt_1 1'b1
`define  a0_accl0_r_v3_0 1'b0
`define  a0_accl0_r_v3_1 1'b1
```

```
`define  a0_r_dx_alult_0 1'b0
`define  a0_r_dx_alult_1 1'b1
`define  a0_r_v3_alurt_0 1'b0
`define  a0_r_v3_alurt_1 1'b1
`define  a0_r_dx_accl0_0 1'b0
`define  a0_r_dx_accl0_1 1'b1
`define  a0_r_u_accl0_0 1'b0
`define  a0_r_u_accl0_1 1'b1
`define  a0_alu_r_u_0 1'b0
`define  a0_alu_r_u_1 1'b1
`define  a0_r_v1_v2_v4_alult_0 1'b0
`define  a0_r_v1_v2_v4_alult_1 1'b1
`define  a0_r_v0_v5_alurt_0 1'b0
`define  a0_r_v0_v5_alurt_1 1'b1
`define  r_dx_0 1'b0
`define  r_dx_1 1'b1
`define  r_u_0 1'b0
`define  r_u_1 1'b1
`define  r_v1_v2_v4_0 1'b0
`define  r_v1_v2_v4_1 1'b1
`define  r_v0_v5_0 1'b0
`define  r_v0_v5_1 1'b1
`define  r_v3_0 1'b0
`define  r_v3_1 1'b1
`define  a1_accl0_r_x_0 1'b0
`define  a1_accl0_r_x_1 1'b1
`define  a1_accl0_r_a_0 1'b0
`define  a1_accl0_r_a_1 1'b1
`define  a1_accl0_r_y_0 1'b0
`define  a1_accl0_r_y_1 1'b1
`define  a1_alu_accl0_0 1'b0
`define  a1_alu_accl0_1 1'b1
`define  a1_r_3_alult_0 1'b0
`define  a1_r_3_alult_1 1'b1
`define  a1_r_x_alurt_0 1'b0
`define  a1_r_x_alurt_1 1'b1
`define  a1_r_y_alurt_0 1'b0
`define  a1_r_y_alurt_1 1'b1
`define  a1_alu_r_x_0 1'b0
`define  a1_alu_r_x_1 1'b1
`define  a1_r_x_alult_0 1'b0
`define  a1_r_x_alult_1 1'b1
`define  a1_accl0_alurt_0 1'b0
`define  a1_accl0_alurt_1 1'b1
`define  a1_accl0_r_dx_v6_0 1'b0
`define  a1_accl0_r_dx_v6_1 1'b1
`define  a1_alu_r_dx_v6_0 1'b0
`define  a1_alu_r_dx_v6_1 1'b1
`define  a1_accl0_alult_0 1'b0
`define  a1_accl0_alult_1 1'b1
`define  a1_r_dx_v6_alurt_0 1'b0
`define  a1_r_dx_v6_alurt_1 1'b1
`define  a1_alu_r_y_0 1'b0
`define  a1_alu_r_y_1 1'b1
`define  a1_r_y_alult_0 1'b0
`define  a1_r_y_alult_1 1'b1
`define  a1_r_a_alurt_0 1'b0
`define  a1_r_a_alurt_1 1'b1
`define  a1_r_x_accl0_0 1'b0
`define  a1_r_x_accl0_1 1'b1
`define  a1_r_y_accl0_0 1'b0
`define  a1_r_y_accl0_1 1'b1
`define  r_x_0 1'b0
`define  r_x_1 1'b1
`define  r_a_0 1'b0
`define  r_a_1 1'b1
`define  r_y_0 1'b0
`define  r_y_1 1'b1
```

```
'define  r_dx_v6_0 1'b0
'define  r_dx_v6_1 1'b1

'define  a0_accl0_bus0_0 1'b0
'define  a0_accl0_bus0_1 1'b1
'define  a0_accl0_bus1_0 1'b0
'define  a0_accl0_bus1_1 1'b1
'define  a1_accl0_bus0_0 1'b0
'define  a1_accl0_bus0_1 1'b1
'define  a1_bus1_accl0_0 1'b0
'define  a1_bus1_accl0_1 1'b1
'define  port0_bus0_0 1'b0
'define  port0_bus0_1 1'b1
'define  port2_bus0_0 1'b0
'define  port2_bus0_1 1'b1
'define  port3_bus0_0 1'b0
'define  port3_bus0_1 1'b1
'define  bus0_port3_0 1'b0
'define  bus0_port3_1 1'b1
'define  port4_bus0_0 1'b0
'define  port4_bus0_1 1'b1
'define  bus0_port4_0 1'b0
'define  bus0_port4_1 1'b1
'define  port1_bus1_0 1'b0
'define  port1_bus1_1 1'b1
'define  port3_bus1_0 1'b0
'define  port3_bus1_1 1'b1
'define  bus1_port3_0 1'b0
'define  bus1_port3_1 1'b1

/* Defining Symbolic constants to the constants that are used in the code */
'define   CONST0     0

/* ---- A-Block 0 ------ */

module ablock0 (r_dx_datain_accl0_r_v3_datain, switch_ctrl, write_en, alu_ctrl, clk);
inout   [7:0] r_dx_datain_accl0_r_v3_datain;
input   [13:0] switch_ctrl; // set of control signals to all switches in the A-Block
input   [4:0] write_en; // enables a register to write into
input   [0:0] alu_ctrl; // controls the operation in the ALU
input           clk;

/* Registers in the A-Block */
reg     [7:0] r_dx;
reg     [7:0] r_u;
reg     [7:0] r_v1_v2_v4;
reg     [7:0] r_v0_v5;
reg     [7:0] r_v3;

wire    [7:0] alu_left_in;
wire    [7:0] alu_right_in;

/* In and Out wires connected to each Register */
wire    [7:0] r_dx_datain_accl0_r_v3_datain;
wire    [7:0] r_dx_dataout;
wire    [7:0] r_u_datain;
wire    [7:0] r_u_dataout;
wire    [7:0] r_v1_v2_v4_datain;
wire    [7:0] r_v1_v2_v4_dataout;
wire    [7:0] r_v0_v5_datain_alu_out;
wire    [7:0] r_v0_v5_dataout;
wire    [7:0] r_v3_dataout;

/* ALU */
alu0 opr_mul_sub(r_v0_v5_datain_alu_out, alu_left_in, alu_right_in, alu_ctrl);

/* Switches for bus interconnections */
switch r_dx_alult (alu_left_in, r_dx_dataout, switch_ctrl[0]);
```

```
switch r_u_alult (alu_left_in, r_u_dataout, switch_ctrl[1]);
switch r_v1_v2_v4_alult (alu_left_in, r_v1_v2_v4_dataout, switch_ctrl[2]);
switch r_v0_v5_alult (alu_left_in, r_v0_v5_dataout, switch_ctrl[3]);
switch r_dx_alurt (alu_right_in, r_dx_dataout, switch_ctrl[4]);
switch r_v1_v2_v4_alurt (alu_right_in, r_v1_v2_v4_dataout, switch_ctrl[5]);
switch r_v0_v5_alurt (alu_right_in, r_v0_v5_dataout, switch_ctrl[6]);
switch r_v3_alurt (alu_right_in, r_v3_dataout, switch_ctrl[7]);
switch r_dx_accl0 (r_dx_datain_accl0_r_v3_datain, r_dx_dataout, switch_ctrl[8]);
switch r_u_accl0 (r_dx_datain_accl0_r_v3_datain, r_u_dataout, switch_ctrl[9]);
switch alu_r_u (r_u_datain, r_v0_v5_datain_alu_out, switch_ctrl[10]);
switch accl0_r_u (r_u_datain, r_dx_datain_accl0_r_v3_datain, switch_ctrl[11]);
switch alu_r_v1_v2_v4 (r_v1_v2_v4_datain, r_v0_v5_datain_alu_out, switch_ctrl[12]);
switch accl0_r_v1_v2_v4 (r_v1_v2_v4_datain, r_dx_datain_accl0_r_v3_datain, switch_ctrl[13]);

assign r_dx_dataout = r_dx;
assign r_u_dataout = r_u;
assign r_v1_v2_v4_dataout = r_v1_v2_v4;
assign r_v0_v5_dataout = r_v0_v5;
assign r_v3_dataout = r_v3;

/* Depending on the 'write_en' signal data will be written into the register */
always @(posedge write_en[0])
r_dx = r_dx_datain_accl0_r_v3_datain;

always @(posedge write_en[1])
r_u = r_u_datain;

always @(posedge write_en[2])
r_v1_v2_v4 = r_v1_v2_v4_datain;

always @(posedge write_en[3])
r_v0_v5 = r_v0_v5_datain_alu_out;

always @(posedge write_en[4])
r_v3 = r_dx_datain_accl0_r_v3_datain;

endmodule

/* ---- A-Block 1 ------ */

module ablock1 (alu_status, r_a_datain_accl0, switch_ctrl, write_en, alu_ctrl, clk);
output          alu_status;
inout   [7:0] r_a_datain_accl0;
input   [17:0] switch_ctrl; // set of control signals to all switches in the A-Block
input   [3:0] write_en; // enables a register to write into
input   [1:0] alu_ctrl; // controls the operation in the ALU
input           clk;

/* Registers in the A-Block */
reg     [7:0] r_x;
reg     [7:0] r_a;
reg     [7:0] r_y;
reg     [7:0] r_3;
reg     [7:0] r_dx_v6;

wire    [7:0] alu_left_in;
wire    [7:0] alu_right_in;
wire    [7:0] alu_out;

/* In and Out wires connected to each Register */
wire    [7:0] r_x_datain;
wire    [7:0] r_x_dataout;
wire    [7:0] r_a_datain_accl0;
wire    [7:0] r_a_dataout;
wire    [7:0] r_y_datain;
wire    [7:0] r_y_dataout;
wire    [7:0] r_3_dataout;
wire    [7:0] r_dx_v6_datain;
```

```
wire    [7:0] r_dx_v6_dataout;

/* ALU */
alu1 opr_mul_add_lt(alu_out, alu_status, alu_left_in, alu_right_in, alu_ctrl);

/* Switches for bus interconnections */
switch r_x_alult (alu_left_in, r_x_dataout, switch_ctrl[0]);
switch r_y_alult (alu_left_in, r_y_dataout, switch_ctrl[1]);
switch r_3_alult (alu_left_in, r_3_dataout, switch_ctrl[2]);
switch accl0_alult (alu_left_in, r_a_datain_accl0, switch_ctrl[3]);
switch r_x_alurt (alu_right_in, r_x_dataout, switch_ctrl[4]);
switch r_a_alurt (alu_right_in, r_a_dataout, switch_ctrl[5]);
switch r_y_alurt (alu_right_in, r_y_dataout, switch_ctrl[6]);
switch r_dx_v6_alurt (alu_right_in, r_dx_v6_dataout, switch_ctrl[7]);
switch accl0_alurt (alu_right_in, r_a_datain_accl0, switch_ctrl[8]);
switch r_x_accl0 (r_a_datain_accl0, r_x_dataout, switch_ctrl[9]);
switch r_y_accl0 (r_a_datain_accl0, r_y_dataout, switch_ctrl[10]);
switch alu_r_x (r_x_datain, alu_out, switch_ctrl[11]);
switch accl0_r_x (r_x_datain, r_a_datain_accl0, switch_ctrl[12]);
switch alu_r_y (r_y_datain, alu_out, switch_ctrl[13]);
switch accl0_r_y (r_y_datain, r_a_datain_accl0, switch_ctrl[14]);
switch alu_r_dx_v6 (r_dx_v6_datain, alu_out, switch_ctrl[15]);
switch accl0_r_dx_v6 (r_dx_v6_datain, r_a_datain_accl0, switch_ctrl[16]);
switch alu_accl0 (r_a_datain_accl0, alu_out, switch_ctrl[17]);

assign r_x_dataout = r_x;
assign r_a_dataout = r_a;
assign r_y_dataout = r_y;
assign r_3_dataout = r_3;
assign r_dx_v6_dataout = r_dx_v6;

/* Depending on the ’write_en’ signal data will be written into the register */
always @(posedge write_en[0])
r_x = r_x_datain;

always @(posedge write_en[1])
r_a = r_a_datain_accl0;

always @(posedge write_en[2])
r_y = r_y_datain;

always @(posedge write_en[3])
r_dx_v6 = r_dx_v6_datain;

endmodule

/* ALU for A-Block 0 */
module alu0 (alu_out, left_in, right_in, ctrl);
output  [7:0] alu_out;
input   [7:0] left_in;
input   [7:0] right_in;
input   [0:0] ctrl;

wire    [7:0] mul_alu_out;
wire    [7:0] sub_alu_out;

alu_mul_1_1 mul (mul_alu_out, left_in, right_in);
alu_sub_1_1 sub (sub_alu_out, left_in, right_in);

/* alu output or alu status is controlled by ’ctrl’ signal */
assign alu_out = ((ctrl == ‘A0_ALUMUL) ? mul_alu_out : ((ctrl == ‘A0_ALUSUB) ? sub_alu_out : alu_out));

endmodule

/* ALU for A-Block 1 */
module alu1 (alu_out, alu_status, left_in, right_in, ctrl);
output  [7:0] alu_out;
output          alu_status;
```

```
input   [7:0] left_in;
input   [7:0] right_in;
input   [1:0] ctrl;

wire    [7:0] mul_alu_out;
wire    [7:0] add_alu_out;
wire          lt_alu_status;

alu_mul_1_1 mul (mul_alu_out, left_in, right_in);
alu_add_1_1 add (add_alu_out, left_in, right_in);
alu_lt_1_1 lt (lt_alu_status, left_in, right_in);

/* alu output or alu status is controlled by 'ctrl' signal */
assign alu_out = ((ctrl == `A1_ALUMUL) ? mul_alu_out : ((ctrl == `A1_ALUADD) ? add_alu_out : alu_out));
assign alu_status = ((ctrl == `A1_ALULT) ? lt_alu_status : alu_status);

endmodule

/* ---- Controller ---- */

module controller (bus_port_ctrl, bus_accl_ctrl, a0_switch_ctrl, a0_write_en, a0_alu_ctrl, a1_switch_ctrl,
                   a1_write_en, a1_alu_ctrl, a1_alu_status, clk);
output  [5:0] bus_port_ctrl;
output  [6:0] bus_accl_ctrl;
output  [13:0] a0_switch_ctrl;
output  [4:0] a0_write_en;
output  [0:0] a0_alu_ctrl;
output  [17:0] a1_switch_ctrl;
output  [3:0] a1_write_en;
output  [1:0] a1_alu_ctrl;
input         a1_alu_status;
input         clk;

wire    [3:0] state;
wire    [3:0] statePi;

generateControlsignals genctrls (bus_port_ctrl, bus_accl_ctrl, a0_switch_ctrl, a0_alu_ctrl, a1_switch_ctrl,
  a1_alu_ctrl, state);
generateWriteEnable genwten (a0_write_en, a1_write_en, statePi, clk);
sequenceGenerator seq (state, a1_alu_status, clk);

/* phase shifting state by 180 degrees
 statePi contains the phase shifted state
 statePi is used to generate 'write_en' signal to each A-Block */
d_ff df0 (statePi[0], state[0], ~clk);
d_ff df1 (statePi[1], state[1], ~clk);
d_ff df2 (statePi[2], state[2], ~clk);
d_ff df3 (statePi[3], state[3], ~clk);

endmodule

/* Generates the next control step */
module sequenceGenerator (state, a1_alu_status, clk);
output  [3:0] state;
input         a1_alu_status;
input         clk;
reg     [3:0] state;

always @(posedge clk)
begin
case (state)
`CS0 : state = `CS1;
`CS1 : state = `CS2;
`CS2 : state = `CS3;
`CS3 : state = `CS4;
`CS4 : state = `CS5;
`CS5 : state = `CS6;
`CS6 : state = `CS7;
```

```
`CS7 : state = `CS8;
`CS8 : state = ((a1_alu_status == 0) ? `CS9 : `CS3);
`CS9 : state = `CS10;
`CS10 : state = `CS0;
`RESET : state = `CS0;
endcase
end
endmodule

/* Generates the control signals for all A-Blocks */
module generateControlsignals (bus_port_ctrl, bus_accl_ctrl, a0_switch_ctrl, a0_alu_ctrl, a1_switch_ctrl,
                               a1_alu_ctrl, state);
output  [5:0] bus_port_ctrl;
output  [6:0] bus_accl_ctrl;
output  [13:0] a0_switch_ctrl;
output  [0:0] a0_alu_ctrl;
output  [17:0] a1_switch_ctrl;
output  [1:0] a1_alu_ctrl;
input   [3:0] state;

reg     [5:0] bus_port_ctrl;
reg     [6:0] bus_accl_ctrl;
reg     [13:0] a0_switch_ctrl;
reg     [0:0] a0_alu_ctrl;
reg     [17:0] a1_switch_ctrl;
reg     [1:0] a1_alu_ctrl;

always @(state)
begin
case (state)
`CS0 : begin
bus_port_ctrl = {`bus1_port3_0, `port1_bus1_1, `bus0_port4_0, `bus0_port3_0, `port2_bus0_0, `port0_bus0_1};
bus_accl_ctrl = {`a1_bus1_accl0_1, `a1_accl0_bus0_0, `a1_accl0_bus0_0, `a0_accl0_bus1_0, `a0_accl0_bus1_0,
                              `a0_accl0_bus0_1, `a0_accl0_bus0_0};
a0_switch_ctrl = {`a0_accl0_r_v1_v2_v4_0, `a0_alu_r_v1_v2_v4_0, `a0_accl0_r_u_0, `a0_alu_r_u_0,
                               `a0_r_u_accl0_0, `a0_r_dx_accl0_0, `a0_r_v3_alurt_0, `a0_r_v0_v5_alurt_0,
                               `a0_r_v1_v2_v4_alurt_0, `a0_r_dx_alurt_0, `a0_r_v0_v5_alult_0,
                               `a0_r_v1_v2_v4_alult_0, `a0_r_u_alult_0, `a0_r_dx_alult_0};
a1_switch_ctrl = {`a1_alu_accl0_0, `a1_accl0_r_dx_v6_0, `a1_alu_r_dx_v6_0, `a1_accl0_r_y_0,
                               `a1_alu_r_y_0, `a1_accl0_r_x_1, `a1_alu_r_x_0, `a1_r_y_accl0_0, `a1_r_x_accl0_0,
                               `a1_accl0_alurt_0, `a1_r_dx_v6_alurt_0, `a1_r_y_alurt_0, `a1_r_a_alurt_0,
                               `a1_r_x_alurt_0, `a1_accl0_alult_0, `a1_r_3_alult_0, `a1_r_y_alult_0, `a1_r_x_alult_0};
end

`CS1 : begin
bus_port_ctrl = {`bus1_port3_0, `port1_bus1_1, `bus0_port4_0, `bus0_port3_0, `port2_bus0_1, `port0_bus0_0};
bus_accl_ctrl = {`a1_bus1_accl0_0, `a1_accl0_bus0_1, `a1_accl0_bus0_0, `a0_accl0_bus1_1, `a0_accl0_bus1_0,
                              `a0_accl0_bus0_0, `a0_accl0_bus0_0};
a0_switch_ctrl = {`a0_accl0_r_v1_v2_v4_0, `a0_alu_r_v1_v2_v4_0, `a0_accl0_r_u_1, `a0_alu_r_u_0,
                               `a0_r_u_accl0_0, `a0_r_dx_accl0_0, `a0_r_v3_alurt_0, `a0_r_v0_v5_alurt_0,
                               `a0_r_v1_v2_v4_alurt_0, `a0_r_dx_alurt_0, `a0_r_v0_v5_alult_0, `a0_r_v1_v2_v4_alult_0,
                               `a0_r_u_alult_0, `a0_r_dx_alult_0};
a1_switch_ctrl = {`a1_alu_accl0_0, `a1_accl0_r_dx_v6_0, `a1_alu_r_dx_v6_0, `a1_accl0_r_y_0, `a1_alu_r_y_0,
                               `a1_accl0_r_x_0, `a1_alu_r_x_0, `a1_r_y_accl0_0, `a1_r_x_accl0_0, `a1_accl0_alurt_0,
                               `a1_r_dx_v6_alurt_0, `a1_r_y_alurt_0, `a1_r_a_alurt_0, `a1_r_x_alurt_0,
                               `a1_accl0_alult_0, `a1_r_3_alult_0, `a1_r_y_alult_0, `a1_r_x_alult_0};
end

`CS2 : begin
bus_port_ctrl = {`bus1_port3_0, `port1_bus1_0, `bus0_port4_0, `bus0_port3_0, `port2_bus0_0, `port0_bus0_1};
bus_accl_ctrl = {`a1_bus1_accl0_0, `a1_accl0_bus0_1, `a1_accl0_bus0_0, `a0_accl0_bus1_0, `a0_accl0_bus1_0,
                              `a0_accl0_bus0_0, `a0_accl0_bus0_0};
a0_switch_ctrl = {`a0_accl0_r_v1_v2_v4_0, `a0_alu_r_v1_v2_v4_0, `a0_accl0_r_u_0, `a0_alu_r_u_0,
                               `a0_r_u_accl0_0, `a0_r_dx_accl0_0, `a0_r_v3_alurt_0, `a0_r_v0_v5_alurt_0,
                               `a0_r_v1_v2_v4_alurt_0, `a0_r_dx_alurt_0, `a0_r_v0_v5_alult_0,
                               `a0_r_v1_v2_v4_alult_0, `a0_r_u_alult_0, `a0_r_dx_alult_0};
a1_switch_ctrl = {`a1_alu_accl0_0, `a1_accl0_r_dx_v6_0, `a1_alu_r_dx_v6_0, `a1_accl0_r_y_1, `a1_alu_r_y_0,
                               `a1_accl0_r_x_0, `a1_alu_r_x_0, `a1_r_y_accl0_0, `a1_r_x_accl0_0, `a1_accl0_alurt_0,
```

```
                                `a1_r_dx_v6_alurt_0, `a1_r_y_alurt_0, `a1_r_a_alurt_0, `a1_r_x_alurt_0,
                                `a1_accl0_alult_0, `a1_r_3_alult_0, `a1_r_y_alult_0, `a1_r_x_alult_0};
end

`CS3 : begin
bus_port_ctrl = {`bus1_port3_0, `port1_bus1_0, `bus0_port4_0, `bus0_port3_0, `port2_bus0_0, `port0_bus0_0};
bus_accl_ctrl = {`a1_bus1_accl0_0, `a1_accl0_bus0_1, `a1_accl0_bus0_1, `a0_accl0_bus1_0, `a0_accl0_bus1_0,
                               `a0_accl0_bus0_1, `a0_accl0_bus0_0};
a0_switch_ctrl = {`a0_accl0_r_v1_v2_v4_1, `a0_alu_r_v1_v2_v4_0, `a0_accl0_r_u_0, `a0_alu_r_u_0,
                                `a0_r_u_accl0_0, `a0_r_dx_accl0_0, `a0_r_v3_alurt_0, `a0_r_v0_v5_alurt_0,
                                `a0_r_v1_v2_v4_alurt_0, `a0_r_dx_alurt_1, `a0_r_v0_v5_alult_0,
                                `a0_r_v1_v2_v4_alult_0, `a0_r_u_alult_1, `a0_r_dx_alult_0};
a0_alu_ctrl = `A0_ALUMUL;
a1_switch_ctrl = {`a1_alu_accl0_1, `a1_accl0_r_dx_v6_0, `a1_alu_r_dx_v6_0, `a1_accl0_r_y_0, `a1_alu_r_y_0,
                                `a1_accl0_r_x_0, `a1_alu_r_x_0, `a1_r_y_accl0_0, `a1_r_x_accl0_0, `a1_accl0_alurt_0,
                                `a1_r_dx_v6_alurt_0, `a1_r_y_alurt_0, `a1_r_a_alurt_0, `a1_r_x_alurt_1,
                                `a1_accl0_alult_0, `a1_r_3_alult_1, `a1_r_y_alult_0, `a1_r_x_alult_0};
a1_alu_ctrl = `A1_ALUMUL;
end

`CS4 : begin
bus_port_ctrl = {`bus1_port3_0, `port1_bus1_0, `bus0_port4_0, `bus0_port3_0, `port2_bus0_0, `port0_bus0_0};
bus_accl_ctrl = {`a1_bus1_accl0_0, `a1_accl0_bus0_1, `a1_accl0_bus0_1, `a0_accl0_bus1_0, `a0_accl0_bus1_0,
                               `a0_accl0_bus0_1, `a0_accl0_bus0_0};
a0_switch_ctrl = {`a0_accl0_r_v1_v2_v4_0, `a0_alu_r_v1_v2_v4_1, `a0_accl0_r_u_0, `a0_alu_r_u_0,
                                `a0_r_u_accl0_0, `a0_r_dx_accl0_0, `a0_r_v3_alurt_0, `a0_r_v0_v5_alurt_0,
                                `a0_r_v1_v2_v4_alurt_1, `a0_r_dx_alurt_0, `a0_r_v0_v5_alult_1,
   `a0_r_v1_v2_v4_alult_0, `a0_r_u_alult_0, `a0_r_dx_alult_0};
a0_alu_ctrl = `A0_ALUMUL;
a1_switch_ctrl = {`a1_alu_accl0_1, `a1_accl0_r_dx_v6_0, `a1_alu_r_dx_v6_0, `a1_accl0_r_y_0, `a1_alu_r_y_0,
                                `a1_accl0_r_x_0, `a1_alu_r_x_0, `a1_r_y_accl0_0, `a1_r_x_accl0_0, `a1_accl0_alurt_0,
                                `a1_r_dx_v6_alurt_0, `a1_r_y_alurt_1, `a1_r_a_alurt_0, `a1_r_x_alurt_0,
                                `a1_accl0_alult_0, `a1_r_3_alult_1, `a1_r_y_alult_0, `a1_r_x_alult_0};
a1_alu_ctrl = `A1_ALUMUL;
end

`CS5 : begin
bus_port_ctrl = {`bus1_port3_0, `port1_bus1_0, `bus0_port4_0, `bus0_port3_0, `port2_bus0_0, `port0_bus0_0};
bus_accl_ctrl = {`a1_bus1_accl0_0, `a1_accl0_bus0_1, `a1_accl0_bus0_0, `a0_accl0_bus1_0, `a0_accl0_bus1_0,
                               `a0_accl0_bus0_1, `a0_accl0_bus0_1};
a0_switch_ctrl = {`a0_accl0_r_v1_v2_v4_0, `a0_alu_r_v1_v2_v4_0, `a0_accl0_r_u_0, `a0_alu_r_u_0,
                                `a0_r_u_accl0_0, `a0_r_dx_accl0_1, `a0_r_v3_alurt_1, `a0_r_v0_v5_alurt_0,
                                `a0_r_v1_v2_v4_alurt_0, `a0_r_dx_alurt_0, `a0_r_v0_v5_alult_0, `a0_r_v1_v2_v4_alult_0,
                                `a0_r_u_alult_0, `a0_r_dx_alult_1};
a0_alu_ctrl = `A0_ALUMUL;
a1_switch_ctrl = {`a1_alu_accl0_0, `a1_accl0_r_dx_v6_1, `a1_alu_r_dx_v6_0, `a1_accl0_r_y_0, `a1_alu_r_y_0,
                                `a1_accl0_r_x_0, `a1_alu_r_x_1, `a1_r_y_accl0_0, `a1_r_x_accl0_0, `a1_accl0_alurt_1,
                                `a1_r_dx_v6_alurt_0, `a1_r_y_alurt_0, `a1_r_a_alurt_0, `a1_r_x_alurt_0,
                                `a1_accl0_alult_0, `a1_r_3_alult_0, `a1_r_y_alult_0, `a1_r_x_alult_1};
a1_alu_ctrl = `A1_ALUADD;
end

`CS6 : begin
bus_port_ctrl = {`bus1_port3_0, `port1_bus1_0, `bus0_port4_0, `bus0_port3_0, `port2_bus0_0, `port0_bus0_0};
bus_accl_ctrl = {`a1_bus1_accl0_0, `a1_accl0_bus0_1, `a1_accl0_bus0_0, `a0_accl0_bus1_0, `a0_accl0_bus1_0,
                               `a0_accl0_bus0_1, `a0_accl0_bus0_1};
a0_switch_ctrl = {`a0_accl0_r_v1_v2_v4_0, `a0_alu_r_v1_v2_v4_1, `a0_accl0_r_u_0, `a0_alu_r_u_0,
                                `a0_r_u_accl0_1, `a0_r_dx_accl0_0, `a0_r_v3_alurt_0, `a0_r_v0_v5_alurt_0,
                                `a0_r_v1_v2_v4_alurt_1, `a0_r_dx_alurt_0, `a0_r_v0_v5_alult_0,
                                `a0_r_v1_v2_v4_alult_0, `a0_r_u_alult_1, `a0_r_dx_alult_0};
a0_alu_ctrl = `A0_ALUSUB;
a1_switch_ctrl = {`a1_alu_accl0_0, `a1_accl0_r_dx_v6_0, `a1_alu_r_dx_v6_1, `a1_accl0_r_y_0, `a1_alu_r_y_0,
                                `a1_accl0_r_x_0, `a1_alu_r_x_0, `a1_r_y_accl0_0, `a1_r_x_accl0_0, `a1_accl0_alurt_0,
                                `a1_r_dx_v6_alurt_1, `a1_r_y_alurt_0, `a1_r_a_alurt_0, `a1_r_x_alurt_0,
                                `a1_accl0_alult_1, `a1_r_3_alult_0, `a1_r_y_alult_0, `a1_r_x_alult_0};
a1_alu_ctrl = `A1_ALUMUL;
end
```

```
`CS7 : begin
bus_port_ctrl = {`bus1_port3_0, `port1_bus1_0, `bus0_port4_0, `bus0_port3_0, `port2_bus0_0, `port0_bus0_0};
bus_accl_ctrl = {`a1_bus1_accl0_0, `a1_accl0_bus0_0, `a1_accl0_bus0_0, `a0_accl0_bus1_0, `a0_accl0_bus1_0,
                        `a0_accl0_bus0_0, `a0_accl0_bus0_0};
a0_switch_ctrl = {`a0_accl0_r_v1_v2_v4_0, `a0_alu_r_v1_v2_v4_0, `a0_accl0_r_u_0, `a0_alu_r_u_1,
                         `a0_r_u_accl0_0, `a0_r_dx_accl0_0, `a0_r_v3_alurt_0, `a0_r_v0_v5_alurt_1,
                         `a0_r_v1_v2_v4_alurt_0, `a0_r_dx_alurt_0, `a0_r_v0_v5_alult_0,
                         `a0_r_v1_v2_v4_alult_1, `a0_r_u_alult_0, `a0_r_dx_alult_0};
a0_alu_ctrl = `A0_ALUSUB;
a1_switch_ctrl = {`a1_alu_accl0_0, `a1_accl0_r_dx_v6_0, `a1_alu_r_dx_v6_0, `a1_accl0_r_y_0, `a1_alu_r_y_1,
                         `a1_accl0_r_x_0, `a1_alu_r_x_0, `a1_r_y_accl0_0, `a1_r_x_accl0_0, `a1_accl0_alurt_0,
                         `a1_r_dx_v6_alurt_1, `a1_r_y_alurt_0, `a1_r_a_alurt_0, `a1_r_x_alurt_0,
                         `a1_accl0_alult_0, `a1_r_3_alult_0, `a1_r_y_alult_1, `a1_r_x_alult_0};
a1_alu_ctrl = `A1_ALUADD;
end

`CS8 : begin
bus_port_ctrl = {`bus1_port3_0, `port1_bus1_0, `bus0_port4_0, `bus0_port3_0, `port2_bus0_0, `port0_bus0_0};
bus_accl_ctrl = {`a1_bus1_accl0_0, `a1_accl0_bus0_0, `a1_accl0_bus0_0, `a0_accl0_bus1_0, `a0_accl0_bus1_0,
                        `a0_accl0_bus0_0, `a0_accl0_bus0_0};
a0_switch_ctrl = {`a0_accl0_r_v1_v2_v4_0, `a0_alu_r_v1_v2_v4_0, `a0_accl0_r_u_0, `a0_alu_r_u_0,
                         `a0_r_u_accl0_0, `a0_r_dx_accl0_0, `a0_r_v3_alurt_0, `a0_r_v0_v5_alurt_0,
                         `a0_r_v1_v2_v4_alurt_0, `a0_r_dx_alurt_0, `a0_r_v0_v5_alult_0,
                         `a0_r_v1_v2_v4_alult_0, `a0_r_u_alult_0, `a0_r_dx_alult_0};
a1_switch_ctrl = {`a1_alu_accl0_0, `a1_accl0_r_dx_v6_0, `a1_alu_r_dx_v6_0, `a1_accl0_r_y_0, `a1_alu_r_y_0,
                         `a1_accl0_r_x_0, `a1_alu_r_x_0, `a1_r_y_accl0_0, `a1_r_x_accl0_0, `a1_accl0_alurt_0,
                         `a1_r_dx_v6_alurt_0, `a1_r_y_alurt_0, `a1_r_a_alurt_1, `a1_r_x_alurt_0,
                         `a1_accl0_alult_0, `a1_r_3_alult_0, `a1_r_y_alult_0, `a1_r_x_alult_1};
a1_alu_ctrl = `A1_ALULT;
end

`CS9 : begin
bus_port_ctrl = {`bus1_port3_0, `port1_bus1_0, `bus0_port4_0, `bus0_port3_1, `port2_bus0_0, `port0_bus0_0};
bus_accl_ctrl = {`a1_bus1_accl0_0, `a1_accl0_bus0_1, `a1_accl0_bus0_1, `a0_accl0_bus1_0, `a0_accl0_bus1_0,
                        `a0_accl0_bus0_0, `a0_accl0_bus0_0};
a0_switch_ctrl = {`a0_accl0_r_v1_v2_v4_0, `a0_alu_r_v1_v2_v4_0, `a0_accl0_r_u_0, `a0_alu_r_u_0,
                         `a0_r_u_accl0_0, `a0_r_dx_accl0_0, `a0_r_v3_alurt_0, `a0_r_v0_v5_alurt_0,
                         `a0_r_v1_v2_v4_alurt_0, `a0_r_dx_alurt_0, `a0_r_v0_v5_alult_0,
                         `a0_r_v1_v2_v4_alult_0, `a0_r_u_alult_0, `a0_r_dx_alult_0};
a1_switch_ctrl = {`a1_alu_accl0_0, `a1_accl0_r_dx_v6_0, `a1_alu_r_dx_v6_0, `a1_accl0_r_y_0, `a1_alu_r_y_0,
                         `a1_accl0_r_x_0, `a1_alu_r_x_0, `a1_r_y_accl0_0, `a1_r_x_accl0_1, `a1_accl0_alurt_0,
                         `a1_r_dx_v6_alurt_0, `a1_r_y_alurt_0, `a1_r_a_alurt_0, `a1_r_x_alurt_0,
                         `a1_accl0_alult_0, `a1_r_3_alult_0, `a1_r_y_alult_0, `a1_r_x_alult_0};
end

`CS10 : begin
bus_port_ctrl = {`bus1_port3_1, `port1_bus1_0, `bus0_port4_1, `bus0_port3_0, `port2_bus0_0, `port0_bus0_0};
bus_accl_ctrl = {`a1_bus1_accl0_0, `a1_accl0_bus0_1, `a1_accl0_bus0_1, `a0_accl0_bus1_1, `a0_accl0_bus1_1,
                        `a0_accl0_bus0_0, `a0_accl0_bus0_0};
a0_switch_ctrl = {`a0_accl0_r_v1_v2_v4_0, `a0_alu_r_v1_v2_v4_0, `a0_accl0_r_u_0, `a0_alu_r_u_0,
                         `a0_r_u_accl0_1, `a0_r_dx_accl0_0, `a0_r_v3_alurt_0, `a0_r_v0_v5_alurt_0,
                         `a0_r_v1_v2_v4_alurt_0, `a0_r_dx_alurt_0, `a0_r_v0_v5_alult_0,
                         `a0_r_v1_v2_v4_alult_0, `a0_r_u_alult_0, `a0_r_dx_alult_0};
a1_switch_ctrl = {`a1_alu_accl0_0, `a1_accl0_r_dx_v6_0, `a1_alu_r_dx_v6_0, `a1_accl0_r_y_0, `a1_alu_r_y_0,
                         `a1_accl0_r_x_0, `a1_alu_r_x_0, `a1_r_y_accl0_1, `a1_r_x_accl0_0, `a1_accl0_alurt_0,
                         `a1_r_dx_v6_alurt_0, `a1_r_y_alurt_0, `a1_r_a_alurt_0, `a1_r_x_alurt_0,
                         `a1_accl0_alult_0, `a1_r_3_alult_0, `a1_r_y_alult_0, `a1_r_x_alult_0};
end

default : begin
bus_port_ctrl = `CONST0;
bus_accl_ctrl = `CONST0;
a0_switch_ctrl = `CONST0;
a1_switch_ctrl = `CONST0;
end
endcase
end
endmodule
```

```
/* Generates the write enable signals to all A-Blocks */
module generateWriteEnable (a0_write_en, a1_write_en, statePi, clk);
output  [4:0] a0_write_en;
output  [3:0] a1_write_en;
input   [3:0] statePi;
input          clk;

reg     [4:0] a0_write_en;
reg     [3:0] a1_write_en;

always @(posedge clk)
begin
case (statePi)
`CS0 : begin
a0_write_en = {`r_v3_0, `r_v0_v5_0, `r_v1_v2_v4_0, `r_u_0, `r_dx_1};
a1_write_en = {`r_dx_v6_0, `r_y_0, `r_a_0, `r_x_1};
end

`CS1 : begin
a0_write_en = {`r_v3_0, `r_v0_v5_0, `r_v1_v2_v4_0, `r_u_1, `r_dx_0};
a1_write_en = {`r_dx_v6_0, `r_y_0, `r_a_1, `r_x_0};
end

`CS2 : begin
a0_write_en = {`r_v3_0, `r_v0_v5_0, `r_v1_v2_v4_0, `r_u_0, `r_dx_0};
a1_write_en = {`r_dx_v6_0, `r_y_1, `r_a_0, `r_x_0};
end

`CS3 : begin
a0_write_en = {`r_v3_0, `r_v0_v5_1, `r_v1_v2_v4_1, `r_u_0, `r_dx_0};
a1_write_en = {`r_dx_v6_0, `r_y_0, `r_a_0, `r_x_0};
end

`CS4 : begin
a0_write_en = {`r_v3_1, `r_v0_v5_0, `r_v1_v2_v4_1, `r_u_0, `r_dx_0};
a1_write_en = {`r_dx_v6_0, `r_y_0, `r_a_0, `r_x_0};
end

`CS5 : begin
a0_write_en = {`r_v3_0, `r_v0_v5_1, `r_v1_v2_v4_0, `r_u_0, `r_dx_0};
a1_write_en = {`r_dx_v6_1, `r_y_0, `r_a_0, `r_x_1};
end

`CS6 : begin
a0_write_en = {`r_v3_0, `r_v0_v5_0, `r_v1_v2_v4_1, `r_u_0, `r_dx_0};
a1_write_en = {`r_dx_v6_1, `r_y_0, `r_a_0, `r_x_0};
end

`CS7 : begin
a0_write_en = {`r_v3_0, `r_v0_v5_0, `r_v1_v2_v4_0, `r_u_1, `r_dx_0};
a1_write_en = {`r_dx_v6_0, `r_y_1, `r_a_0, `r_x_0};
end

`CS8 : begin
a0_write_en = {`r_v3_0, `r_v0_v5_0, `r_v1_v2_v4_0, `r_u_0, `r_dx_0};
a1_write_en = {`r_dx_v6_0, `r_y_0, `r_a_0, `r_x_0};
end

`CS9 : begin
a0_write_en = {`r_v3_0, `r_v0_v5_0, `r_v1_v2_v4_0, `r_u_0, `r_dx_0};
a1_write_en = {`r_dx_v6_0, `r_y_0, `r_a_0, `r_x_0};
end

`CS10 : begin
a0_write_en = {`r_v3_0, `r_v0_v5_0, `r_v1_v2_v4_0, `r_u_0, `r_dx_0};
a1_write_en = {`r_dx_v6_0, `r_y_0, `r_a_0, `r_x_0};
end
```

```
default : begin
a0_write_en = `CONST0;
a1_write_en = `CONST0;
end
endcase
end
always @(negedge clk)
begin
a0_write_en = `CONST0;
a1_write_en = `CONST0;
end
endmodule

/* stimulas module, which instanciates A-Blocks, Controller, Ports and interconnections among those */
module system ( port0, port1, port2, port3, port4, clk );
input   [7:0] port0;
input   [7:0] port1;
input   [7:0] port2;
output   [7:0] port3;
output   [7:0] port4;
input          clk;

wire    [7:0] bus0;
wire    [7:0] bus1;

wire    [7:0] a0_accl0;
wire    [7:0] a1_accl0;

wire          a1_alu_status;

wire    [13:0] a0_switch_ctrl;
wire    [17:0] a1_switch_ctrl;

wire    [4:0] a0_write_en;
wire    [3:0] a1_write_en;

wire    [0:0] a0_alu_ctrl;
wire    [1:0] a1_alu_ctrl;

wire    [5:0] bus_port_ctrl;
wire    [6:0] bus_accl_ctrl;

/* 'switch2way' is the bidirectional switch */
switch port0_bus0 (bus0, port0, bus_port_ctrl[0]);
switch port2_bus0 (bus0, port2, bus_port_ctrl[1]);
switch bus0_port3 (port3, bus0, bus_port_ctrl[2]);
switch bus0_port4 (port4, bus0, bus_port_ctrl[3]);
switch port1_bus1 (bus1, port1, bus_port_ctrl[4]);
switch bus1_port3 (port3, bus1, bus_port_ctrl[5]);

switch2way a0_accl0_bus0 (a0_accl0, bus0, bus_accl_ctrl[1:0]);
switch2way a0_accl0_bus1 (a0_accl0, bus1, bus_accl_ctrl[3:2]);
switch2way a1_accl0_bus0 (a1_accl0, bus0, bus_accl_ctrl[5:4]);
switch a1_bus1_accl0 (a1_accl0, bus1, bus_accl_ctrl[6]);

ablock0 ablk0 (a0_accl0, a0_switch_ctrl, a0_write_en, a0_alu_ctrl, clk);
ablock1 ablk1 (a1_alu_status, a1_accl0, a1_switch_ctrl, a1_write_en, a1_alu_ctrl, clk);
controller cntllr (bus_port_ctrl, bus_accl_ctrl, a0_switch_ctrl, a0_write_en, a0_alu_ctrl,
                  a1_switch_ctrl, a1_write_en, a1_alu_ctrl, a1_alu_status, clk);

endmodule
```

Appendix B

Publications out of this work

1. **C. Karfa**, J. S. Reddy, S. Biswas, C. R. Mandal, D. Sarkar, SAST: An Interconnection aware high level synthesis tool, In 9th VLSI Design and Test (VDAT'05), Page 285-293. August 10-13, Bangalore, India.

2. **C. Karfa**, C. Mandal, D. Sarkar, S R Pentakota, C. Reade. Verification of Scheduling in High-level Synthesis. In IEEE Computer Society Annual Symposium on VLSI (ISVLSI'06). Page 141-146, March 2-3, 2006. Karlsruhe, Germany.

3. **C. Karfa**, C. Mandal, D. Sarkar, S R Pentakota, C. Reade. A Formal Verification Method of Scheduling in High-level Synthesis. In 7th IEEE International Symposium on Quality Electronic Design (ISQED'06), page 71-78, San Jose, CA, USA.

4. **C. Karfa**, D. Sarkar, C Mandal and C. Reade. Register Sharing Verification During Data-path Synthesis. In IEEE International Conference on Computing: Theory and Applications (ICCTA'07), pase 135-140, March 5-7, 2007, Kolkata, India.

5. **C. Karfa**, D. Sarkar, C Mandal and C. Reade. Hand-in-hand verification of high-level synthesis. In 17th ACM Great Lakes Symposium on VLSI 2007 (GLSVLSI'07), page 429-434, March 11-13, 2007, Stresa - Lago Maggiore, Italy.

6. **C. Karfa**, D. Sarkar, C Mandal. An Equivalence Checking Method for Scheduling Verification in High-level Synthesis. In IEEE Transaction on Computer Aided Design on ICs (accepted).

Appendix C

Bio-data

Chandan Karfa was born in Shyamsundar, Burdwan, West Bengal on 9th of May, 1982. He received the B.Tech. degree in Information Technology from University Science Instrumentation Centre of University of Kalyani, Kalyani, West Bengal in 2004. He is currently pursuing his M.S. degree in Computer Science and Engineering from Indian Institute of Technology, Kharagpur, India. He is also working as Junior Project Assistance (JPA) in "High-level Synthesis and verification of Digital Circuits" project, sponsored by Ministry of Human Resource Development (MHRD) under Sponsored Research and Industrial Consultancy, IIT Kharagpur still July, 2004. His current research interests include design automation, verification and optimization of digital circuits. He has published six research papers in different reputed IEEE/ACM international conferences.

Bibliography

[1] D. D. Gajski, N. D. Dutt, A. C. Wu, and S. Y. Lin, *High-Level Synthesis: Introduction to Chip and System Design*. Kluwer Academic Publishers, 1992.

[2] M. Shadad, “An overview of VHDL language and technology,” *Procs. of the 23rd Design Automation Conference*, 1986.

[3] D. E. Thomas and P. Moorby, *The Verilog Hardware Description Language*. Kluwer Academic Publishers, 1991.

[4] A. C. Parker, J. T. Pizarro, and M. Mlinar, “Maha: A program for data path synthesis,” *Procs. of the 23rd Design Automation Conference*, 1986.

[5] G. D. Micheli and D. C. Ku, “Hercules: A system for high level synthesis,” in *Procs. of the 25th ACM/IEEE DAC*, pp. 483–488, 1988.

[6] P. G. Paulin and J. P. Knight, “Force-directed scheduling for the behavioural synthesis of asics,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 8, pp. 661–679, June 1989.

[7] F. Brewer and D. D. Gajski, “Chippe: A system for constraint driven behavioural synthesis,” *IEEE Trans. on CAD.*, pp. 681–695, July 1990.

[8] F.-S. Tsai and Y.-C. Hsu, “STAR - An automatic data path allocator,” *IEEE Trans. on CAD.*, vol. 11, pp. 1053–1064, Sep. 1992.

[9] R. J. Cloutier and D. E. Thomas, “The combination of scheduling, allocation and mapping in a single algorithm,” in *Procs. of the 27th ACM/IEEE DAC*, pp. 71–76, June 1990.

[10] C. Mandal, P. P. Chakrabarti, and S. Ghose, “Gabind: a ga approach to allocation and binding for the high-level synthesis of data paths,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 8, no. 6, pp. 747–750, 2000.

[11] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau, “Spark: a high-level synthesis framework for applying parallelizing compiler transformations,” in *Proceedings of 16th International Conference on VLSI Design*, pp. 461–466, Jan 2003.

[12] D. Gajski and L. Ramachandran, “Introduction to high-level synthesis,” *IEEE transactions on Design and Test of Computers*, pp. 44–54, 1994.

[13] R. Kumar, C. Blumenhr, and D. Schmid, “Formal synthesis in circuit design - a classification and survey,” in *Formal methods in computer-aided design. FMCAD ’96. (LNCS)*, pp. 1166 – 1182, 1996.

[14] H. Eveking, H. Hinrichsen, and G. Ritter, “Automatic verification of scheduling results in high-level synthesis.,” in *Proc. Conf. Design, Automation and Test in Europe 1999*, pp. 59–64, March 1999.

[15] R. Chapman, G. Brown, and M. Leeser, “Verified high-level synthesis in BEDROC,” in *Proc DAC, 1992.*, pp. 59–63, 1992.

[16] N. Narashima, E. Teica, S. Radhakrishnan, S. Govindarajan, and R. Vemuri, “Theorem proving guided development of formal assertions in a resource-constrained scheduler for high-level synthesis,” *Formal Methods in System Design*, vol. Vol 19 No 3, pp. 237–273, 2001.

[17] C. Blumenrohr, D. Eisenbiegler, and R. Kumar, “Applicability of formal synthesis illustrated via scheduling,” in *Proceedings of IWLAS*, 1996.

[18] J. M. Mendias, R. Hermida, M. C. Molina, and O. Penalba, “Efficient verification of scheduling, allocation and binding in high-level synthesis,” *Proc. Digital system Design*, pp. 308–315, 2002.

[19] R. Ernst and J. Bhasker, “Simulation-based verification for high-level synthesis,” *IEEE Des. Test*, vol. 8, no. 1, pp. 14–20, 1991.

[20] C.-J. Tseng, R.-S. Wei, S. G. Rothweiler, M. M. Tong, and A. K. Bose, “Bridge: a versatile behavioral synthesis system,” in *DAC ’88: Proceedings of the 25th ACM/IEEE conference on*

Design automation, (Los Alamitos, CA, USA), pp. 415–420, IEEE Computer Society Press, 1988.

[21] R. A. Bergamaschi and S. Raje, “Observable time windows: Verifying high-level synthesis results,” *IEEE Des. Test*, vol. 14, no. 2, pp. 40–50, 1997.

[22] I. Ghosh, K. Sekar, and V. Boppana, “Design for verification at the register transfer level,” in *ASP-DAC '02: Proceedings of the 2002 conference on Asia South Pacific design automation/VLSI Design*, (Washington, DC, USA), pp. 420–425, IEEE Computer Society, 2002.

[23] R. A. Bergamaschi, R. A. O’Connor, L. Stok, M. Z. Moricz, S. Prakash, A. Kuehlmann, and D. S. Rao, “High-level synthesis in an industrial environment,” *IBM J. Res. Dev.*, vol. 39, no. 1-2, pp. 131–148, 1995.

[24] D. Shepherd and G. Wilson., “Making chips that work,” *New Scientist*, pp. 61–64, 1989.

[25] R. Radhakrishnan, E. Teica, and R. Vermuri, “An approach to high-level synthesis system validation using formally verified transformations,” in *HLDVT '00: Proceedings of the IEEE International High-Level Validation and Test Workshop (HLDVT'00)*, (Washington, DC, USA), p. 80, IEEE Computer Society, 2000.

[26] S. P. Rajan, “Correctness of transformations in high level synthesis,” in *CHDL '95: 12th Conference on Computer Hardware Description Languages and their Applications*, (Chiba, Japan), pp. 597–603, 1995.

[27] T. Krol, J. van Meerbergen, C. Niessen, W. Smits, and J. Huisken, “The sprite input language-an intermediate format for high levelsynthesis,” in *Proceedings. [3rd] European Conference on Design Automation*, pp. 186–192, 1992.

[28] M. Fujita, “Equivalence checking between behavioral and rtl descriptions with virtual controllers and datapaths,” *ACM Trans. Des. Autom. Electron. Syst.*, vol. 10, no. 4, pp. 610–626, 2005.

[29] N. Mansouri and R. Vemuri, “A methodology for automated verification of synthesized rtl designs and its integration with a high-level synthesis tool,” in *Proceedings of FMCAD '98*, (London, UK), pp. 204–221, Springer-Verlag, 1998.

[30] J. Roy, N. Kumar, R. Dutta, and R. Vemuri, "Dss: a distributed high-level synthesis system," *IEEE Design and Test of Computers*, vol. 9, no. 2, pp. 18–32, 1992.

[31] D. Anderson and J. Ainscough, "The verification of scheduling algorithms," in *IEE Colloquium on Structured Methods for Hardware Systems Design*, pp. 7/1–7/5, 1994.

[32] M.J.C. Gordon, "Mechanizing programming logics in higher-order logic," in *Current Trends in Hardware Verification and Automatic Theorem Proving (Proceedings of the Workshop on Hardware Verification)* (G.M. Birtwistle and P.A. Subrahmanyam, eds.), (Banff, Canada), pp. 387–439, Springer-Verlag, Berlin, 1988.

[33] N. Narasimhan, E. Teica, R. Radhakrishnan, S. Govindarajan, and R. Vemuri, "Theorem proving guided development of formal assertions in a resource-constrained scheduler for high-level synthesis," in *Proceedings. of International Conference on Computer Design*, pp. 392–399, 1998.

[34] Y. Kim, S. Kopuri, and N. Mansouri, "Automated formal verification of scheduling process using finite state machine with datapath (FSMD)," in *5th International Symposium on Quality Electronic Design (ISQED'04)*, (Carlifornia), pp. 110–115, March 2004.

[35] J. Lee, Y. Hsu, and Y. Lin, "A new integer linear programming formulation of the scheduling problem in data path synthesis," in *Procs. of the International Conference on Computer-Aided Design*, pp. 20–23, 1988.

[36] R. Jain, A. Majumdar, A. Sharma, and H. Wang, "Empirical evalution of some high-level synthesis scheduling heuristics," in *Procs. of 28th DAC*, pp. 210–215, 1991.

[37] R. Camposano, "Path-based scheduling for synthesis," *IEEE transactions on computer-Aided Design of Integrated Circuits and Systems*, vol. Vol 10 No 1, pp. 85–93, Jan. 1991.

[38] M. Rahmouni and A. A. Jerraya, "Formulation and evaluation of scheduling techniques for control flow graphs," in *Proceedings of EuroDAC'95*, (Brighton), pp. 386–391, 18-22 September 1995.

[39] G. Lakshminarayana, A. Raghunathan, and N. Jha, "Incorporating speculative execution into scheduling of control-flow-intensive design," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 19, pp. 308–324, March 2000.

[40] M. Rim, Y. Fann, and R. Jain, "Global scheduling with code motions for high-level synthesis applications," *IEEE Transactions on VLSI Systems*, vol. 3, pp. 379–392, Sept. 1995.

[41] L. C. V. d. Santosh and J. Jress, "A reordering technique for efficient code motion," in *Procs. of the 36th ACM/IEEE Design Automation Conference*, pp. 296–299, 1999.

[42] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau, "Dynamically increasing the scope of code motions during the high-level synthesis of digital circuits," *IEE Proceedings: Computer and Digital Technique*, vol. 150, pp. 330–337, September 2003.

[43] D. Borrione, J. Dushina, and L. Pierre, "A compositional model for the functional verification of high-level synthesis results," *IEEE Transactions on VLSI Systems*, vol. 8, pp. 526–530, October 2000.

[44] N. Mansouri and R. Vemuri, "Accounting for various register allocation schemes during post-synthesis verification of rtl designs," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE'99)*, pp. 223–230, March 1999.

[45] P. Ashar, S. Bhattacharya, A. Raghunathan, and A. Mukaiyama, "Verification of rtl generated from scheduled behavior in a high-level synthesis flow," in *Proceedings of the 1998 IEEE/ACM international conference on computer-aided design*, (New York, NY, USA), pp. 517–524, ACM Press, 1998.

[46] J. Dushina, D. Borrione, and A. A. Jerraya, "Formal verification of the allocation step in high level synthesis," in *Forum on Design Languages (FDL'98)*, (Lausanne, Switzerland), pp. 1–10, 1998.

[47] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau, "Using global code motions to improve the quality of results for high-level synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 23, pp. 302–312, Feb 2004.

[48] D. Sarkar and S. De Sarkar, "Some inference rules for integer arithmetic for verification of flowchart programs on integers," *IEEE Trans Software. Engg.*, vol. 15, no. 1, pp. 1–9, 1989.

[49] S. Gupta, R. Gupta, N. Dutt, and A. Nicolau, "Coordinated parallelizing compiler optimizations and high-level synthesis," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 9, pp. 1–31, October 2004.

[50] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*. MIT Press and McGraw-Hill, 2001.

[51] D. Gries, *The Science of Programming*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1987.

[52] Z. Manna, *Mathematical Theory of Computation*. Tokyo: McGraw-Hill Kogakusha, 1974.

[53] R. W. Floyd, "Assigning meaning to programs," in *Proceedings the 19th Symposium on Applied Mathematics* (J. T. Schwartz, ed.), (Providence, R.I.), pp. 19–32, American Mathematical Society, 1967. Mathematical Aspects of Computer Science.

[54] C. A. R. Hoare, "An axiomatic basis of computer programming," *Communications of the ACM*, pp. 576–580, 1969.

[55] J. C. King, "Program correctness: On inductive assertion methods," *IEEE Trans. on Software Engineering*, vol. SE-6, no. 5, pp. 465–479, 1980.

[56] W. E. Howden, *Functional program testing and analysis*. New York: McGraw-Hill, 1987.

[57] D. Sarkar and S. De Sarkar, "A set of inference rules for quantified formula handling and array handling in verification of programs over integers," *IEEE Trans Software. Engg.*, vol. 15, no. 11, pp. 1368–1381, 1989.

[58] R. Jain, A. Majumdar, A. Sharma, and H. Wang, "Empirical evalution of some high-level synthesis scheduling heuristics," in *Procs. of 28th DAC*, pp. 210–215, 1991.

[59] S. Devadas and A. R. Newton, "Algorithms for hardware allocation in data path synthesis," *IEEE Trans. on CAD.*, vol. 8, July 1989.

[60] L. J. Hafer and A. C. Parker, "A formal method for the specification, analysis and design of register-transfer level digital logic," *IEEE Trans. on CAD.*, vol. vol. CAD-2, pp. 4–18, Jan. 1983.

[61] P. G. Paulin and J. P. Knight, "Force–directed scheduling in automatic data path synthesis," *Procs. of the 24th Design Automation Conference*, 1987.

[62] G. Lakshminarayana, K. Khouri, and N. Jha, "Wavesched: A novel scheduling technique for control-flow intensive behavioural descriptions," in *Proceedings of International Conference on Computer-Aided Design*, pp. 244–250, Nov 1997.

[63] T.-F. Lee, A.-H. Wu, Y.-L. Lin, and D. Gajski, "A transformation-based method for loop folding," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 13, pp. 439–450, April 1994.

[64] S. Gupta, N. Savoiu, N. Dutt, R. Gupta, and A. Nicolau, "Conditional speculation and its effects on performance and area for high-level synthesis," in *International Symposium on System Synthesis*, pp. 171–176, 2001.

[65] S. Gupta, N. Savoiu, S. Kim, N. Dutt, R. Gupta, and A. Nicolau, "Speculation techniques for high level synthesis of control intensive designs," in *Proceedings of DAC'01*, pp. 269–272, 2001.

[66] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau, "Dynamic conditional branch balancing during the high-level synthesis of control-intensive designs," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE'03)*, pp. 270–275, 2003.

[67] S. Gupta, M. Reshadi, N. savoiu, N. Dutt, R. Gupta, and A. Nicolau, "Dynamic common sub-expression elimination during scheduling in high-level synthesis," in *Proceedings of 15th International Symposium on System Synthesis (ISS'02)*, pp. 261–266, Oct 2002.

[68] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau, "Loop shifting and compaction for the high-level synthesis of designs with complex control flow," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE'04)*, vol. 1, pp. 114–119, Feb 2004.

[69] O. Penalba, J. Mendias, and R. Hermida, "Source code transformation to improve conditional hardware reuse," in *Procs. of the Euromicro Symposium on Digital System Design*, pp. 324–330, September 2002.

[70] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. The MIT Press, 2002.

[71] A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri, "NuSMV: A new symbolic model checker," *International Journal on Software Tools for Technology Transfer*, vol. 2, no. 4, pp. 410–425, 2000.

[72] N.-S. Woo, "A global, dynamic register allocation and binding for data path synthesis system," in *Procs. of 27th DAC*, pp. 505–510, 1990.

[73] F. Kurdhai and A. Parker, "Real: A program for register alocation," in *Procs. of 24th DAC*, pp. 210–215, 1987.

[74] C. Blank, "Formal verification of register binding," in *Procs. of Workshop on Advances in Verification (WAVE) 2000*, 2000.

[75] Y. Morihiro and T. Toneda, "Formal verification of data-path circuits based on symbolic simulation," in *Procs. of 9th Asian Test Symposium 2000 (ATS 2000)*, pp. 329–336, Dec 2000.

[76] D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. Inc., 1989.

[77] J. Roy, "Parallel algorithms for high level synthesis," *Ph.D. Thesis*, Feb. 1983.

[78] S. Y. Kung and H. John, *VLSI and Modern Signal Processing*. Prentice Hall, 1984.

[79] A. Kundig, R. E. Suhrer, and J. Dahler, *Embedded Systems: New approaches to their formal description and design*. Springer-Verlag, 1986.

[80] O. H. Bailey., *Embedded Systems: Desktop Integration*. Wordware Publishing, Inc, 2005.

[81] F. J. Budinsky, M. A. Finnie, J. M. Vlissides, and P. S. Yu, "Automatic code generation from design patterns," *IBM Systems Journal*, vol. 35, no. 2, pp. 151–171, 1996.

[82] J. Ali and J. Tanaka, "Automatic code generation from the omt-based dynamic model," in *In Proceedings of the Second World Conference on Integrated Design and Process Technology*, (Austin, Texas), pp. 407–414, 1996.

[83] B. Vogel-Heuser, D. Witsch, and U. Katzke, "Automatic code generation from a uml model to iec 61131-3 and system configuration tools," *International Conference on Control and Automation, 2005. ICCA '05.*, vol. 2, pp. 1034–1039, 2005.

[84] C. Xi, L. JianHua, Z. ZuCheng, and S. YaoHui, "Modeling systemc design in uml and automatic code generation," in *ASP-DAC '05: Proceedings of the 2005 conference on Asia South Pacific design automation*, (New York, NY, USA), pp. 932–935, ACM Press, 2005.

[85] I. A. Niaz, *Automatic Code Generation From UML Class and Statechart Diagrams*. PhD thesis, University of Tsukuba, Japan, 2005.