

**Portland State University**  
**PDXScholar**

---

Dissertations and Theses

Dissertations and Theses

---


Spring 6-10-2013

## Equivalence Checking for High-Assurance Behavioral Synthesis

Kecheng Hao  
*Portland State University*

Let us know how access to this document benefits you.

Follow this and additional works at: [http://pdxscholar.library.pdx.edu/open\\_access\\_etds](http://pdxscholar.library.pdx.edu/open_access_etds)

 Part of the [Hardware Systems Commons](#), and the [Other Computer Sciences Commons](#)

---

### Recommended Citation

Hao, Kecheng, "Equivalence Checking for High-Assurance Behavioral Synthesis" (2013). *Dissertations and Theses*. Paper 1066.

[10.15760/etd.1066](https://doi.org/10.15760/etd.1066)

This Dissertation is brought to you for free and open access. It has been accepted for inclusion in Dissertations and Theses by an authorized administrator of PDXScholar. For more information, please contact [pdxscholar@pdx.edu](mailto:pdxscholar@pdx.edu).

Equivalence Checking for High-Assurance Behavioral Synthesis

by

Kecheng Hao

A dissertation submitted in partial fulfillment of the  
requirements for the degree of

Doctor of Philosophy  
in  
Computer Science

Dissertation Committee:

Fei Xie, Chair

Feng Liu

Sandip Ray

Bryant York

Fu Li

Portland State University  
2013

## ABSTRACT

The rapidly increasing complexities of hardware designs are forcing design methodologies and tools to move to the Electronic System Level (ESL), a higher abstraction level with better productivity than the state-of-the-art Register Transfer Level (RTL). Behavioral synthesis, which automatically synthesizes ESL behavioral specifications to RTL implementations, plays a central role in this transition. However, since behavioral synthesis is a complex and error-prone translation process, the lack of designers' confidence in its correctness becomes a major barrier to its wide adoption. Therefore, techniques for establishing equivalence between an ESL specification and its synthesized RTL implementation are critical to bring behavioral synthesis into practice.

The major research challenge to equivalence checking for behavioral synthesis is the significant semantic gap between ESL and RTL. The semantics of ESL involve untimed, sequential execution; however, the semantics of RTL involve timed, concurrent execution. We propose a sequential equivalence checking (SEC) framework for certifying a behavioral synthesis flow, which exploits information on successive intermediate design representations produced by the synthesis flow to bridge the semantic gap. In particular, the intermediate design representation after scheduling and pipelining transformations permits effective correspondence of internal operations between this design representation and the synthesized RTL implementation, enabling scalable, compositional equivalence checking. Certifications of loop and

function pipelining transformations are possible by a combination of theorem proving and SEC through exploiting pipeline generation information from the synthesis flow (*e.g.*, the iteration interval of a generated pipeline). The complexity brought by bubbles in function pipelines is creatively reduced by symbolically encoding all possible bubble insertions in one pipelined design representation. The result of this dissertation is a robust, practical, and scalable framework for certifying RTL designs synthesized from ESL specifications. We have validated the robustness, practicality, and scalability of our approach on industrial-scale ESL designs that result in tens of thousands of lines of RTL implementations.

DEDICATION

*To my parents, Fengming and Fanying*

*To my wife, Kai*

*To my daughter, Sophia*

## ACKNOWLEDGMENTS

It has been a long journey to finish this dissertation research. During my research, I got help from many people, including my teachers, collaborators, friends, and family, therefore I would like to take this opportunity to thank all of them.

First and foremost, I wish to express my thanks and great appreciation to my advisor Prof. Fei Xie for his patient guidance, enthusiastic encouragement and useful critiques. He is a wonderful advisor. He guided me to learn how to identify critical problems and resolve them, which is invaluable to my research. He is also an amazing researcher. His rare combination of strengths in both the practical and the theoretical has been a continuous inspiration to me. Without his help, I do not believe I can accomplish this dissertation.

I would also like to thank Dr. Sandip Ray for his enormous amount of time and effort that he spent on my research. All my papers related to this dissertation are finished by collaborating with Sandip. He is a great collaborator and an outstanding researcher. He has always helped me with my research through email and phone, even in weekends. My thanks to Dr. Jin Yang and Dr. Naren Narasimhan for generously providing feedback and discussing future work from an industrial point of view.

I am grateful to my other committee members, Prof. Fu Li, Prof. Feng Liu, and Prof. Bryant W. York, for inspiration in many ways and valuable feedback on my research proposal and dissertation. Thank Prof. Xiaoyu Song, who initially brought me on board to electronic design automation.

The dissertation also benefits from various discussions with current and former lab members: Yan Chen, Ping Hang Cheung, Nicholas T. Pilkington, Juncao Li, Sharookh Daruwalla, Zhenkun Yang, and Disha Gandhi. I also would like to thank my colleagues at Xilinx for exchanging knowledge of high-level synthesis, including Dr. Yiping Fan, Dr. Zhiru Zhang, Dr. Peichen Pan, and Dr. Guoling Han.

Finally, I must acknowledge my family for supporting me all the time. Thank you to my parents for always being supportive. Especially thanks to my lovely wife, Kai, for her love, patience, encouragement, and immense personal sacrifice.

## TABLE OF CONTENTS

<b>Abstract</b> . . . . .	<b>i</b>
<b>Dedication</b> . . . . .	<b>iii</b>
<b>Acknowledgments</b> . . . . .	<b>iv</b>
<b>List of Tables</b> . . . . .	<b>viii</b>
<b>List of Figures</b> . . . . .	<b>ix</b>
<b>Chapter 1 INTRODUCTION</b> . . . . .	<b>1</b>
1.1 Motivation and Problem Statement . . . . .	1
1.1.1 Motivation . . . . .	1
1.1.2 Problem Statement . . . . .	2
1.2 Contribution . . . . .	3
1.3 Related Work . . . . .	5
1.4 Dissertation Outline . . . . .	7
<b>Chapter 2 BACKGROUND</b> . . . . .	<b>8</b>
2.1 Behavioral Synthesis . . . . .	8
2.2 Solver Technology . . . . .	11
2.2.1 Binary Decision Diagram . . . . .	11
2.2.2 Boolean Satisfiability Problem . . . . .	13
2.2.3 Satisfiability Modulo Theories . . . . .	15
2.3 Scalable Verification Techniques . . . . .	16
2.3.1 Symbolic Simulation . . . . .	16
2.3.2 Equivalence Checking for Logic Synthesis . . . . .	17
<b>Chapter 3 EQUIVALENCE CHECKING</b> . . . . .	<b>19</b>
3.1 Clocked Control/Data Flow Graphs . . . . .	19
3.2 Circuit Model . . . . .	21
3.3 Correspondence between CCDFGs and Circuits . . . . .	22



3.4	Dual-Rail Simulation for Equivalence Checking . . . . .	23
3.5	Tool Implementation . . . . .	25
3.6	Experimental Results . . . . .	27
<b>Chapter 4</b>	<b>OPTIMIZATIONS . . . . .</b>	<b>29</b>
4.1	Motivation and Overview . . . . .	29
4.2	Cut-points . . . . .	29
4.3	Cut-loop Optimization . . . . .	30
4.4	Modular Analysis . . . . .	33
4.5	Experimental Results . . . . .	35
<b>Chapter 5</b>	<b>SEC FOR SYNTHESIZED LOOP PIPELINES . . . . .</b>	<b>38</b>
5.1	Motivation and Overview . . . . .	38
5.2	Challenges with Loop Pipelines . . . . .	39
5.3	SEC with Reference Model . . . . .	40
5.4	Experimental Results . . . . .	49
<b>Chapter 6</b>	<b>SEC FOR SYNTHESIZED FUNCTION PIPELINES . . . . .</b>	<b>51</b>
6.1	Motivation and Overview . . . . .	51
6.2	Challenges with Function Pipelining . . . . .	53
6.3	SEC for Function Pipelining . . . . .	55
6.3.1	Algorithm to build Reference Model . . . . .	57
6.3.2	SEC between CCDFGs and the RTL . . . . .	70
6.4	Experimental Results . . . . .	71
<b>Chapter 7</b>	<b>CONCLUSION AND FUTURE WORK . . . . .</b>	<b>74</b>
7.1	Summary of Contributions . . . . .	74
7.2	Future Research Directions . . . . .	75
7.2.1	Hierarchical Function Pipelines . . . . .	75
7.2.2	Verification of Behaviorally Synthesized Interfaces . . . . .	76
7.2.3	SEC for Compiler Transformations in Behavioral Synthesis . . . . .	76
<b>References</b>	<b>. . . . .</b>	<b>77</b>

## LIST OF TABLES

3.1	Bit-level equivalence checking statistics . . . . .	27
3.2	Word-level equivalence checking statistics . . . . .	28
4.1	Designs, features, and optimizations . . . . .	35
4.2	Word-level equivalence checking statistics . . . . .	36
5.1	Loop pipelining experimental results . . . . .	50
6.1	Function pipelining experimental results . . . . .	73

## LIST OF FIGURES

2.1	Input and output of behavioral synthesis . . . . .	10
2.2	Decision tree representation . . . . .	12
2.3	BDD transformations . . . . .	13
2.4	Simple cut-point example . . . . .	18
3.1	CCDFGs for the TEA encryption function . . . . .	20
3.2	Operation mapping between CCDFG and circuit . . . . .	22
3.3	Dual-rail simulation scheme for equivalence checking between CCDFG and circuit. . . . .	24
3.4	Framework of equivalence checker . . . . .	26
4.1	C source code and CCDFG for GCD . . . . .	31
4.2	Cut-loop optimization for GCD example . . . . .	32
4.3	Modular SEC for 3DES . . . . .	33
5.1	Example of loop pipeline . . . . .	39
5.2	Input and output CCDFGs of loop pipelining transformation . . . . .	42
5.3	Construction of scheduling steps . . . . .	44
5.4	Construction of edges . . . . .	45
5.5	Pipeline registers and forwarding . . . . .	48
6.1	Example of function pipeline . . . . .	52
6.2	Difference between un-pipelined version and pipelined version . . . . .	53
6.3	Hardware interface . . . . .	54
6.4	Pipelined CCDFGs for different bubble insertion scenarios . . . . .	56
6.5	Input and output CCDFGs of function pipelining transformation . . . . .	59
6.6	Generate pipeline registers . . . . .	61
6.7	Construction of scheduling steps and edges . . . . .	64
6.8	Insert guard variables and assignment . . . . .	65
6.9	Final pipelined CCDFG . . . . .	68

6.10 Waveform of pipeline forwarding . . . . . 69

## Chapter 1

# INTRODUCTION

## 1.1 MOTIVATION AND PROBLEM STATEMENT

### 1.1.1 Motivation

Recent years have seen increasingly higher complexities in hardware designs, resulting from advances in VLSI technology as well as growing demands on performance and power imposed by modern applications. Such complexities, in addition to stringent time-to-market requirements, make it challenging to develop reliable, high-quality systems through hand-crafted Register Transfer Level (RTL) implementations. This underlines the needs for modeling, synthesis, and validation of hardware at higher levels of abstraction and has motivated a gradual migration away from RTL towards Electronic System Level (ESL) which allows design functionality to be described abstractly in high-level languages such as SystemC or C/C++. However, practicality of ESL designs crucially depends on reliable tools for behavioral synthesis, that is, automated synthesis of a hardware circuit from its ESL description. Behavioral synthesis tools apply a sequence of transformations to compile the behavioral description to an RTL implementation.

Several behavioral synthesis tools are available today, *e.g.*, AutoESL [72], CatapultC [54], C-to-Silicon [11], Cynthesizer [23], Spark [25], and LegUp [13]. Nevertheless, and despite a great need, behavioral synthesis has not yet found wide acceptance in current industrial practice. A major barrier to its adoption is the

lack of designers’ confidence in correctness of synthesis tools themselves. The large semantic gap between a synthesized implementation and its behavioral description makes it hard to ensure that the synthesized implementation indeed conforms to its behavioral description. On the other hand, many employed behavioral synthesis transformations include complex optimizations to satisfy the growing demands of performance and power. Consequently, current synthesis tools are often either (a) error-prone or (b) overly conservative, thus often producing circuits of poor quality and performance. Therefore, developing tools and technologies to ensure correctness of behavioral synthesis tools is a critical issue to bring behavioral synthesis into practice. This is the main motivation of this dissertation research.

### 1.1.2 Problem Statement

To ensure correctness of behavioral synthesis, we employ a formal verification technique called sequential equivalence checking (SEC). We need to address the following three key research challenges brought about by the semantic gap between ESL and RTL descriptions of the same hardware:

- *How can we build a practical sequential equivalence checking framework?* The significant semantic gap makes the direct equivalence checking between ESL and RTL impractical. The challenge is to effectively close the semantic gap and build a practical checking framework.
- *How can we scale to industry designs?* The scale of real-world industrial designs synthesized by a state-of-the-art behavioral synthesis tool is typically tens of thousands of lines of RTL. The running time for equivalence checking is exponential in the size of a design. Due to the semantic gap, the existing optimizations for traditional hardware verification cannot be directly applied. To make our approach scalable, we need to develop optimizations that specially target equivalence checking for behavioral synthesis.

- *How can we verify the correctness of overlapping executions introduced by pipelines?* Loop pipelining and function pipelining are key transformations to improve the quality of results (QoR) of RTL implementations generated by behavioral synthesis. Loop pipelining reduces the latency of synthesized designs by concurrently executing operations in successive loop iterations. Function pipelining improves the throughput by allowing operations from successive invocations of a function to execute in parallel. However, they are complex transformations involving aggressive scheduling strategies to allow overlapping executions and careful control generation to eliminate hazards. This makes formal equivalence checking even more challenging.

## 1.2 CONTRIBUTION

We developed a scalable SEC framework for certifying behavioral synthesis flows, that makes use of the intermediate design representation from the synthesis tool as well as other synthesis information to achieve scalability. We defined a graph-based representation of the ESL specification, namely Clocked Control/Data Flow Graph (CCDFG) as the intermediate design representation. In particular, this dissertation makes the following contributions:

- A scalable SEC algorithm based on symbolic simulation for comparing CCDFG and RTL. Equivalence checking involves a word-level dual-rail symbolic simulation, which simulates two design representations simultaneously. In our approach, we simulate the CCDFG and the RTL implementation, respectively, and their simulations are synchronized by clock cycle.
- A set of key optimizations, which exploit the close correspondence between CCDFGs and their synthesized RTL designs. *Cutpoints* reduce lengths of symbolic expressions by replacing verified sub-circuits by new symbol values.

*Cut-loop* partitions SEC for a loop into three checks to avoid expensive fix-point computation. *Modular analysis* optimizes SEC by replacing verified sub-modules by uninterpreted functions.

- An approach to certifying behaviorally synthesized loop pipelines. Our approach works by (1) constructing a provably correct loop pipeline reference model from the ESL specification, and (2) applying sequential equivalence checking between this reference model and synthesized RTL. The key insight is that a parameterized, synthesis-guided reference transformation on CCDFG permits comparison with RTL even after mappings with the original ESL specification have been destroyed by an aggressive transformation such as pipelining. Furthermore, the approach permits smooth integration with pipeline-oblivious optimizations such as cut-loop.
- An approach to certifying function pipelining in behavioral synthesis. We develop a reference function pipelining transformation, which takes certain pipeline parameters from behavioral synthesis to generate a pipeline reference model. The key is that bubble insertion is faithfully captured by symbolically encoding bubbles in the reference model. We check the equivalence between the reference model and the RTL implementation. The mapping between behavioral level operations and RTL functional units are still preserved, therefore some key optimizations, such as cutpoints, are applicable.

Note that certification with our approach assumes that the high-level transformations performed by behavioral synthesis before pipelining are correct. Such transformations are typically compiler transformations (e.g., loop unrolling, code motion, dead code elimination, etc.) and scheduling. Certainly many of these transformations are complex, and a complete certification of the synthesis flow requires certification of these high-level transformations as well. We do not address



certification these high-level transformations for this dissertation for several reasons. First, these transformations are generic compiler transformations. For instance, AutoESL, a widely used commercial synthesis tool, and LegUp, an academic synthesis tool, make use of the open-source LLVM compiler transformations [50]. As such they are more trusted than the low-level transformations which often involve manual tweaks to squeeze out extra efficiency. Second, these transformations are already being studied elsewhere in the context of compiler verification [48, 67, 74]. Finally, related efforts at the University of Texas and the Portland State University are focused on the use of theorem proving techniques for certification of many of the high-level transformations necessary. When combined with their certified transformations, our framework can be used to certify an entire synthesis flow.

### 1.3 RELATED WORK

Several early approaches have been proposed to verify the correctness of the pioneering behavioral synthesis tools. An early effort on verification of high-level synthesis targeting the behavioral portion of VHDL was proposed by Chapman [14], which aimed to verify parts of a high-level synthesis system by giving semantics to the representation languages used. A translation from behavioral VHDL to dependence flow graphs [36] was verified by structural induction based on the CSP [32] semantics. There has been research on certified synthesis of hardware from formal languages such as HOL [28] in which a compiler that automatically translates recursive function definitions in HOL to clocked synchronous hardware has been developed. A certified hardware synthesis from programs in Esterel (a synchronous design language) has also been developed [61] in which a variant of Esterel was embedded in HOL to enable formal reasoning.

There has been much research on sequential equivalence checking between RTL and gate-level hardware designs [4, 38]. Research has also been done on combinational equivalence checking between high-level designs in software-like languages

(*e.g.*, SystemC) and RTL designs [33]. There has also been effort for SEC between software specifications and hardware implementations [21]: GSTE assertion graphs [73] were extended so that an assertion graph edge have pre and post condition labels, and also associated assignments that update state variables. There has also been work on equivalence checking with other graph representations, *e.g.*, Signal Flow Graph [17].

In recent years, promising progress on equivalence verification between system-level models and RTL has been made in both academia and industry [53]. Kundu et al. [46] presents an approach to validate the result of behavioral synthesis using insights from translation validation, automated theorem proving and relational approaches to reasoning about programs. This approach targeted compiler transformations, so it cannot verify the correctness of scheduling, binding and finite state machine (FSM) generation. Clark et al. [18] proposed an algorithm that checks behavioral consistency between an ANSI-C program and a circuit given in Verilog using Bounded Model Checking. Both the circuit and the program are unwound and translated into a formula that represents behavioral consistency. The formula is then checked using a SAT solver. Kroening [44] has further enhanced this algorithm by using predicate abstraction and induction. These approaches aim to check if the RTL holds the same property as the corresponding ANSI-C program, not equivalence checking. The Sequential Logic Equivalence Checker (SLEC) from Calypto [12] can verify RTL implementations using system models written in C/C++ or SystemC, without requiring additional testbenches or assertions. SLEC utilizes a novel technique to reduce the SEC problem to a cycle-accurate designs from the original designs, on which standard equivalence checking techniques can then be deployed [15]. Hector [42] from Synopsys is a formal equivalence checking framework to address the system level to RTL formal verification problem, which integrates multiple bit-level and word-level equivalence checking

techniques. It employs an efficient formal model constructed from high-level descriptions using symbolic simulation [43]. Several optimizations can be applied to minimize the size of the formal model to reduce the complexity. Furthermore, an approach has been proposed to handle memory interfaces by using memory mapping provided by the user as invariants for an induction proof [41]. However, these two industrial tools do not have published benchmarks for comparison.

There is a significant literature on verifying pipelined microprocessors [10, 37, 49, 69], which has parallels with our work. Comparing function pipelines generated by behavioral synthesis with pipelines in microprocessors, certifying pipelines generated by behavioral synthesis is more challenging due to: (1) pipelines can be very deep; (2) each pipeline stage can be quite complex. There has been very little published work on formal verification of pipelines generated by behavioral synthesis. Nevertheless, any viable SEC framework for behavioral synthesis must handle pipelining transformations. To our knowledge current implementations either involve cost-prohibitive input-output comparison or require the user to provide the requisite mappings.

## 1.4 DISSERTATION OUTLINE

The rest of this dissertation is organized as follows. In Chapter 2, we give a brief overview of background including behavioral synthesis flows and verification technologies. In Chapter 3, we present our intermediate representation and discuss in detail our approach to equivalence checking based on word-level symbolic simulation. In Chapter 4, we discuss three optimizations targeting different design features. We illustrate our approach to equivalence checking for behaviorally synthesized loop pipelines and function pipelines in Chapter 5 and Chapter 6, respectively. In Chapter 7, we summarize the contribution and discuss future work.

## Chapter 2

**BACKGROUND****2.1 BEHAVIORAL SYNTHESIS**

With the rapid increase of complexity in System-on-Chip (SoC) designs, the Electronic Design Automation (EDA) community is becoming more interested in designing hardware with a behavioral level model, rather than an RTL description. This and the increased use of high-level languages in behavioral modeling has led to a renewed interest in behavioral synthesis, both in industry and in academia [70].

A behavioral synthesis tool accepts a behavioral description, together with a library of hardware resources; it performs a sequence of transformations on the description to generate an RTL implementation. The transformations can be roughly partitioned into the following three phases.

- The first phase involves *compiler transformations*. These include loop unrolling, common subexpression elimination, etc. Furthermore, expensive operations (*e.g.*, division) are often replaced with simpler ones (*e.g.*, subtraction).
- The second phase is *scheduling*, which determines the clock step for each operation. The ordering between operations is constrained by the data and control dependencies. Scheduling transformations include chaining operations across conditional blocks and decomposing one operation into a sequence of multi-cycle operations based on resource constraints.

- The third phase is *resource binding and control synthesis*, which binds operations to functional units, allocates and binds registers, and generates the control circuit to implement the schedule.

After these transformations, the RTL implementation is generated, which is subjected to further manual optimizations to fine-tune for performance and power. Each synthesis transformation is non-trivial. The result of their composition is a hardware implementation with large semantic distance from its input description. As an example, consider the synthesis of the Tiny Encryption Algorithm (TEA) [71]. Figure 2.1 shows a high-level C specification and the circuit synthesized by AutoESL. TEA, of course, is only a pedagogical algorithm, and indeed, rather weak in cryptographic strength; nevertheless, the example highlights some transformations involved in behavioral synthesis. The following transformations are involved in synthesis of the circuit.

- In the first phase, *constant propagation* removes unnecessary variables and operations.<sup>1</sup> For instance, variable *delta* is replaced by constant value.
- In the second phase, a key scheduling transformation performed is *pipelining*, to enable overlapping execution of operations from different loop iterations.
- In the third phase, operations are bound to hardware resources (*e.g.*, the “+” operation to a hardware adder); furthermore, a finite-state machine is generated to control circuit operations.

Each synthesis transformation must respect a number of implicit design invariants. For instance, paralleling operations along different loop iterations must avoid race conditions, and scheduling must respect underlying data dependencies. Since such considerations are entangled with low-level optimization heuristics, it is easy

---

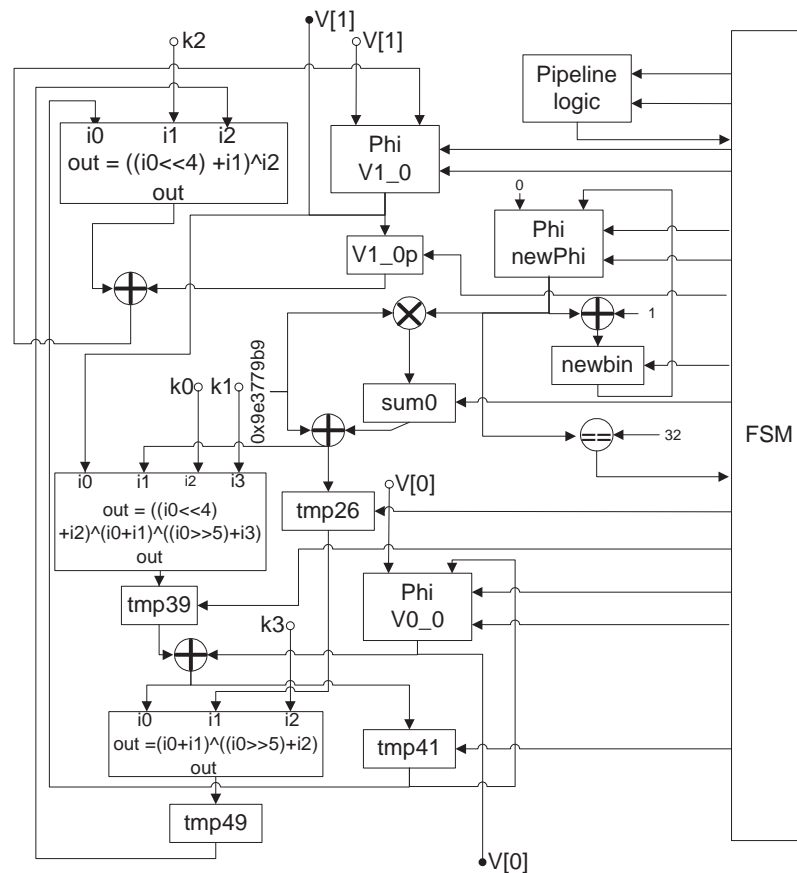
<sup>1</sup>Another compiler transformation that could be performed is loop unrolling. We avoided it for presentation simplicity.

```

void encrypt (uint32_t* v, uint32_t* k) {
    uint32_t v0=v[0], v1=v[1], sum=0, i;
    uint32_t delta=0x9e3779b9;
    uint32_t k0=k[0], k1=k[1], k2=k[2], k3=k[3];
    for (i = 0; i < 32; i++) {
        sum += delta;
        v0 += ((v1 << 4)+ k0) ^ (v1 + sum) ^((v1 >> 5) + k1);
        v1 += ((v0 << 4)+ k2) ^ (v0 + sum) ^((v0 >> 5) + k3);
    }
    v[0]=v0; v[1]=v1;
}

```

(A) C code for TEA



(B) Schema of RTL

Figure 2.1: Input and output of behavioral synthesis

to have errors in the synthesis tool itself, leading to synthesis of buggy designs. On the other hand, the semantic distance pointed to above makes direct comparison of executions of the synthesized RTL and its input description very challenging if not infeasible. Indeed, attempts to perform such comparison through sequential equivalence checking requires full, cost-prohibitive symbolic co-simulation between the C and the RTL to check their input/output correspondence [33].

## 2.2 SOLVER TECHNOLOGY

### 2.2.1 Binary Decision Diagram

Binary Decision Diagrams (BDDs) have been widely employed in many applications. Though BDDs are not new [47], Bryant’s pioneering work renewed the interest of many researchers [8]. He proposed Reduced Ordered Binary Decision Diagrams (ROBDDs, or OBDDs for short). The key insight is that reduced and ordered binary decision diagrams are a canonical representation of Boolean functions. Canonicity reduces the semantic notion of equivalence to the syntactic notion of isomorphism. Thus, checking the equivalence of two Boolean formulas can be reduced to comparisons of BDDs which can be checked in constant time.

Given a Boolean function, it can be represented as a rooted, directed acyclic graph, which is actually a tree. Figure 2.2 (b) illustrates a representation of function  $f(x_1, x_2, x_3)$  defined by the truth table given in Figure 2.2 (a). The variable ordering is given as  $x_1 < x_2 < x_3$ . Each nonterminal vertex  $v$  has arcs directed toward two children:  $lo(v)$  (shown as a dashed line) corresponding to the case where  $v$  is assigned 0, and  $hi(v)$  (shown as a solid line) corresponding to the case where  $v$  is assigned 1. Each terminal vertex is labeled 0 or 1. For a given assignment to the input variables of  $f$ , the return value of  $f$  can be determined by a path from the root to a terminal vertex, following the branches indicated by the values of nonterminal vertices.

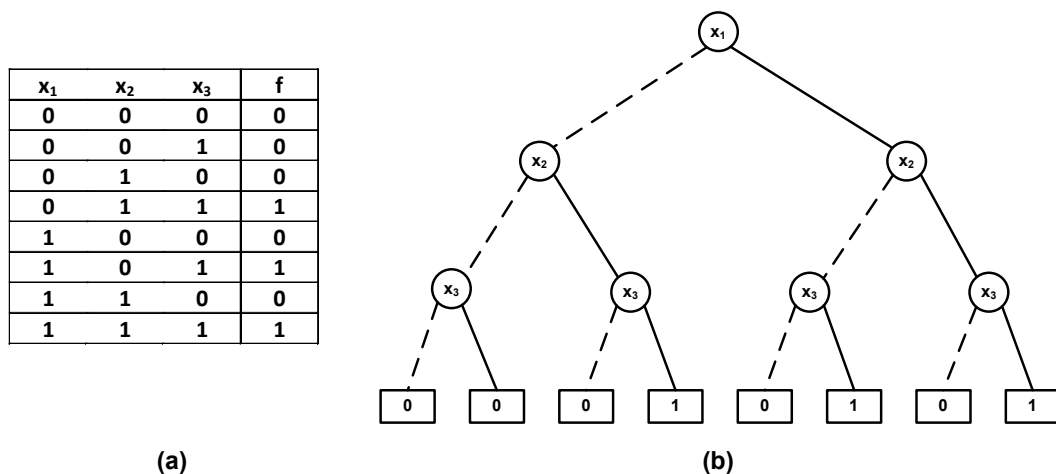


Figure 2.2: Decision tree representation

A decision tree can be reduced to a BDD by applying the following three transformations: (1) Remove duplicate terminals. This transformation eliminates all but one terminal vertex with a given label and redirects all arcs into the eliminated vertices to the remaining one (shown in Figure 2.3 (a)). (2) Remove duplicate nonterminals. In this step, if nonterminal vertices  $u$  and  $v$  have  $lo(u) = lo(v)$ , and  $hi(u) = hi(v)$ , then one of the two vertices can be eliminated. All incoming arcs to the eliminated one are redirected to the other vertex (shown in Figure 2.3 (b)). (3) Remove redundant tests. If nonterminal vertex  $v$  has  $lo(v) = hi(v)$ , then this vertex can be eliminated. All incoming arcs to  $v$  are redirected to one of its children (shown in Figure 2.3 (c)).

BDDs have proven to be a successful representation for model checking on many practical applications. However one limitation of BDD-based approaches is that the size of the BDD heavily depends on the variable ordering. For instance, given a Boolean expression  $a_1 \cdot b_1 + a_2 \cdot b_2 \dots + a_n \cdot b_n$ , ordering variables as  $a_1 < b_1 < \dots < a_n < b_n$  yields an BDD with  $2n$  nonterminal vertices. On the other hand, ordering variables as  $a_1 < \dots < a_n < b_1 < \dots < b_n$  yields an BDD with  $2(2^n - 1)$  nonterminal vertices. For large values of  $n$ , the exponential growth of the



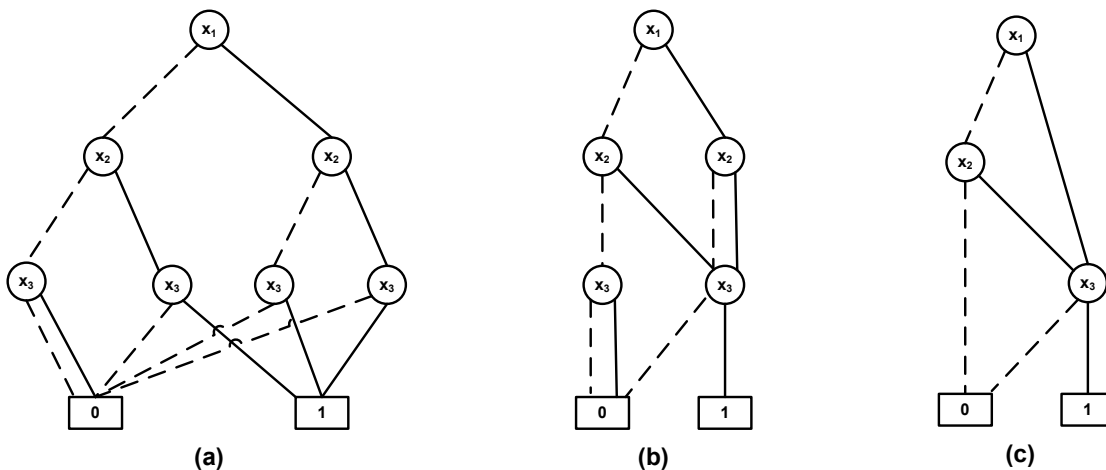


Figure 2.3: BDD transformations

second ordering has a dramatic effect on runtime and memory usage comparing with the first linear growth. Unfortunately, finding the best variable ordering is an NP-complete problem. In practice, the ordering is chosen either manually or by a heuristic analysis of the particular system to be represented [24, 51, 35].

### 2.2.2 Boolean Satisfiability Problem

Boolean Satisfiability (SAT) is a decision problem of determining if there exist suitable value assignments to the variables to satisfy the propositional logic formula. SAT is the first known example of NP-complete decision problem [19], which means that, unless  $P = NP$ , all SAT algorithms require worst-case exponential time. Modern SAT algorithms are effective to deal with large search spaces by exploiting the structure of the problem [65, 55, 27]. SAT techniques are widely used in a number of areas, such as combinational equivalence checking [7], model checking [6, 63], automatic test pattern generation (ATPG) [16], FPGA routing [56] and planning [40].

Currently, most state-of-the-art SAT solvers require the propositional formulas to be represented in Conjunctive Normal Form (CNF) as defined in Definition 3 [34]. A CNF formula may be viewed as a set of clauses and a clause may be viewed as a set of literals.

**Definition 1 Literal.** A literal is either a variable  $p$  or its negation  $\neg p$ . The first case is called a positive literal; the second is called a negative literal.

**Definition 2 Clause.** A clause is a finite disjunction of literals, *e.g.*,  $l_1 \vee l_2 \vee l_3 \dots$ , where  $l_i$  is a literal.

**Definition 3 Conjunctive Normal Form.** A propositional formula is in Conjunctive Normal Form (CNF) if it is a finite conjunction of clauses, *e.g.*,  $C_1 \wedge C_2 \wedge C_3 \dots$ , where  $C_i$  is a clause.

An *assignment*  $A$  for a set of variables  $X$  is a function  $A : X \rightarrow \{0, u, 1\}$ , where  $0 \leq u \leq 1$ . Here, 0 and 1 represent *false* and *true*, respectively. Given an assignment, clauses and CNF formulas can be characterized as *unsatisfied*, *satisfied*, or *unresolved* [64]. The SAT problem for a CNF formula  $\varphi$  consists in deciding whether there exists an assignment to the problem variables, such that  $\varphi$  is satisfied, or proving that no such assignment exists. An assignment that satisfies a formula  $\varphi$  is called a *satisfying assignment*.

A combinational circuit can be translated into some intermediate representation, which can be used to generate CNF formulas. Combinational Boolean circuits [6] is one of the most accepted intermediate representations. Combinational Boolean circuits are composed of gates and connections between gates. In Combinational Boolean circuits, the notation  $y = Op(x_1, x_2)$  denotes a gate which has two inputs  $x_1$  and  $x_2$  and single output  $y$ , and  $Op$  is one of the basic logic operations, such as *AND*, *OR*, etc. Converting Boolean circuits to CNF is straightforward, and follows the procedure first outlined by G. Tseitin [68].

### 2.2.3 Satisfiability Modulo Theories

Although SAT solvers have achieved success in many practical applications, some applications require greater modeling flexibility than plain SAT; for instance, a theory of array of integers is more effective in modeling memory usage of a program. On the other hand, general-purpose first-order theorem provers are typically not able to solve such formulas directly. The main reason for this is that many applications require not only general first-order satisfiability, but rather satisfiability with respect to some background theory, which fixes the interpretations of certain predicate and function symbols [2].

Satisfiability Modulo Theories (SMT) is the problem of deciding the satisfiability of a first-order formula with respect to some decidable first-order theory  $\mathcal{T}$ . It requires deciding the satisfiability of formulae which are Boolean combinations of atomic Boolean propositions and atomic propositions in  $\mathcal{T}$ , so that Boolean reasoning is carried out by powerful SAT solvers while reasoning in the theory  $\mathcal{T}$  is carried out by efficient theory-specific decision procedures.

Most SMT solvers use Nelson-Oppen [57] method which combines decision procedures for different decidable theories under certain conditions to generate a decision procedure for their composition. These solvers support the theories of integers, reals, lists, arrays, bit vectors, etc. Therefore, it allows us to model hardware circuits at word-level rather than bit-level. SMT solvers also support uninterpreted functions. For instance, an SMT solver can determine whether  $f(x) = f(y)$ , if  $x = y$ . This allows us to model the hierarchies in hardware circuits by modeling a lower-level circuit as an uninterpreted function.

Currently there are two main approaches to SMT solving: *eager* and *lazy* [42]. *Eager* SMT solvers first try to solve the word-level problem by employing pre-processing, rewrites and abstraction. If the rewrites are not sufficient, the word-level formula is translated into a bit-level formula and use a SAT solver to determine the satisfiability. One of the advantages of *eager* SMT solvers is that they can

directly leverage any efficient SAT solver. Some *eager* SMT solvers are BAT [52] and STP [26]. In contrast, *lazy* SMT solvers integrate theory specific procedures for the background theory with a SAT-solver. A given formula  $\phi$  is abstracted to a Boolean formula  $\phi_b$ . The abstraction is generated by replacing all atomic theory predicates in  $\phi$  by Boolean variables. The Boolean variables of the abstracted formula  $\phi_b$  are sub-formulas of formula  $\phi$  corresponding to sub-formulas of the background theory. The satisfiability of  $\phi_b$  is checked using a SAT solver. If  $\phi_b$  is unsatisfiable, then  $\phi$  is also unsatisfiable. Some *lazy* SMT solvers are Yices [20] and CVC3 [3].

In this research, we employ CVC3, one of the most successful SMT solvers, which is being developed at New York University and University of Iowa. CVC3 provides several different user interfaces including high-level APIs for both C and C++, an interactive command-driven interface, and a file interface.

## 2.3 SCALABLE VERIFICATION TECHNIQUES

### 2.3.1 Symbolic Simulation

Simulation is the most common method for testing and debugging hardware designs. But the problem is that one simulation run can only validate one test case. To fully verify a hardware design, engineers must exhaustively simulate the entire set of test cases to explore the whole state space, which is extremely time-consuming. Symbolic simulation allows us to compute information on the entire set of values in a single simulation run, because the set of test vectors is encoded symbolically, instead of using a specific element of the set [9]. This approach dramatically improves the efficiency of design validation.

Consider a 2-bit *AND* operator which has two 2-bit inputs  $A$  and  $B$  and a 1-bit output  $C$ . In order to fully verify this operator, a conventional simulator must try all 16 possible test vectors. But for symbolic simulation, we treat the inputs as

two 2-bit symbols  $A$  and  $B$  and the output of the simulation is  $C = A \wedge B$  by only one simulation run.

However, symbolic simulation also has two main bottlenecks when applying to verify large designs [33]. First, because symbolic simulation enumerates all possible execution paths, the number of paths to be explored may grow exponentially. Second, the terms representing the symbolic values of variables may also blow-up exponentially. Moreover, symbolic exploration of loops may lead to long executions, which may cause further blow-up. Although modern SMT solvers are able to handle such blow-ups to a certain extent, the performance is reduced significantly [66].

### 2.3.2 Equivalence Checking for Logic Synthesis

Our research leverages the success of equivalence checking for logic synthesis [33, 5, 7], and employs these ideas for equivalence checking for behavioral synthesis. Next, we provide an overview of the equivalence checking concepts.

Equivalence checking between RTL descriptions and gate-level implementations of combinational circuits is a mature field with decades of research [38, 4]. To check whether two combinational circuits are functionally equivalent, we need to prove that, for all possible inputs, both combinational circuits have the same outputs. Hardware circuits are modeled as Boolean expressions, so the problem of checking whether two circuits are equivalent is converted to the problem of determining whether two Boolean expressions are equivalent.

Consider the simple example in Figure 2.4. We can use symbolic simulation to compute the relationship between inputs and outputs. Given the two circuits and the same input symbols, the outputs are symbolic expressions in terms of the inputs. For the left-hand circuit in Figure 2.4, the value of the final output  $f$  is  $a \oplus ((b \wedge c) \wedge d)$ . Similarly, we can compute that the output of the right-hand circuit is  $a \oplus (b \wedge (c \wedge d))$ . To verify the equivalence of these two circuits, we just

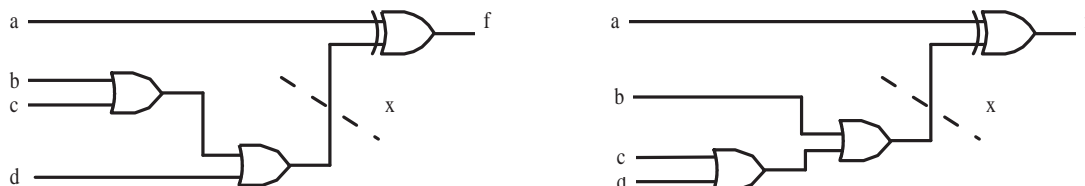


Figure 2.4: Simple cut-point example

need to verify whether these two expressions are equivalent.

As discussed above, one bottleneck of symbolic simulation is the exponential blow-up of the expression lengths. The major practical reduction technique for combinational equivalence checking is cut-points [5, 7]. The main idea is to look for the corresponding points in the two circuits that can be proven to be equivalent; then the equivalent circuits can be cut out of circuits and replaced by new primary symbols. For instance, in Figure 2.4, to introduce cut-point  $x$ , we first verify that  $(b \wedge c) \wedge d$  is equivalent to  $b \wedge (c \wedge d)$ . Then we cut the sub-circuits off and introduce new symbol  $x$  to represent the equivalent circuits. Then we can verify that  $f$  is equivalent to  $g$  because they are both equal to  $a \oplus x$ . Therefore, the complexity of verification is reduced. In general, the method is conservative: if the proof fails, we cannot conclude that the two circuits are inequivalent. The reason is that when we introduce new symbols for cut-points, we may lose constraints. The situation that two circuits are equivalent but equivalence checker reports inequivalence is called *false negative*. In general, the solution to this problem is to re-introduce constraints on the cut-points [33, 7].

## Chapter 3

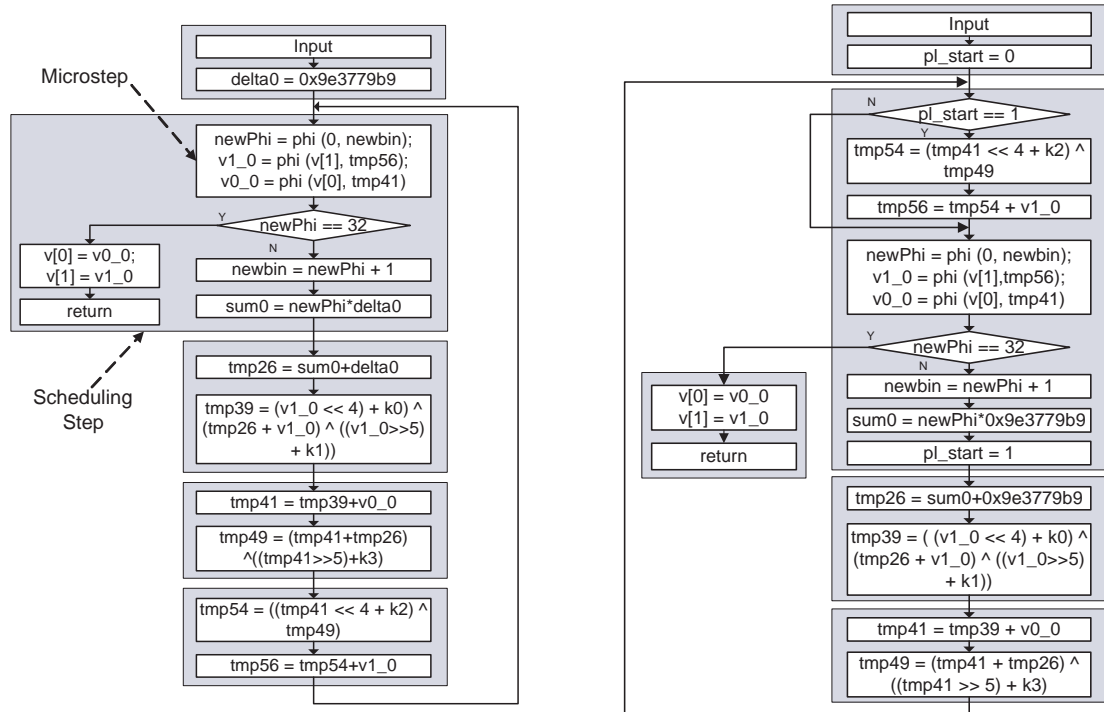
**EQUIVALENCE CHECKING**

In this chapter, we present a graph-based design representation, called Clock Control/Data Flow Graph (CCDFG), as our intermediate representation. Our equivalence checking between a CCDFG and its synthesized RTL implementation is based on dual-rail symbolic simulation. The checking approach has been implemented to be fully automatic.

**3.1 CLOCKED CONTROL/DATA FLOW GRAPHS**

A CCDFG can be viewed as a formal *control/data flow graph* (CDFG) — used as internal representation in most synthesis tools — augmented with a schedule. The semantics of CCDFG are formalized in the logic of the ACL2 theorem prover [39]. Figure 3.1 shows two CCDFGs for the TEA encryption function: an initial CCDFG derived from the C code, and its successive transformation after pipelining. This section briefly discusses the formulation of a CCDFG; for a more complete account, see [59].

The formalization of CCDFG assumes that the underlying language provides the semantics for *primitive operations* (*e.g.*, arithmetic operations, comparison, etc.). The key components of the formalization are (1) control and data flow graphs, (2) microstep partition, and (3) schedule. Following standard conventions, the control flow is broken up into of basic blocks; correspondingly data dependencies follow the “read after write” paradigm:  $op_j$  is dependent on  $op_i$  if  $op_j$  occurs after  $op_i$  in a control path and computes an expression over some variable  $v$  that is



(A) Initial CCDFG of TEA

(B) CCDFG after pipelining.

Figure 3.1: CCDFGs for the TEA encryption function

assigned most recently by  $op_i$  in the path. A *microstep partition* is a partitioning of operations in a basic block such that if  $op_i$  and  $op_j$  are in the same partition then their execution order is irrelevant to control and data dependencies. Each component of a microstep partition is a *microstep*. A *schedule* is a *grouping* of microsteps; informally, if  $m_0$  and  $m_1$  belong to the same scheduling step then they are executed within the same clock cycle. A CCDFG execution is formalized through state-based semantics. A *CCDFG state* (resp., *CCDFG input*) is a valuation of the state (resp., input) variables. Given a sequence of inputs, an *execution* of a CCDFG  $G$  with microstep partition  $M$  and schedule  $T$  is a sequence of CCDFG states that corresponds to an evaluation of the microsteps of  $M$  respecting  $T$ .

*Remark Conventions.* For a given CCDFG  $G \triangleq \langle G_{CD}, M, T \rangle$  and a set  $t \in T$ , we



use the term “projection of  $G$  on  $t$ ” to mean the CCDFG  $G_t \triangleq \langle G'_{CD}, M', \{t\} \rangle$  where  $G'_{CD}$  and  $M'$  contain only the operations in  $G_{CD}$  and  $M$  respectively, that are members of  $t$ . For a set  $T_0 \subseteq T$ , we use “projection of  $G$  on  $T_0$ ” to denote the following graph  $G'$ . The nodes of  $G'$  are given by the set  $\mathcal{N} \triangleq \{G_t : t \in T_0\}$ ; given  $g_0, g_1 \in \mathcal{N}$ , there is an edge from  $g_0$  to  $g_1$  if there are operations  $o_1$  and  $o_2$  such that  $o_1 \in g_0$ ,  $o_2 \in g_1$  and there is an edge from  $o_1$  to  $o_2$  in  $G_{CD}$ .

Since a schedule is a partition of microsteps,  $T_0$  induces a partition of  $G_{CD}$  such that if  $t_0 \neq t_1$  the partition induced by  $t_0$  is disjoint from that induced by  $t_1$ . Given a set  $T$  of scheduling steps, one can describe the CCDFG  $G \triangleq \langle G_{CD}, M, T \rangle$  uniquely as the triple  $\langle S, E, M \rangle$  where  $S$  and  $E$  denote the nodes and edges of the projection of  $G$  on  $T$ , and  $M$  is the set of microstep partitions refined by  $T$ . We use this view in the rest of the dissertation.

### 3.2 CIRCUIT MODEL

We represent a circuit as a Mealy machine specifying the updates to the state elements (latches) in each clock cycle. Our formalization of circuits is typical in traditional hardware verification, but we make combinational nodes explicit to facilitate the correspondence with CCDFGs. A circuit is a tuple  $M = \langle I, N, F \rangle$  where  $I$  is a vector of inputs;  $N$  is a pair  $\langle N_c, N_d \rangle$  where  $N_c$  is a set of *combinational nodes* and  $N_d$  is a set of *latches*; and  $F$  is a pair  $\langle F_c, F_d \rangle$  where  $F_c$  maps each combinational node  $c \in N_c$  to an expression over  $N_c \cup N_d \cup I$  and for each latch  $d \in N_d$ ,  $F_d$  maps each latch  $d$  to  $n \in N_c \cup N_d \cup I$  where  $F_d$  is a delay function which takes the current value of  $n$  to be the next-state value of  $d$ .

A *circuit state* is an assignment to the latches in  $N_d$ . Given a sequence of valuations to the inputs  $i_0, i_1, \dots$ , a *circuit trace* of  $M$  is the sequence of states  $s_0, s_1, \dots$ , where (1)  $s_0$  is the initial state and (2) for each  $j > 0$ , the state  $s_j$  is obtained by updating the elements in  $N_d$  given the state valuation  $s_{j-1}$  and input valuation  $i_{j-1}$ . The

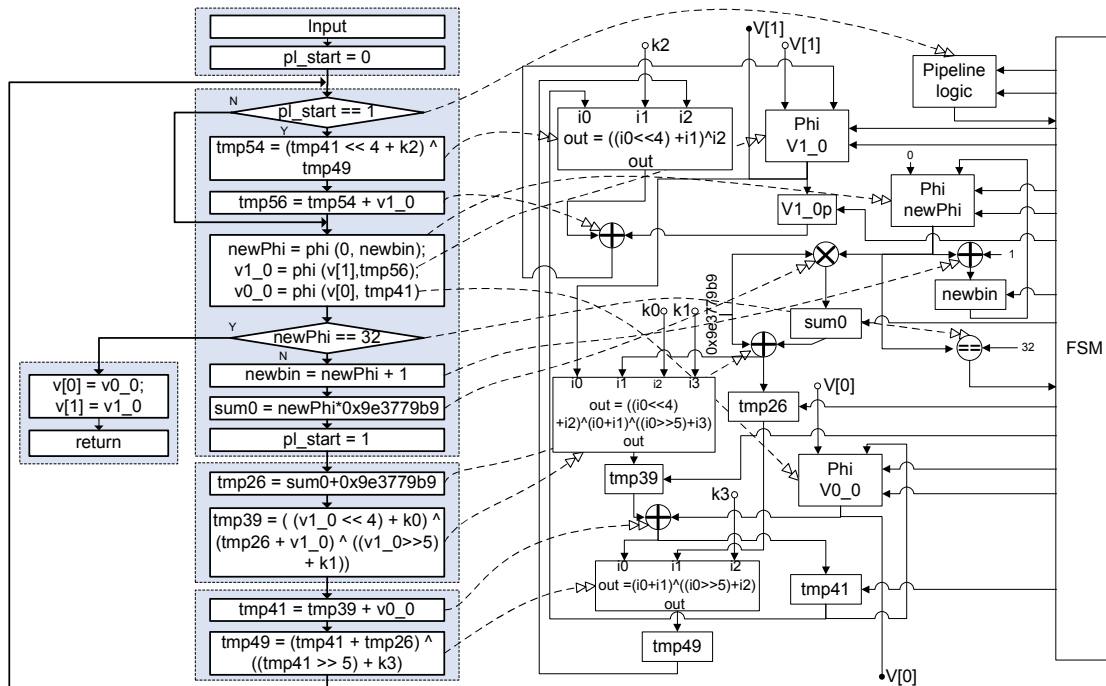


Figure 3.2: Operation mapping between CCDFG and circuit

*observable behavior* of the circuit is the sequence of valuations of the *outputs* which are a subset of latches and combinational nodes.

### 3.3 CORRESPONDENCE BETWEEN CCDFGS AND CIRCUITS

Given a CCDFG  $G$  and a synthesized circuit  $M$ , it is tempting to define a notion of correspondence as follows: (1) establish a fixed mapping between the state variables of  $G$  and the latches in  $M$ , and (2) stipulate an execution of  $G$  to be equivalent to an execution of  $M$  if they have the same observable behavior. However, this does not work in general since the mappings between state variables and latches may be different in each clock cycle. To address this, we introduce  $EMap : ops \rightarrow N_c$ , mapping CCDFG *operations* to the combinational nodes in the circuit: each operation is mapped to the combinational node that implements the operation;

the mapping is independent of clock cycles. Figure 3.2 shows the mapping for the synthesized circuit of TEA. Recall from Section 2.1 that the FSM decides the *control signals* for the circuit; the FSM is thus excluded from the mapping.

We now define the equivalence between  $G$  and  $M$ . A CCDFG state  $x$  of  $G$  is equivalent to a circuit state  $s$  of  $M$  with respect to an input  $i$  and a microstep partition  $t$ , if for each operation  $op$  in  $t$ , the inputs to  $op$  according to  $x$  and  $i$  are equivalent to the inputs to  $EMap(op)$  according to  $s$  and  $EMap(i)$ , i.e., the values of each input to  $op$  and the corresponding input to  $EMap(op)$  are equivalent, and the outputs of  $op$  are equivalent to the outputs of  $EMap(op)$ .

Given a CCDFG  $G$  and a circuit  $M$ ,  $G$  is equivalent to  $M$  if and only if for any execution  $[x_0, x_1, x_2, \dots]$  of  $G$  generated by an input sequence  $[i_0, i_1, i_2, \dots]$  and by microstep partition  $[t_0, t_1, \dots]$  of  $G$ , and the state sequence  $[s_0, s_1, s_2, \dots]$  of  $M$  generated by the input sequence  $[EMap(i_0), EMap(i_1), EMap(i_2), \dots]$ ,  $x_k$  and  $s_k$  are equivalent with respect to  $t_k$  under  $i_k$ ,  $k \geq 0$ .

### 3.4 DUAL-RAIL SIMULATION FOR EQUIVALENCE CHECKING

We check equivalence between CCDFG  $G$  and circuit  $M$  by dual-rail symbolic simulation (Figure 3.3); the two rails simulate  $G$  and  $M$  respectively, and are synchronized by clock cycle. The equivalence checking in clock cycle  $k$  is conducted as follows:

1. The current CCDFG state  $x_k$  and circuit state  $s_k$  are checked to see whether for the input  $i_k$ , the inputs to each operation  $op$  in the scheduling step  $t_k$  are equivalent to the inputs to  $EMap(op)$ . If yes, continue; otherwise, report inequivalence.
2.  $G$  is simulated by executing  $t_k$  on  $x_k$  under  $i_k$  to compute  $x_{k+1}$  and recording the outputs of each  $op \in t_k$ .  $M$  is simulated for one clock cycle from  $s_k$  under input  $EMap(i_k)$  to compute  $s_{k+1}$ . The outputs for each  $op$  are checked for

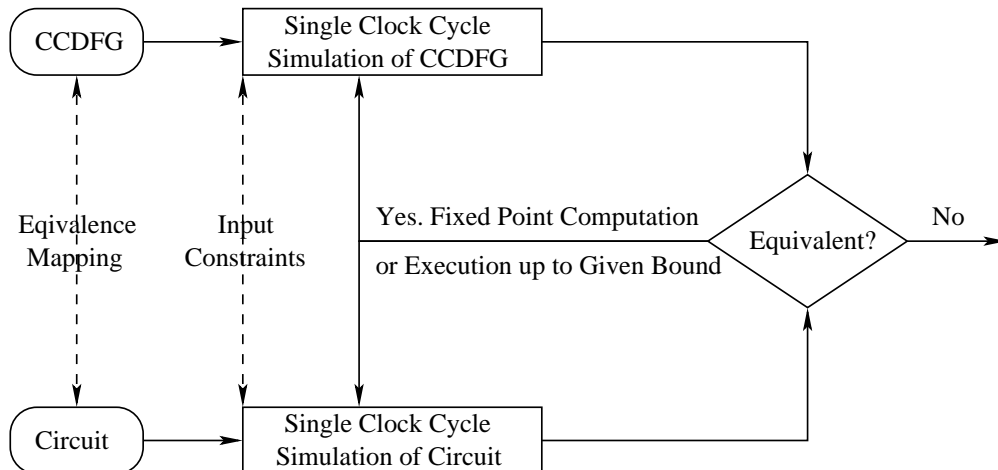


Figure 3.3: Dual-rail simulation scheme for equivalence checking between CCDFG and circuit.

equivalence with the outputs of  $EMap(op)$ . If yes, continue; otherwise, report inequivalence.

3. The next scheduling step  $t_{k+1}$  is determined from control flow. If  $t_k$  has multiple outgoing control edges, the last microstep of  $t_k$  executed is identified. The outgoing control edge from this microstep whose condition evaluates to true leads to  $t_{k+1}$ .

We permit both bounded and unbounded (fixed-point) simulations. In particular, the simulation proceeds until (i) the equivalence check fails, (ii) the end of a bounded input sequence is reached, or (iii) a fixed point is reached for an unbounded input sequence.

The bit-level and word-level checkers are complementary. The bit-level checker ensures that the equivalence checking is decidable, while the word-level checker provides the optimizations which are crucial to scalability. The word-level checker can make effective use of results from bit-level checking in many cases. One typical scenario is as follows. Suppose  $M$  is a design module of modest complexity but

is awkward to check at word-level. Then the bit-level checker is used to check the equivalence of the CCDFG of  $M$  with its circuit implementation; when the word-level checker is used for equivalence checking of a module that calls  $M$ , it skips the check of  $M$ , treating the CCDFG of  $M$  and its circuit implementation as equivalent black boxes.

### 3.5 TOOL IMPLEMENTATION

We first implemented the dual-rail simulation on bit-level in the Intel *Forte* environment [62], where symbolic states are represented using BDDs. However, experimental results clearly show that bit-level checking does not scale (cf. Section 3.6). Therefore, we re-implemented our equivalence checker on word-level in OCaml [58]. This is viable since word-level mappings between operations and circuit nodes are explicit. We use bit-vectors to encode the variables in the CCDFG and the circuit; the SMT engine checks input/output equivalence and determines control paths. Our word-level checker employs CVC3 SMT engine [3]. Figure 3.4 shows the framework of our equivalence checker. Behavioral synthesis generates RTL circuits in terms of Hardware Description Languages (HDLs), such as Verilog or VHDL. Currently, we only developed the parser for Verilog, but we can easily extend our HDL parser to support VHDL. Our HDL parser parses HDL files into an intermediate representation for symbolic simulation. Our HDL parser and simulator support a synthesizable subset of HDL. This subset of HDL can be synthesized into gate-level. The CCDFG parser parses CCDFG files generated by our certified compiler. The RTL and CCDFG symbolic simulators simulate circuits and CCDFGs simultaneously, synchronized by clock cycle following our dual-rail simulation scheme. The state checker check the equivalence of the outputs of symbolic simulators by utilizing SMT solvers.

Our checker provides three optimizations targeting different circuit features (see Chapter 4). Users can specify which optimizations are involved in a particular

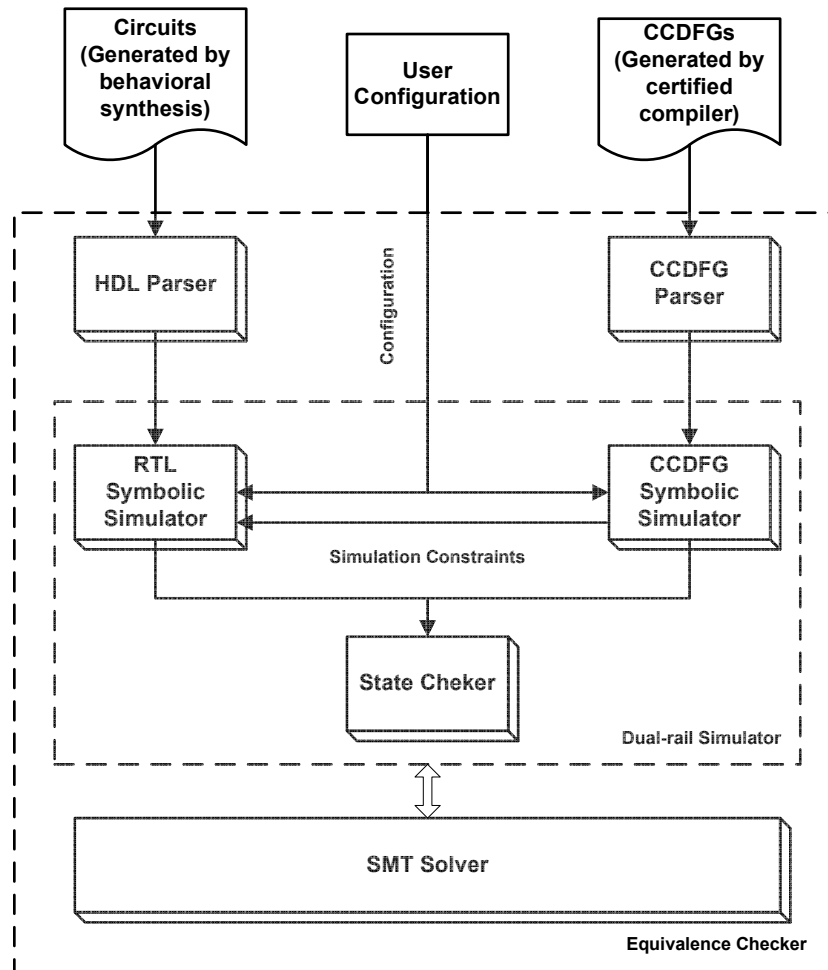


Figure 3.4: Framework of equivalence checker

Table 3.1: Bit-level equivalence checking statistics

Bit Width	# of Circuit Nodes	Time (Sec.)	BDD Nodes
2	96	0.02	503
3	164	0.05	4772
4	246	0.11	42831
5	342	0.59	16244
6	452	12.50	39968
7	576	369.31	220891
8	714	6850.56	1197604

check. The dual-rail simulator will be automatically configured according to the user’s specification.

### 3.6 EXPERIMENTAL RESULTS

To establish a baseline, we use the bit-level checker on a set of CCDFGs for GCD and the corresponding circuits synthesized by AutoESL. The experiments were conducted on a workstation with 3GHz Intel Xeon processor with 2GB memory. The checking time bound is set up to 4 hours.

Table 3.1 shows the results of bit-level SEC for GCD. GCD contains a loop whose number of iterations depends on the inputs. Since all operations are decomposed into bit-level, the running time grows exponentially with bitwidth. For 8-bit GCD, SEC takes about 2 hours. Pure bit-level SEC is thus not feasible for more complex designs.

To experiment with our word-level checking scheme, we have checked several designs which have different design features. The statistics are shown in Table 3.2. “-” signifies “out of time or memory”. DCT (Discrete Cosine Transform) is a widely used algorithm in image processing domain, which contains sequential computation

Table 3.2: Word-level equivalence checking statistics

Design	GCD	TEA	DCT	3DES
C Code Size (# of Lines)	14	12	52	325
RTL Size (# of Lines)	364	1001	688	18053
Time (Seconds)	-	-	30.1	-
Memory (Megabytes)	-	-	49.2	-

without loop. SEC for DCT only takes half a minute. Unfortunately, we cannot finish the checking for GCD, TEA, and 3DES. These designs either requires a very expensive fix-point computation or have complex modular hierarchies. Next, we present how to further optimize our checking scheme in Chapter 4.



## Chapter 4

# OPTIMIZATIONS

### 4.1 MOTIVATION AND OVERVIEW

In Chapter 3, we proposed a framework for certifying behaviorally synthesized RTL through SEC with our CCDFG representation. However, we realized that naive word-level checking ran into scalability issues. In this chapter, we present a suite of optimizations for the SEC step above, which exploit both the explicit control and data flow representations in the CCDFG and the module structures in the ESL description. We have applied these optimizations in verification of RTL synthesized by AutoESL. Our experiments show that they scale SEC to tens of thousands of lines of synthesized RTL from complex behavioral specifications (*e.g.*, unbounded loops, modules, etc.), making it viable for industrial designs. We know of no other SEC framework that can handle behaviorally synthesized RTL of such complexity.

### 4.2 CUT-POINTS

The cutpoint optimization involves pre-verifying comparison of specific CCDFG operations and their circuit implementations off-line. Subsequently, during SEC, these operations are replaced in the CCDFG and RTL by equivalent symbols. Note that only the equivalences (not computations) are relevant to SEC; if the inputs to a cutpoint are equivalent, their outputs can be replaced by equivalent symbols, causing only equivalences (not outputs themselves) to be propagated.

We utilize two types of cutpoints, *combinational* and *sequential*. Combinational cutpoints are applicable to combinational portions, and have been studied extensively [45]. RTL designs with complex combinational circuits are generated due to transformations such as loop unrolling: in the TEA example, the behavioral synthesis tool can fully unroll the *for* loop, creating complex combinational circuits by aggregating operations from different iterations. Sequential cutpoints cut sequential circuits and keep complex expressions from propagating across clock cycles.

In the TEA example (Figure 3.2), the scheduling step starting with the conditional  $pl\_start==1$  and ending with the assignment  $pl\_start=1$  is implemented as a combinational block that can be cut at all operations, *e.g.*, the one computing  $tmp54$ ; the equivalence of this operation with the corresponding RTL is certified separately (*e.g.*, by theorem proving). On the other hand, the operation that computes  $tmp49$  can be used as a sequential cutpoint since it connects two scheduling steps.

To explain the role of post-scheduling CCDFGs in cutpoint optimization, note that the ESL specification is unlocked while the RTL is clocked. Furthermore, after application of high-level transformations, the RTL has little correspondence in internal operations with the behavioral description, making it difficult to identify cutpoints. However, this problem is eliminated in our framework since there is a readily available correspondence with the post-scheduling CCDFG, *e.g.*, the operation-to-resource mapping, which provides natural candidates for cutpoints.

### 4.3 CUT-LOOP OPTIMIZATION

A major challenge in SEC is termination, which typically requires expensive fixed-point computation. Termination becomes a problem when the input description contains unbounded loops. Consider the CCDFG of the Greatest Common Divisor (GCD) algorithm shown in Figure 4.1. The bit-level symbolic simulation for GCD,

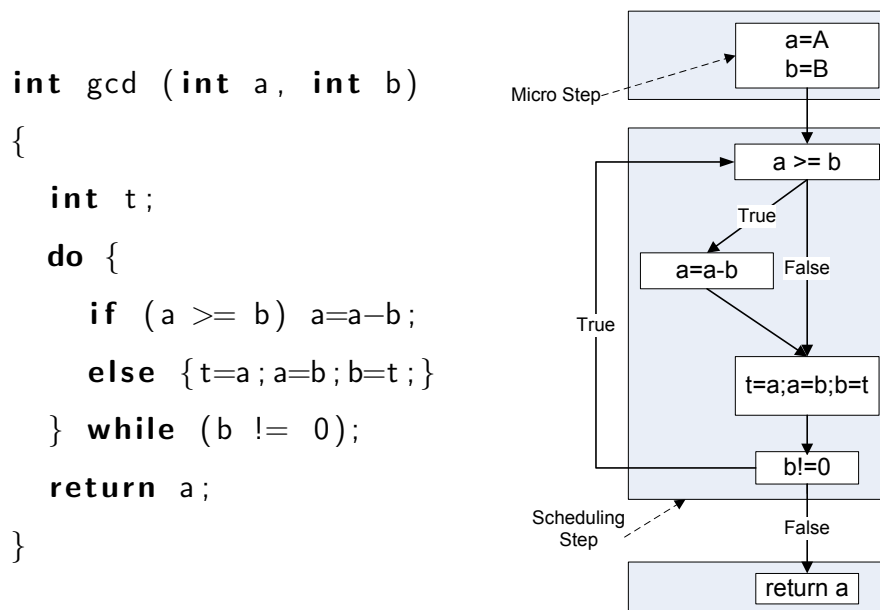


Figure 4.1: C source code and CCDFG for GCD

even for 8-bit integers, involves more than 6850 seconds and 1197606 BDD nodes (cf. Section 3.6). A naive fixed-point computation at word-level is also expensive. Even for designs with deep bounded loops (*e.g.*, TEA), full unrolling is too expensive for both bit-level and word-level simulations.

Our solution is the *cut-loop optimization*, which “cuts” the loop, reducing the fixed-point computation to three checks, *i.e.*, at the entry, body, and exit. The idea is inspired by theorem proving approaches to verifying software loops. At entry, we check equivalence between the CCDFG and the RTL for the path to the initial loop entry. For the body, we check that if (1) equivalence is maintained at the loop join point, and (2) the loop does not exit, then equivalence is maintained after one iteration. For the exit, we check that if (1) equivalence is maintained at the loop join point, and (2) the loop exits, then equivalence is maintained at the loop exit. The loop structure and entry point information are available from the synthesis tool. The checks above are inspired by inductive assertions in software verification [22, 31]: the three checks are essentially the proof obligations

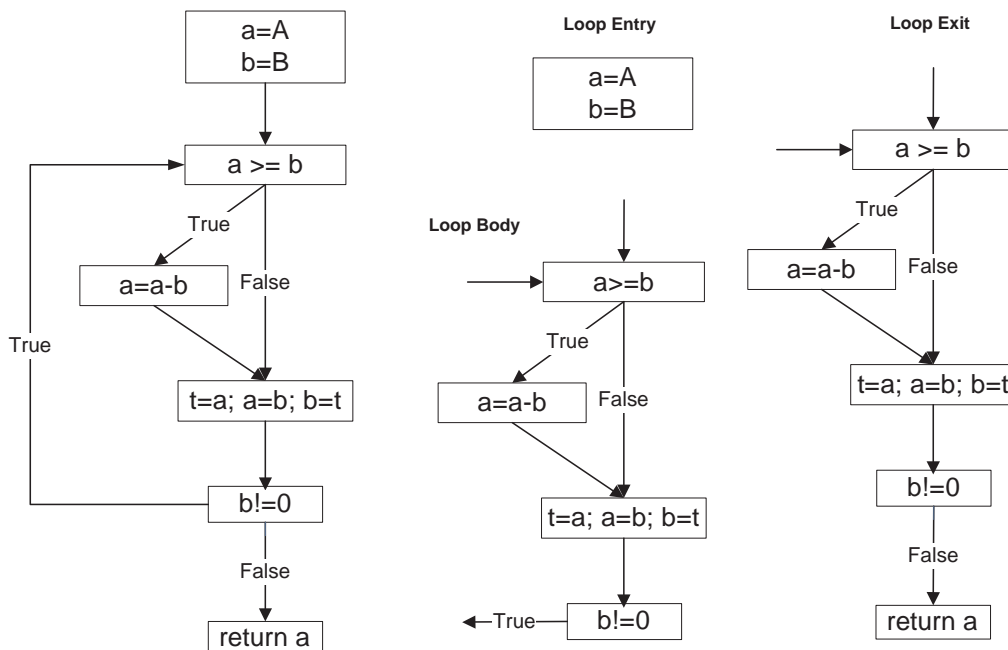


Figure 4.2: Cut-loop optimization for GCD example

discharged by a verification condition generator, if we think of equivalence with RTL as the invariant maintained by the loop. Using ACL2, we proved that the checks guarantee word-level equivalence over the entire loop execution. The proof follows a reasoning analogous to that used in justifying the use of loop invariants to cut loops for program verification using inductive assertions.

We illustrate cut-loop optimization on the GCD example in Figure 4.2. At the loop entry, the check that  $a$  and  $b$  are equivalent to their RTL counterparts is trivially true since they are inputs. For the body check the condition  $b \neq 0$  is applied to ensure the iteration does not exit, and for the exit check the condition  $b = 0$  is applied to ensure the loop exits. For both body and exit checks, the condition being checked is that if  $a$  and  $b$  are equivalent before executing  $a \geq b$  then they are equivalent after one iteration. With this optimization, word-level SEC on GCD finishes within two seconds. The cut-loop optimization is also useful

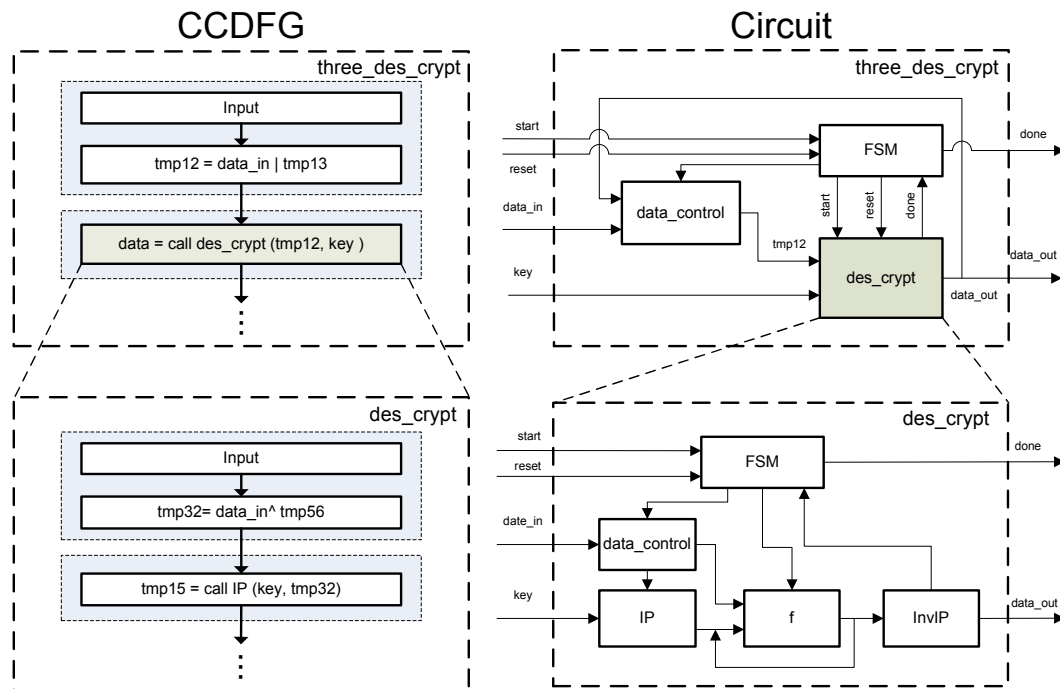


Figure 4.3: Modular SEC for 3DES

for deep bounded loops, *e.g.*, we achieved major speed-up for word-level SEC on TEA (cf. Section 4.5).

Note that loop detection is greatly simplified since CCDFGs are derived from ESL designs by applying primitive transformations.

#### 4.4 MODULAR ANALYSIS

Synthesized RTL is often large and complex, *e.g.*, for 3DES design, the behavioral synthesis tool generates 18053 lines of Verilog. Behavioral synthesis reduces RTL size via modular reuse: without modules, the RTL for 3DES would be 128K lines. Modules may be present in input description or introduced by behavioral synthesis. To support modules, CCDFGs are extended with function calls. An example function invocation in the 3DES CCDFG is shown in Figure 4.3.

With modules, a given behavioral description corresponds to several CCDFGs

(each corresponding to a module). A module can be either combinational or sequential. A combinational module returns in the same clock cycle in which it is invoked, while a sequential module takes several cycles. Note that the top-level CCDFG may not capture all the scheduling steps since some are in other sequential modules. In the synthesized RTL, there is a module for each CCDFG. In addition to RTL code implementing functionality, there is additional code for interfaces, *e.g.*, a module commonly needs `reset`, `start`, and `allow` signals besides input/output data signals.

One naive approach to handle modules is to unfold them, causing each module to be analyzed at each invocation. We prefer compositional analysis of each module separately. Our scheme works as follows.

- For each module  $M$ , the CCDFG and RTL for  $M$  are checked for equivalence separately.
- When verifying a module  $M'$  that invokes  $M$ , the invocation of  $M$  in the CCDFG and RTL of  $M'$  are replaced by equivalent uninterpreted functions.

The equivalence between function invocation in CCDFG and module interfacing mechanism in RTL is pre-certified. Modular analysis is possible because of explicit correspondence between the CCDFG and the RTL of a module: since we use the same module structure used in the synthesis, the decomposition does not introduce over-approximations.

Currently, we do not handle recursive modules since recursions in ESL descriptions are typically removed by compiler transformations; however, modular analysis can be extended to recursion by replacing the callee with a “module summary”, analogous to procedure summaries in software verification [1].

Table 4.1: Designs, features, and optimizations

Designs	Features	Optimizations
GCD	Unbounded Loop	Cut-Loop
DCT	Sequential without Loop	Cutpoint
TEA	Bounded Loop Unrolled Loop	Cut-Loop Cutpoint
DES	Bounded Loop Unrolled Loop High Sequential Complexity	Cut-Loop Cutpoint
3DES	Bounded Loop Unrolled Loop High Sequential Complexity	Cut-Loop Cutpoint Modular Analysis
3DES_key	Bounded Loop Unrolled Loop High Sequential Complexity High Combinational Complexity	Cut-Loop Cutpoint Modular Analysis

## 4.5 EXPERIMENTAL RESULTS

Table 4.1 illustrates the designs used to evaluate our optimizations. Each design is synthesized by a behavioral synthesis tool. The designs are selected carefully to exercise different facets of our framework. Encryption algorithms, *e.g.*, TEA, DES, 3DES, and 3DES\_key (3DES with key generation). DES, 3DES and 3DES\_key contains bounded loops and benefit from cut-loop; their sequential and combinational complexities also illustrate the role of cutpoints. 3DES and 3DES\_key have modular structures and modular analysis is vital to discharge their SEC. DES was deliberately synthesized without modules to further investigate the role of modular analysis. All experiments were conducted on a workstation with 3GHz Intel Xeon

Table 4.2: Word-level equivalence checking statistics

Design	RTL Size (# Lines)	Optimizations	Time (Secs)	Memory (MB)
GCD	364	NO	-	-
		CP	-	-
		CP + CL	2	4.1
DCT	688	NO	71	92.16
		CP	30.1	49.2
TEA	1001	NO	-	-
		CP	116	141.3
		CP + CL	15.6	24.6
DES	11520	NO	-	-
		CP	5896	614.4
		CP + CL	1482	426.4
3DES	18053	NO	-	-
		CP + MA	872.5	114.7
		CP + MA + CL	355.7	59.4
3DES_key	79976	NO	-	-
		CP + MA	2868.5	307.2
		CP + MA + CL	2351.7	307.2

processor with 2GB memory.

Table 4.2 shows the results of word-level SEC for all the designs from Table 4.1. Here, “-” signifies “out of time or memory”, “CP” for cutpoints, “CL” for cut-loop, and “MA” for modular analysis. The “NO” column represents “no optimizations”: it is clear that without the optimizations, SEC cannot handle long computation sequences or loops. Since DCT contains only sequential computations and no modules, cut-loop and modular analysis are not applicable; however, cutpoint optimization reduces the symbolic simulation cost to about half, in both time



and memory usage. Cutpoints, together with modular analysis, can handle long computation sequences and bounded loops, (*e.g.*, TEA, 3DES, and 3DES\_key), but blows up on fixed-point computation for unbounded loops (*e.g.*, GCD), underlining the need for cut-loop. The cut-loop optimization handles unbounded loops, while also reducing the time and memory usage for designs with bounded loops. The savings from cut-loop are relatively less for 3DES\_key since the design contains large combinational computations (for generating the key) which overshadow loop unrolling cost. The results on DES highlight the importance of modular analysis when possible: although the RTL is smaller than 3DES and 3DES\_key, the time and memory usage is higher due to lack of modules (and hence, modular analysis); for 3DES and 3DES\_key, even the behavioral synthesis tool fails without modules. The results indicate that word-level SEC with our optimizations can scale to realistic designs. Note that each of DES, 3DES, and 3DES\_key is over 10,000 lines of RTL, and 3DES\_key (even with modules) involves about 80,000 lines. We know of no other framework that can apply SEC on behaviorally synthesized RTL at this scale.

## Chapter 5

**SEC FOR SYNTHESIZED LOOP PIPELINES****5.1 MOTIVATION AND OVERVIEW**

Loop pipelining is a critical transformation in behavioral synthesis to reduce the latency of designs with loop structure by producing temporal overlap of successive loop iterations. It is available in most state-of-the-art tools, (*e.g.*, AutoESL). However, it induces retiming and out-of-order executions; furthermore, the mapping of internal operations is lost between the sequential description and the pipelined RTL. This rules out standard SEC techniques for their comparison. In particular, some key optimizations (*e.g.*, cutloop) become inapplicable.

In this chapter, we discuss the challenges with loop pipelines and present an equivalence checking approach for certifying synthesized hardware designs in the presence of loop pipelining transformations. We have applied our approach on industrial-size designs with thousands of lines of RTL, synthesized by AutoESL. This scalability is derived from tight integration with the synthesis flow. Instead of directly comparing the synthesized RTL with the sequential description, we develop an intermediate *pipeline reference model*. This model provably preserves the semantics of the sequential description. However, our model generation algorithm is parameterized by pipeline parameters, whose values are obtained from the synthesis tool; this ensures that the structure of the generated model is similar to that of the synthesized RTL, and enables internal operation mapping between the reference model and the RTL.

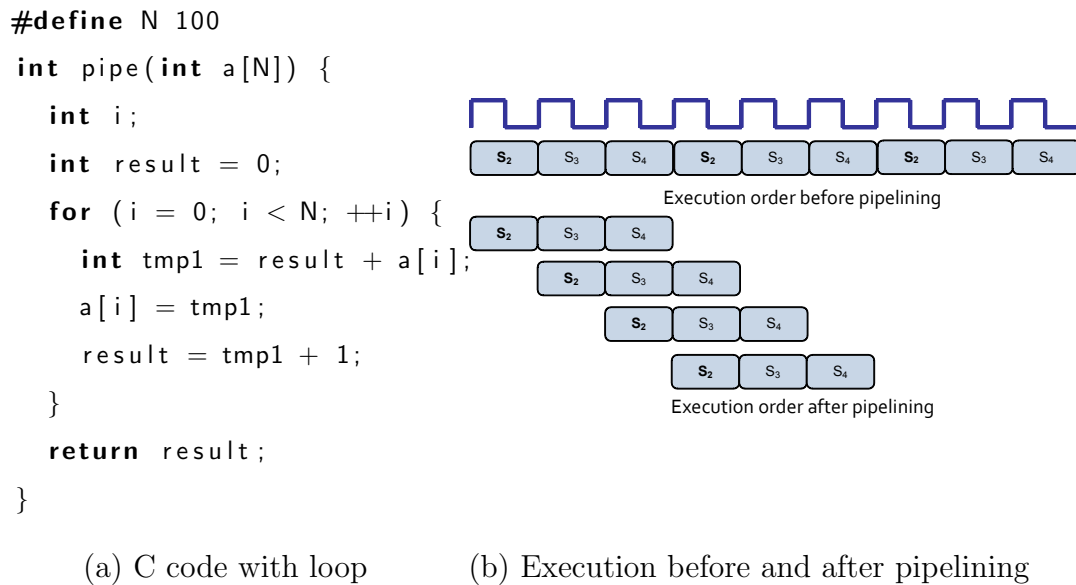


Figure 5.1: Example of loop pipeline

## 5.2 CHALLENGES WITH LOOP PIPELINES

Loop pipelining allows multiple successive iterations of a loop to operate in parallel by executing a new iteration before the previous iteration completes. Consider pipelining the loop in Figure 5.1 (a). Figure 5.1 (b) shows the execution orders of the scheduling steps in the loop body before and after pipelining. In the sequential design, execution of iteration  $i$  involves reading the value of  $a[i]$  from the memory in  $S_2$ , adding  $i$  and  $a[i]$  in  $S_3$ , and storing new value to the memory and computation of  $result$  in  $S_4$ . However, with pipelining, iteration  $i+1$  is initiated before iteration  $i$  completes.

The result of overlapping executions is a significant difference in the schedule of operations between the CCDFG of the sequential design and the RTL generated from the pipeline. Each scheduling step of the pipeline is composed of a number of scheduling steps of the sequential design; there is no longer a direct operation mapping between the CCDFG and RTL. Furthermore, due to the difference in the

execution order of the scheduling steps, the controlling finite-state machines are also different. A direct SEC between the two reduces to comparison of their input-output relations, which is prohibitively expensive for loops with many iterations.

### 5.3 SEC WITH REFERENCE MODEL

Our solution to the above problem is to develop a *reference pipelining transformation* on CCDFGs [29]. Given a CCDFG  $G$  and certain pipeline parameters (see below), we generate a new CCDFG  $G'$  by pipelining the loops. Note that our transformation is different from that used by the synthesis tool to generate the pipelined RTL. The synthesis tool transformation includes algorithms and heuristics to *determine* how many iterations to pipeline, etc.; on the other hand, our algorithm merely *takes* such information as parameters to create  $G'$ . In fact, we obtain this information from the synthesis tool itself. Thus the output CCDFG  $G'$ , if successfully generated by our algorithm,<sup>1</sup> is guaranteed to have close structural correspondence with the synthesized RTL. On the other hand, irrespective of the actual value of these parameters,  $G'$  is guaranteed to be semantically equivalent to  $G$  and can therefore be soundly used instead of  $G$  for SEC.

The following definition characterizes the loops handled by the algorithm.

**Definition 4 Pipelinable Loop.** For a CCDFG  $G \triangleq \langle G_{CD}, M, T \rangle$  and for  $T_0 \subseteq T$ , we say that  $T_0$  induces a “pipelinable loop” if (1) the projection of  $G$  on  $T_0$  is a cycle  $C$ , and (2) in the projection of  $G$  on  $T$  there is a unique node (called the “entry node”) in  $C$  with a predecessor outside  $C$  and a unique node (called the “exit node”) in  $C$  with a successor outside  $C$ .

---

<sup>1</sup>Our algorithm does not use semantic invariants of the program being transformed. Thus we may fail to pipeline a loop for a given number of iterations (and report a spurious hazard) when in fact such a pipeline is hazard-free. However, in practice we have not seen a case where the synthesis tool generates a pipeline with specific parameters and our algorithm reports a spurious hazard on those parameters.

*Remark.* The notion of pipelined loops is more restrictive than the common loop definition in programming languages. In particular, a pipelined loop has a single exit and loop nesting is disallowed. Our definition is based on the kind of loops that are actually pipelined by behavioral synthesis tools. For instance, if a design contains nested loops, then the inner loop can be unrolled completely (possibly by compiler transformations) before the outer loop can be pipelined.

---

**Algorithm 1 PIPELINELOOP**( $L = \langle S, E, M \rangle, I, N$ )

---

- 1:  $S'_1 \leftarrow \text{GenerateSchedulingSteps}(S, I, N)$
  - 2:  $\langle S'_2, M'_1 \rangle \leftarrow \text{GeneratePipelineRegs}(S'_1, M, E, I)$
  - 3:  $E'_1 \leftarrow \text{GenerateEdges}(S'_2, E, I, N)$
  - 4:  $\langle S'_3, M'_2 \rangle \leftarrow \text{GenerateForwarding}(S'_2, M'_1, E'_1, I)$
  - 5: **return**  $\langle S'_3, E'_1, M'_2 \rangle$
- 

Given CCDFG  $G$ , our reference transformation replaces each loop  $L$  in  $G$  with the pipelined refinement of  $L$  as described in Algorithm 1. Here  $I$  is iteration interval, which indicates how many clock cycles later a new iteration is to be “fed” into the pipeline, and  $N$  is the number of scheduling steps in  $L$ . Values of these parameters are readily available from behavioral synthesis. Figure 5.2 illustrates the use of the algorithm on our simple example. We now discuss the different steps of the algorithm in greater detail.

Algorithm 2 describes the construction of scheduling steps of the pipelined C-CDFG. The algorithm simulates the process of “feeding” a new loop iteration into the pipeline until the pipeline is full. Consider the sequence of iterations shown in Figure 5.3. The output is an array (initially empty) of graphs. Each graph represents the projection of the reference pipeline CCDFG at a single scheduling step. We first build the nodes of each graph in the array (Lines 3-6); we then compute the edges within each graph (Lines 8-14). The set of nodes of each graph in  $S_G$  is determined by  $I$  and  $N$ . The algorithm updates  $S_G$  for every iteration. If the

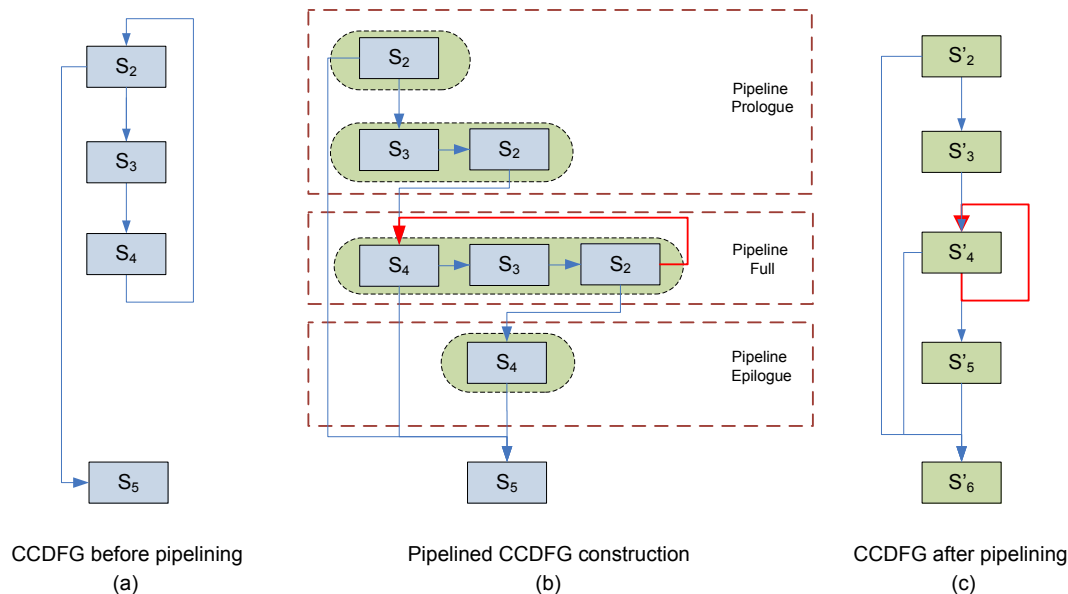


Figure 5.2: Input and output CCDFGs of loop pipelining transformation

pipeline is not yet full, *i.e.*, can accept a new iteration but no iteration is completed yet (Line 3), then a new iteration is introduced and merged with the existing iterations in the pipeline by subroutine *mergeIteration*. Subroutine *mergeIteration* merges each scheduling step in the new iteration with the corresponding steps already in pipeline, returns new scheduling steps as shown in Figure 5.3 (b), (c), (d). To model the exit, the pipeline enters the “flushing” stage in which iterations are completed without new iteration being introduced. The pipeline full stage corresponds to the new loop body for the pipelined CCDFG while the prologue and epilogue correspond to the entry and exit.

We now build the edges for each graph in  $S_G$ . The goal is to ensure that the new control flow respects that of the input loop. The process is demonstrated in Figure 5.4 (a). Recall that a scheduling step of the pipeline involves a number of scheduling steps of the original CCDFG (across several iterations). To ensure that the original control flow is respected, a scheduling step  $s'$  of the pipeline is executed

following the iteration order. This is achieved by adding edges enforcing the evaluation of microsteps from left to right. For instance, in  $S'_4$  shown in Figure 5.4 (a), an edge is created to connect  $S_4$  and  $S_3$ . Since  $S_4$  is from an earlier iteration, the direction is from  $S_4$  to  $S_3$ . The edge condition  $!exitcond$  states that loop does not exit. If the loop exits at iteration  $i$ , all iterations from  $(i+1)$  must be skipped: this is ensured by inserting the exit condition on all such edges. Subroutine *buildEdge* creates the correct edge condition according to the control flow.

---

**Algorithm 2** *GenerateSchedulingSteps* ( $S, I, N$ )

---

```

1:  $S_G \leftarrow \emptyset$ ;
2:  $iter \leftarrow 0$  /*loop iteration*/
3: while  $iter * I < N$  do
4:    $S_G \leftarrow mergeIteration(S_G, S, I, iter)$ 
5:    $iter \leftarrow iter + 1$ 
6: end while
7: /*build new edges within one single scheduling step */
8: for each step  $s'$  in  $S_G$  do
9:   for each step pair  $(s'[pos], s'[pos + 1])$  in  $s'$  do
10:     $e' \leftarrow buildEdge(s'[pos], s'[pos + 1])$ 
11:     $s' \leftarrow append(s', e')$ 
12:   end for
13: end for
14: return  $S_G$ 

```

---

Algorithm 3 inserts “pipeline registers” between iterations to facilitate correct data flow and prevent variables from being overwritten before being consumed. In a CCDFG, the effect of pipeline registers is mimicked using temporary variables as follows. We first compute all program variables that may be overwritten before being consumed (Line 2); this constitutes the variables that potentially require

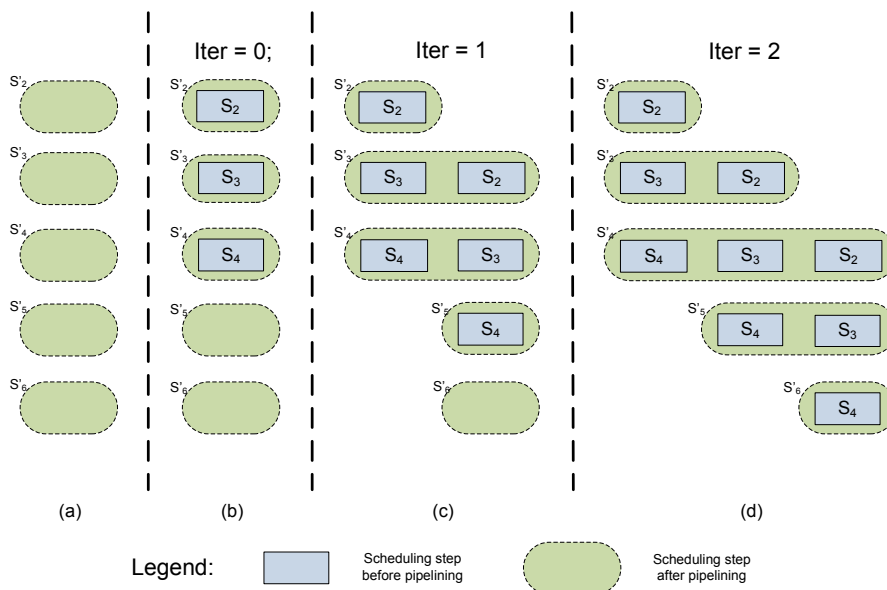


Figure 5.3: Construction of scheduling steps

pipeline registers. To find such variables, we compare the distance between the producer  $ms_p$  and the last consumer  $ms_c$ ; if the distance is greater than  $I$ ,  $v$  is assigned the new data value of the next iteration before current iteration's value has been fully consumed; this warrants insertion of pipeline variables in every scheduling step between  $ms_p$  and  $ms_c$ . The value is propagated every clock cycle following the CCDFG data flow. In Figure 5.5, variable  $\%a\_addr$  is computed in  $S_2$  and the last use scheduling step is  $S_4$ . The distance is greater than  $I = 1$ , therefore, temporary variables  $a\_addr\_pipe1$  and  $a\_addr\_pipe2$  are inserted. Subroutine *addPipelineReg* generates new microsteps for assignments of the pipeline variables, create new edges to integrate these microsteps into the data path, and updates the schedule.



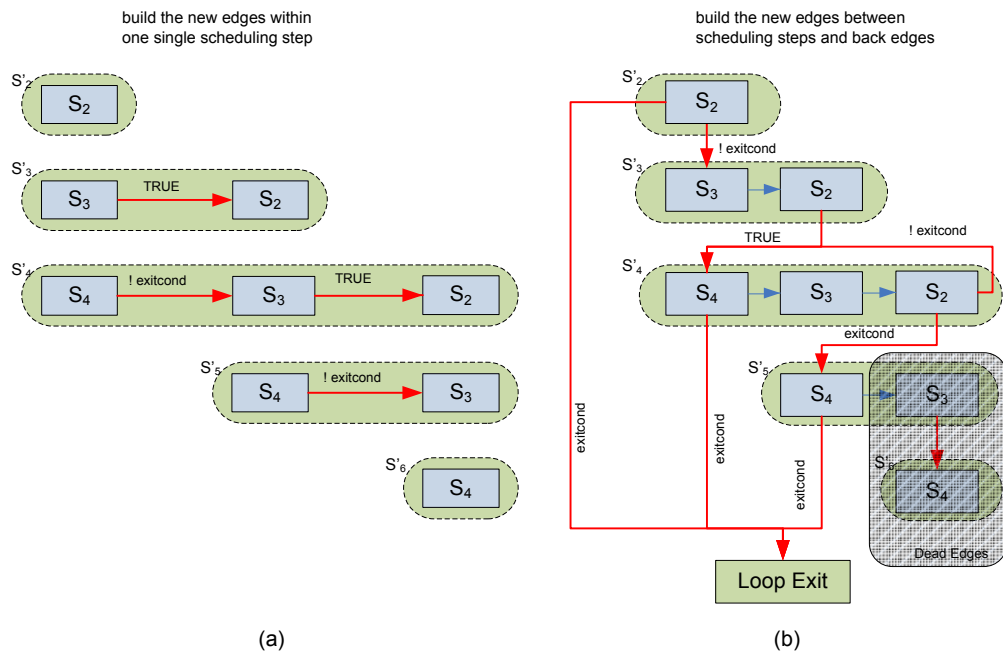


Figure 5.4: Construction of edges

---

**Algorithm 3** GeneratePipelineRegs ( $S, M, E, I$ )
 

---

- 1:  $S' \leftarrow S; M' \leftarrow M$
  - 2:  $V_{pr} \leftarrow getPipelineRegisterVars(S, M, E, I)$
  - 3: **for** each variable  $v$  in  $V_{pr}$  **do**
  - 4:    $ms_p \leftarrow getProducer(v)$
  - 5:    $ms_c \leftarrow getLastConsumer(v)$
  - 6:    $\langle S', M' \rangle \leftarrow addPipelineReg(S', M', E, ms_p, ms_c)$
  - 7: **end for**
  - 8: **return**  $\langle S', M' \rangle$
- 

Algorithm 5.3 shows the construction of edges governing the control flow of the pipelined CCDFG. Figure 5.4 (b) shows how to build edges between new scheduling steps (Lines 3-6). One example is the edge from  $S_2$  in  $S_2'$  to  $S_4$  in  $S_3'$ . Because the pipeline is still in prologue stage, the edge condition is that loop does not exit.

---

**Algorithm 4 GenerateEdges** ( $S, E, I, N$ )

---

```

1:  $E' \leftarrow \emptyset$ 
2: /*build the edges between new scheduling steps*/
3: for each step pair( $S[i], S[i + 1]$ ) in  $S$  do
4:    $e' \leftarrow buildEdge(S[i], S[i + 1])$ 
5:    $E' \leftarrow append(E', e')$ 
6: end for
7: /*build the back edge*/
8:  $s_{src} \leftarrow S[N - 1]$ ;  $s_{dst} \leftarrow S[N - I]$ 
9:  $e_{backedge} \leftarrow buildEdge(s_{src}, s_{dst})$ 
10:  $E' \leftarrow append(E', e_{backedge})$ 
11: /*build the early exit edge*/
12:  $i \leftarrow N - 1$ 
13: while  $i < sizeof(S) - 1$  do
14:    $e' \leftarrow buildEdge(S[i], s_{loopexit})$ 
15:    $E' \leftarrow append(E', e')$ 
16:    $i \leftarrow i + I$ 
17: end while
18: return  $\langle E' \rangle$ 

```

---

The back edge of the new loop connects the last scheduling step of the pipeline full stage to the first one.  $S'[N - 1]$  is the last one and  $S'[N - I]$  is the first step in the pipeline full stage. Finally, for an unbounded loop, exit can occur in any iteration. Thus, we must allow the pipeline to start flushing in any iteration, even when the pipeline is not full (Lines 12-17). In the example shown in Figure 5.4 (b), the exit point of the loop is in  $S_2$ , therefore in pipeline epilogue, the edge from  $S_4$  to  $S_3$  will never be valid. This is because the loop would have already exited and the  $S_3$  and  $S_4$  of the new iteration will not execute. The dead edges will be

removed to simplify the final CCDFG.

---

**Algorithm 5 GenerateForwarding** ( $S, M, E, I$ )

---

```

1: /*find all loop carried dependencies*/
2:  $D_{lc} \leftarrow getCarriedDependencies(S, M, E)$ 
3:  $S' \leftarrow S; M' \leftarrow M$ 
4: for each pair  $(o_w, o_r)$  in  $D_{lc}$  do
5:   if  $checkForwarding(o_r, I, S')$  then
6:      $\langle S', M' \rangle \leftarrow moveOp(o_w, o_r, S', E, M')$ 
7:   else
8:     return ERROR
9:   end if
10: end for
11: return  $\langle S', M' \rangle$ 

```

---

A critical puzzle is computation of data forwarding paths along pipeline iterations (Algorithm 5). Data forwarding is critical to achieving aggressive pipelining and eliminating data hazards. The first key observation is that forwarding is only necessary for loop carried dependencies, which extend back to the previous iteration.  $D_{lc}$  denotes a list of dependencies and Subroutine *getCarriedDependencies* finds all loop carried dependencies. Each dependency is pair of operations  $(o_w, o_r)$ ,  $o_w$  is the last write operation in the loop body and  $o_r$  is the first read operation.

Subroutine *checkForwarding* checks if the data forwarding is possible (*i.e.*, whether the value is computed before use) for these variables in the scheduling steps of the pipeline. We then implement forwarding using so-called “ $\Phi$  nodes”.  $\Phi$  nodes are special operators in compiler transformations and are widely used in resolving conditional branches in a number of compilers, and are used to postpone computation of control flow until run time. In particular, a  $\Phi$  node is introduced in a basic block which has multiple predecessors; the values of variables in a  $\Phi$  node

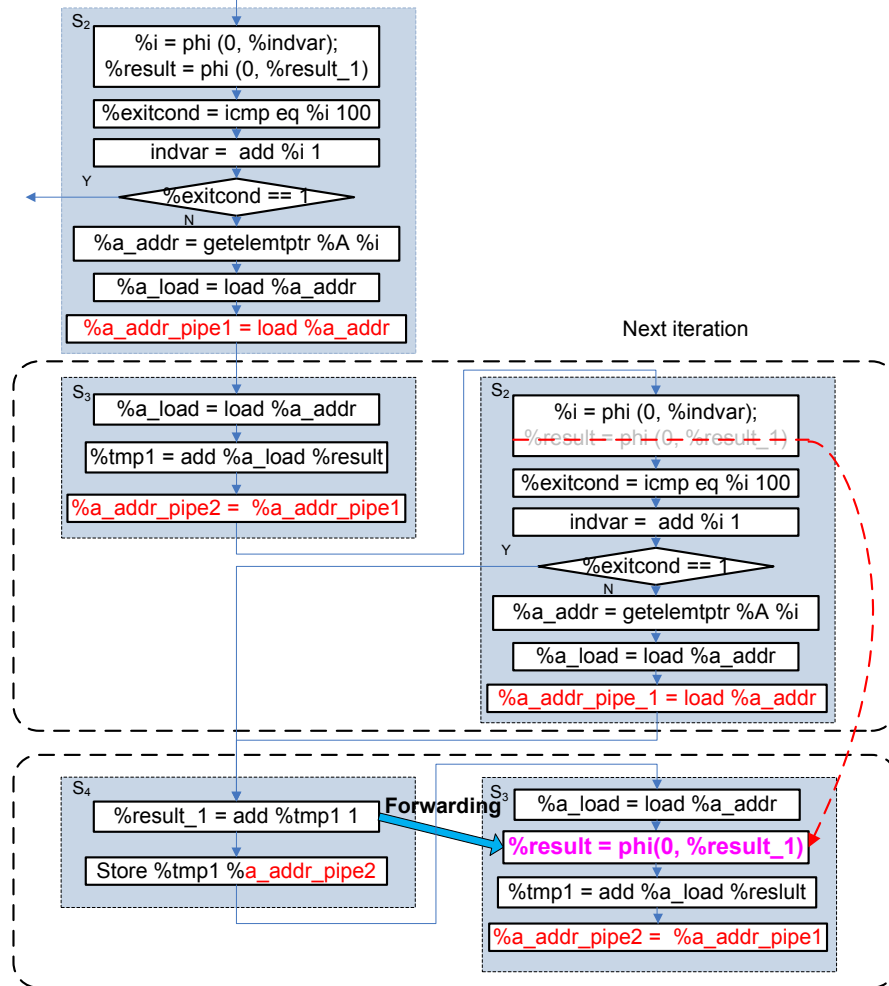


Figure 5.5: Pipeline registers and forwarding

for a specific execution are given by the specific block which actually precedes the node in that execution. To understand its utility for data forwarding, consider Figure 5.5. In the non-pipelined design  $\Phi$  operators can occur only in scheduling step  $S_2$ . The valid value of variable  $\%result$  is computed by the  $\Phi$  node in scheduling step  $S_2$ . Since we desire to execute scheduling steps  $S_2$  and  $S_3$  within a single scheduling step, we move the  $\Phi$  from  $S_2$  to  $S_3$  and forward the value directly from the producer to the consumer. In general, to implement pipeline forwarding, we need to relocate the position of the  $\Phi$  operator for a variable to immediately

before its first consumer, also update the assignment of  $\Phi$  node according to the new control flow. The “move” is implemented in *moveOp*, which will generate a new scheduling  $S'$  and a new microstep partition  $M'$ .

## 5.4 EXPERIMENTAL RESULTS

We implemented the loop pipelining algorithm on top of our certification framework for behavioral synthesis. We ran our tool on a collection of pipelined designs synthesized by AutoESL.

Table 5.1 shows the results. Our framework successfully handled SEC for synthesized designs with pipelined loops involving several thousand lines of RTL within a reasonable time and memory bounds. Note that this success on pipelines depends on the applicability of other optimizations during SEC. The reason is that because of the presence of non-trivial loops, SEC without cut-loop optimization requires an expensive fixed-point computation which runs out of memory and time. For all designs, brute-force SEC between the unpipelined CCDFG and the RTL times out. SEC between the pipelined CCDFG and the RTL can mostly finish. With the optimizations applied, SEC finishes with reduced memory and time usages. The results thus support our preference to compare the RTL with a closely resembling pipelined CCDFG that facilitates the optimizations, rather than develop a specialized SEC algorithm for pipelines.

Table 5.1: Loop pipelining experimental results

Design	RTL #line	App. Domain	Loop Info.			Pipeline Info.		Without Opt.		With Opt.	
			Inter-val	Depth	Operations	Forwarding	Pipeline Register	Mem. (MB)	Time (Sec)	Mem. (MB)	Time (Sec)
MemoryOp	291	Memory operation	1	4	18	2	2	24	38	4	0.3
TEA	383	Cryptography	1	4	28	4	2	-	-	40	6.2
XTEA	483	Cryptography	1	3	37	4	1	-	-	52	7.8
CORDIC	485	Data processing	1	3	31	4	0	38	7.9	5	0.9
SmithWater	517	Data processing	2	3	73	3	0	-	-	134	50.2
FIR	610	Signal processing	3	5	27	3	1	763	127.4	63	10.8
YUVToRGB	756	Image processing	2	6	77	1	5	-	-	335	128.9
MotionComp	1248	Image processing	1	3	53	3	0	434	132.2	50	11.4
DES	3292	Cryptography	1	3	17	2	2	468	364.7	257	163.3

## Chapter 6

**SEC FOR SYNTHESIZED FUNCTION PIPELINES****6.1 MOTIVATION AND OVERVIEW**

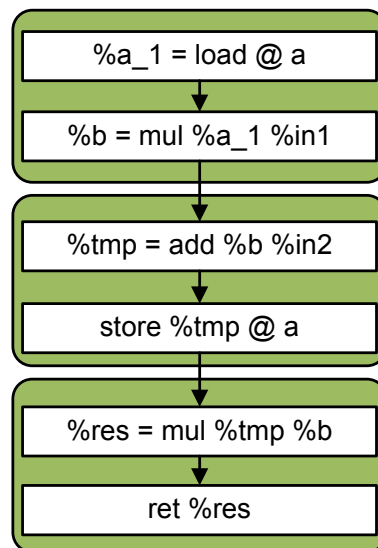
Function pipelining (a.k.a. system-level pipelining) is an important and subtle optimization. It aims to improve the quality of synthesized RTL implementations, by allowing multiple successive transactions of a function to execute concurrently. Most state-of-the-art behavioral synthesis tools, *e.g.*, AutoESL, CatapultC, and Cynthesizer, support function pipelining. However, it is a complex transformation and consequently error-prone. An error in the transformation can manifest itself as subtle bugs in scheduling, binding, or generation of controlling FSM for the synthesized design. For instance, incorrect scheduling can cause two operations to mistakenly overlap in the same clock cycle and data hazards may be introduced by incorrect pipeline forwarding. Thus, sequential equivalence checking support for verifying correctness of the synthesized pipelines is critical in enabling wide adoption of behavioral synthesis.

Function pipelining introduces overlapping execution, which leads to a significant difference between the behavioral specification and the RTL. Furthermore, typical bugs are not likely to be exposed by feeding the pipeline with one transaction: subtle corner cases typically involve the overlapped execution of multiple transactions at different pipeline stages. Therefore, SEC for function pipelines must account for all possible input sequences. Particularly, it must account for insertion of arbitrary “bubbles”, *i.e.*, pipeline stalls between the input sequences. A brute-force SEC approach directly comparing the input/output relations of the

```

int pipe(int in1 , int in2)
{
    static int a = 2;
    int b, c;
    b = a * in1;
    a = b + in2;
    c = a * b;
    return c;
}

```



(a) C Source Code of a Function (b) Corresponding CCDFG

Figure 6.1: Example of function pipeline

behavioral specification and the synthesized RTL does not scale.

We present an approach to certifying the synthesized function pipelines. Our approach is to break the certification into two steps. (1) we develop a *reference function pipelining transformation*, which takes certain pipeline parameters from behavioral synthesis to generate a pipeline reference model. (2) we check the equivalence between the reference model and the RTL implementation. Our approach efficiently reduces the complexity brought by bubbles in pipelines. The mapping between behavioral level operations and RTL functional units is still preserved, therefore some key optimizations, such as cutpoints [30], are applicable. We demonstrate the efficiency and scalability of our approach on a set of industrial-strength designs synthesized by AutoESL.



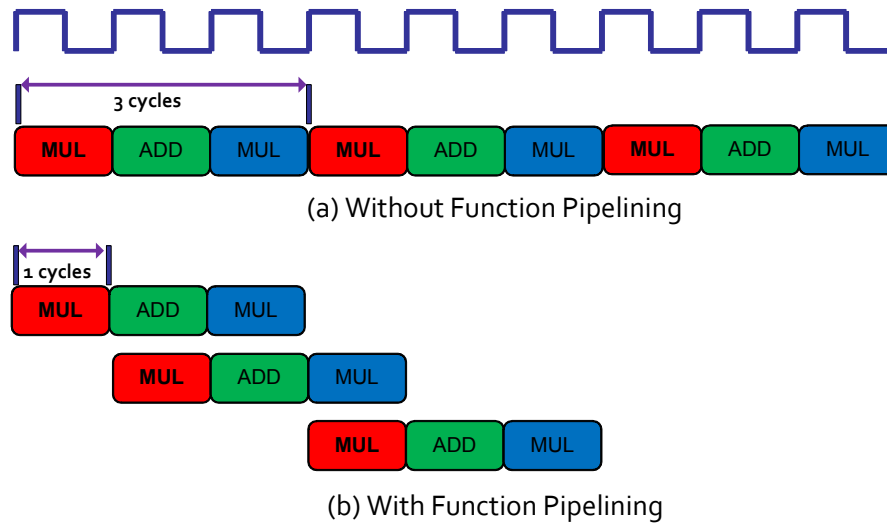


Figure 6.2: Difference between un-pipelined version and pipelined version

## 6.2 CHALLENGES WITH FUNCTION PIPELINING

Function pipelining improves the throughput of the synthesized circuit design by allowing operations from consecutive transactions of a function to execute concurrently. The pipelined function can accept a new input before the previous ones complete. Figure 6.2 compares the execution between the un-pipelined version and a pipelined version of the design shown in Figure 6.1. There are three operations: two *multiplies* and one *add*. Without function pipelining, the circuit can accept a new input every three clock cycles. However, with function pipelining it can accept a new input every clock cycle: thus, function pipelining can dramatically improve the circuit throughput. However, the resource usage may increase, since the two *multiply* operations cannot share the same multiplier now.

Behavioral synthesis generates handshake signals to implement the synchronization between the synthesized pipeline and its surrounding circuits [60]: these signals include *start*, *done* and *allow*, as shown in Figure 6.3. The *start* signal indicates that there are valid inputs ready for execution in the pipeline and the

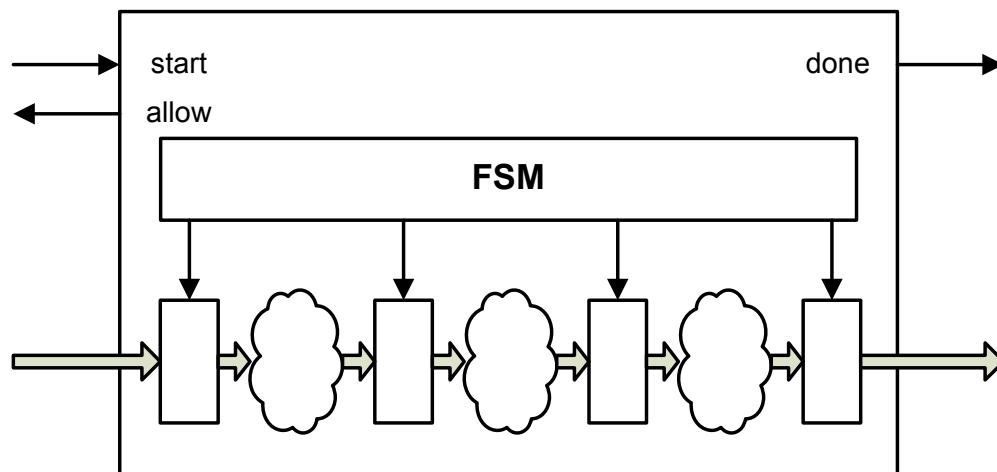


Figure 6.3: Hardware interface

*allow* signal indicates that the pipeline is ready to start a new transaction in the next clock cycle. The handshake happens when both *start* and *allow* are high. The *done* signal indicates that the pipeline produced some valid output data. However, when the pipeline is ready to accept a new input (*i.e.*, when *allow* is high), the upstream circuit may not be able to get the new input data ready (*i.e.*, *start* is low); in this case, a bubble is inserted into the pipeline. For instance, consider the pipeline shown in Figure 6.2 (b). The pipeline can start a new transaction every cycle. However, since there is no input at the third cycle and a new input comes at the fourth cycle instead, one bubble is inserted into the pipeline. When there are bubbles in the pipeline, the pipeline typically disables the corresponding functional units to save power. Correctly disabling the idle functional unit without effecting the rest of the pipeline is challenging and error-prone. Therefore, equivalence checking of function pipeline must carefully take bubbles into account.

In addition to bubbles, complexity of SEC in function pipelining comes from overlapping execution of multiple transactions. It leads to a significant difference in the schedule of operations between the CCDFG of the sequential design and the RTL implementing the function pipeline. Furthermore, overlapping execution

leads to a different FSM. Function pipelines may have fewer states, but each state executes more operations. Thus, standard SEC techniques are not effective.

Our top-level approach for function pipeline verification has analogues to previous SEC approach for loop pipelines [29], *viz.*, developing a reference model for the pipelined CCDFG that is semantically equivalent to the sequential design and can be used for SEC with the RTL. However, the reference model generation for function pipelines is inherently different from loops and involves subtle challenges not encountered in loop pipelines, leading to drastically different algorithms. A major difference between loop and function pipelines is that function pipelines must account for arbitrary bubbles due to non-determinism in function invocation latency, but loop pipelines do not. In loop pipelines, an FSM controls when to start a new loop iteration. The FSM is part of the synthesized RTL, and this fixes the execution of loop iterations. For function pipelines, starting a new transaction is determined by upstream circuits, and is a runtime decision. To fully certify the pipeline, all bubble insertion scenarios must be accounted for. A naive approach is to build one pipelined CCDFG for each such scenario. Therefore, to cover all these scenarios, we have to construct many pipelined CCDFGs. Figure 6.4 (a) shows the CCDFG before pipelining, which has three scheduling steps. Figure 6.4 (b), (c), (d), (e) show the pipelined CCDFGs with different numbers of bubbles inserted in different stages. To certify a function pipeline, we have to apply SEC between all possible pipelined CCDFGs and the synthesized RTL implementation. Unfortunately, the number of such pipelined CCDFGs is exponential in the number of scheduling steps of the CCDFG before pipelining, making this approach impractical.

### 6.3 SEC FOR FUNCTION PIPELINING

Our approach entails building a pipelined reference model, while still avoiding the exponential cost due to bubble insertion mentioned above.

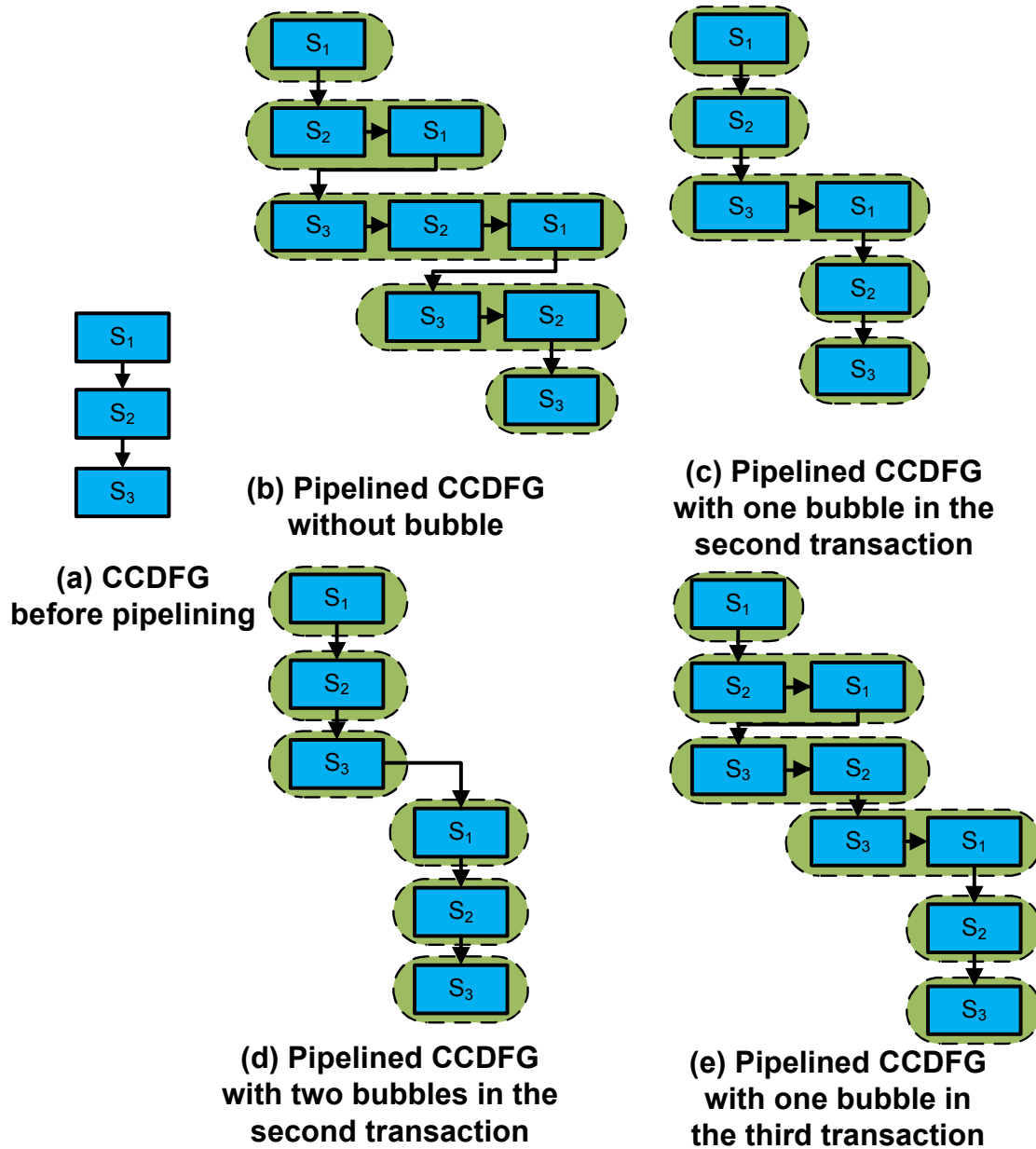


Figure 6.4: Pipelined CCDFGs for different bubble insertion scenarios

Our function pipelining transform algorithm takes a CCDFG before pipelining  $G$  and certain pipeline parameters to generate a functional pipelined CCDFG  $G'$ . Checking the equivalence between CCDFG  $G$  and RTL is equivalently translated to equivalence checking between pipelined CCDFG  $G'$  and RTL. CCDFG  $G'$  allows operations to execute in parallel, closely corresponding with the RTL through careful modeling of bubble insertion. Thus, we can leverage the existing SEC approach to check CCDFG  $G'$  and RTL.

We focus on the pipelines which satisfy the following requirements:

- All sub-functions have been fully inlined.
- All loops have been fully unrolled.
- No global variables (other than static variables).

Our framework actually supports loops and sub-functions by extending the approach discussed here with compositional reasoning; we do not discuss that extension in the dissertation. Global variables can be avoided by explicitly rewriting as static variables plus corresponding interfaces.

### 6.3.1 Algorithm to build Reference Model

As a pedagogical simplification, assume first there is no branch among scheduling steps, but allow branches inside scheduling steps. Note that if CCDFG  $G$  has branches, we can merge the destination scheduling steps into one single scheduling step; thus the branch between scheduling steps is equivalently converted into a branch inside a scheduling step. Without considering branches, we can view CCDFG  $G$  as a sequence of scheduling steps from the entry step to the exit step. *Task Interval* is an important metric to measure the performance of function pipelines: it is the number of clock cycles that must elapse between two transactions. We can partition CCDFG  $G$  into a sequence of sub-CCDFGs according

to task interval  $I$ . Each sub-CCDFG is called a pipeline unit, which is defined in Definition 5. All scheduling steps within one pipeline unit execute sequentially, and different pipeline units can execute in parallel. In the example shown in Figure 6.2 (b), because the pipeline can start a new transaction every clock cycle, each scheduling step is a pipeline unit.

**Definition 5 Pipeline Unit.** Given a pipeline task interval  $I$  and a CCDFG  $G \triangleq \langle G_{CD}, M, T \rangle$ ,  $T$  can be partitioned into a set of sub-schedule  $\{T_0, T_1, \dots, T_n\}$ . Each  $T_i$  takes  $I$  clock cycles (except possibly the last partitioned schedule  $T_n$  which may be less than  $I$ ). Therefore,  $G$  can be partitioned into a set of sub-CCDFGs  $\{G_0, G_1, \dots, G_n\}$ , respectively.  $G_i \triangleq \langle G_{CD}, M, T_i \rangle$  is called a pipeline unit.

---

**Algorithm 6 GeneratePipeUnits** ( $G = \langle S, E, M \rangle, I, N$ )

---

```

1:  $P \leftarrow \emptyset$ ;  $i \leftarrow 0$ 
2: while  $i \leq N$  do
3:    $S' \leftarrow \emptyset$ ;  $pos \leftarrow 0$ 
4:   while  $i + pos \leq N$  do
5:      $s \leftarrow S[i + pos]$ 
6:      $S' \leftarrow S' \cup s$ ;  $pos \leftarrow pos + 1$ 
7:   end while
8:    $p \leftarrow buildPipelineUnit(S', E, M)$ 
9:    $P \leftarrow P \cup p$ ;  $i \leftarrow i + I$ 
10: end while
11: return  $P$ 

```

---

Algorithm 6 describes the process of partitioning CCDFG  $G$  into a set of pipeline units  $P$ . Here,  $G$  is described as the triple  $\langle S, E, M \rangle$  where  $S$  and  $E$  denote the nodes and edges of the projection of  $G$  on  $T$ , and  $M$  is the set of microstep partitions refined by  $T$ . Here  $I$  is the task interval, and  $N$  is the number of

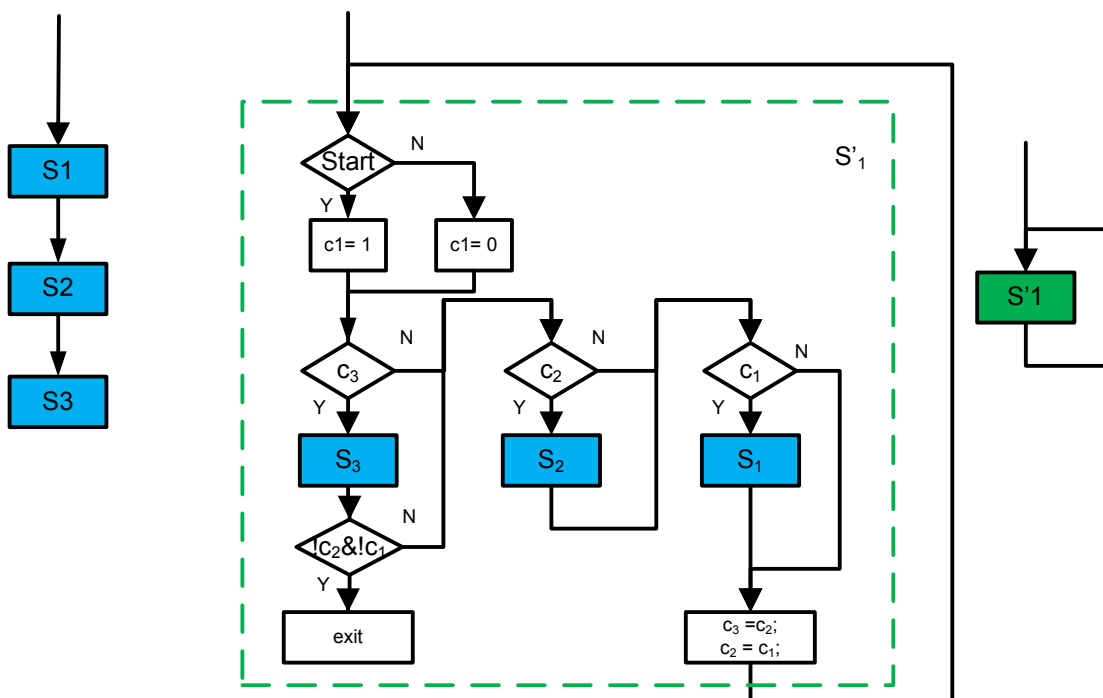


Figure 6.5: Input and output CCDFGs of function pipelining transformation

scheduling steps in  $G$ , which is same as pipeline's latency. We use  $s_k = S[k]$  to represent the  $i$ -th scheduling step in CCDFG  $G$ . The algorithm works by traversing  $G$  from the entry step  $s_0$ , and creating one pipeline unit  $p$  for each group of  $I$  consecutive scheduling steps for one pipeline unit. Subroutine *buildPipelineUnit* creates a pipeline unit  $p$ ; this process proceeds until we finish the traversal.

Algorithm 7 shows the sequence of high-level steps steps involved in generating pipelining reference model. It takes CCDFG  $G$ , task interval  $I$ , and the number of scheduling steps  $N$ . It involves five steps, *viz.*, (1) inserting pipeline registers, (2) constructing new scheduling steps, (3) generating new control edges, (4) restricting control and data flow through guard variables, and (5) implementing data forwarding. We describe these steps in detail below. Figure 6.5 illustrates the result of applying these steps for our simple example of Figure 6.4(a). The

CCDFG on the left is the one before pipelining, the CCDFG on the right is the generated pipelined reference model, and the figure in the middle shows how the scheduling steps of the pipeline correspond to those in the original.

---

**Algorithm 7**  $\text{buildPipeline}(G = \langle S, E, M \rangle, I, N)$

---

```

1: /*first, generate pipeline register*/
2:  $\langle S'_1, M'_1 \rangle \leftarrow \text{GeneratePipelineRegs}(S, M, E, I)$ 
3: /*second, build pipelined scheduling steps*/
4:  $S'_2 \leftarrow \text{GenerateFuncSchedulingSteps}(S'_1, I, N)$ 
5: /*third, generate new control graph edges */
6:  $E'_1 \leftarrow \text{GenerateFuncEdges}(S'_2, E, I, N)$ 
7: /*fourth, insert pipeline guard variables*/
8:  $\langle S'_3, M'_2, E'_2 \rangle \leftarrow \text{GenerateGuardCond}(S'_2, M'_1, E'_1, I)$ 
9: /*fifth, generate forwarding*/
10:  $\langle S'_4, M'_2 \rangle \leftarrow \text{GenerateFuncForwarding}(S'_3, M'_2, E'_2, I)$ 
11: return  $G' = \langle S'_4, E'_2, M'_2 \rangle$ 

```

---

**Inserting Pipeline Registers.** Since the pipeline can accept new inputs before the previous one finishes, it may need extra registers to store the intermediate value to prevent variables from being overwritten. Algorithm 8 describes how the pipelined registers are introduced in the pipelined CCDFG. The basic idea is to insert temporary variables to mimic pipeline registers. Subroutine *getAllVars* returns all variables in CCDFG  $G$ . Subroutine *needPipelineReg* checks the necessity of all variables by comparing the life time  $l_v$  for each variable  $v$  with  $I$ . The life time of a variable is the distance between its producer  $ms_p$  and the last consumer  $ms_c$ . If  $l_v$  is greater than  $I$ , pipeline registers for  $v$  are required, otherwise, not necessary. Equivalently, if  $ms_p$  and  $ms_c$  belongs to two different pipeline units and these two are not consecutive, a pipeline register is required. Subroutine *addPipelineReg* creates pipeline register variables and propagates the value from



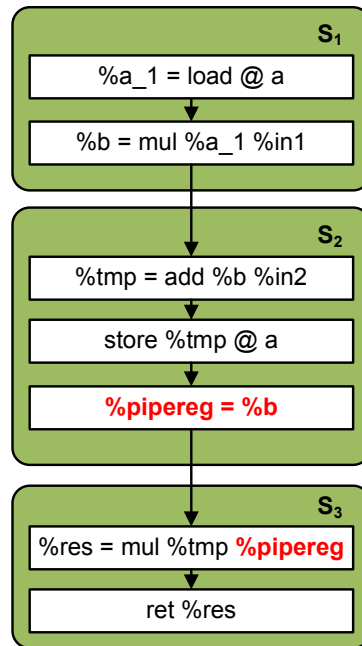


Figure 6.6: Generate pipeline registers

$ms_p$  and  $ms_c$  along pipeline register variables. The number of pipeline registers required is determined by how many pipeline units exist between  $ms_p$  and  $ms_c$ . Figure 6.6 shows a pipeline register inserted for variable  $b$ . The producer is in the first scheduling step, and the consumer is in the third step; a pipeline register is required in the middle step to prevent it from being overwritten after pipelining.

---

**Algorithm 8 GenerateFuncPipelineRegs** ( $S, M, E, I$ )

---

```

1:  $V \leftarrow \text{getAllVars}(M)$ 
2: for each variable  $v$  in  $V$  do
3:    $ms_p \leftarrow \text{getProducer}(v)$ 
4:    $ms_c \leftarrow \text{getLastComsumer}(v)$ 
5:   if  $\text{needPipelineReg}(ms_p, ms_c, I)$  then
6:      $\langle S', M' \rangle \leftarrow \text{addPipelineReg}(S', M', E, ms_p, ms_c)$ 
7:   end if
8: end for
9: return  $\langle S', M' \rangle$ 

```

---

**Constructing Scheduling Steps.** In the pipelined CCDFG  $G'$ , a scheduling step  $s'$  consists of multiple scheduling steps of CCDFG  $G$ . All steps in  $s'$  can execute and finish within one clock cycle. A key step for constructing scheduling steps for pipelined CCDFG is to correctly group scheduling steps from CCDFG  $G$ . The grouping result, according to the pipeline parameters provided by behavioral synthesis, should match the behavior of the synthesized pipeline. To achieve this, for the  $i$ th scheduling step  $s'$  in  $G'$ , we collect the  $i$ th scheduling step from all pipeline units. Scheduling step  $s'$  then must maintain the following two properties: (1) Let  $\alpha$  and  $\beta$  be any two scheduling steps in  $G$  collected to execute in  $s'$ ; then  $\alpha$  and  $\beta$  must belong to different pipeline units. (2) Every pipeline unit (except possibly the last) must have some scheduling step in  $s'$ . Algorithm 9 shows our approach to construct scheduling steps. Subroutine *getPipeUnits* returns pipeline units generated by *generatePipeUnit*. Lines 6-10 collect scheduling steps from pipeline units. We then generate control/data edges between those steps for scheduling step  $s'$  as shown in line 14-19. The edge is from left to right, because the scheduling steps in left are running the transaction entered the pipeline early. Subroutine *buildEdge*

creates the edges between two scheduling steps and subroutine *appendEdge* create a new scheduling step which includes edge  $e$ . Figure 6.7 show the pipelined scheduling steps for the simple example. In this example,  $I$  equals to one, therefore there is only one scheduling step in  $G'$  and this scheduling step consists of all scheduling steps before pipelining.

---

**Algorithm 9 GenerateFuncSchedulingSteps** ( $S, I, N$ )

---

```

1:  $P \leftarrow \text{getPipeUnits}()$ ;  $S' \leftarrow \emptyset$ 
2: /*collect scheduling steps from pipeline units*/
3: for each  $i$  in  $I$  do
4:    $s'_i \leftarrow \emptyset$ 
5:   for each  $p$  in  $P$  do
6:     if  $\text{length}(p) \geq i$  then
7:        $s'_i \leftarrow s'_i \cup p[i]$ 
8:     end if
9:   end for
10:   $S' \leftarrow S' \cup s'_i$ 
11: end for
12: /*build new edges within one single scheduling step */
13: for each step  $s'$  in  $S'$  do
14:   for each consecutive step pair  $(s'[k], s'[k + 1])$  in  $s'$  do
15:      $e' \leftarrow \text{buildEdge}(s'[k], s'[k + 1])$ 
16:      $s' \leftarrow \text{appendEdge}(s', e')$ 
17:   end for
18: end for
19: return  $S'$ 

```

---

**Building Edges.** Algorithm 10 shows the construction of edges governing the control flow of the pipelined CCDFG. Lines 3-6 show the construction of edges

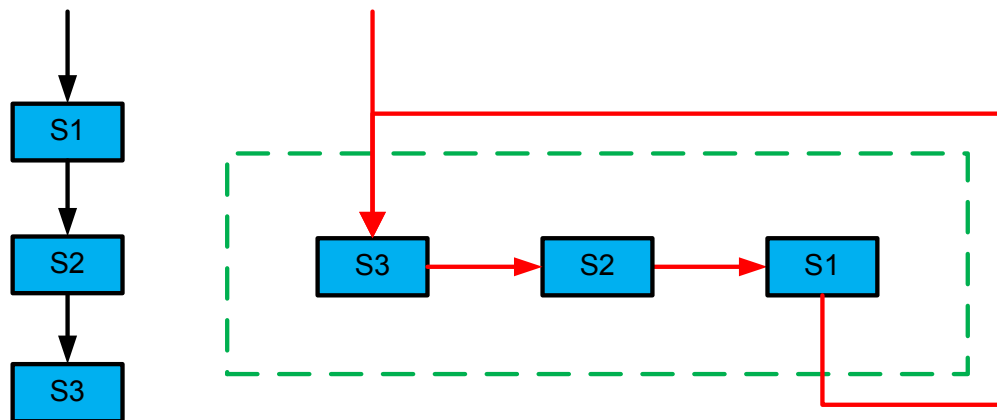


Figure 6.7: Construction of scheduling steps and edges

between scheduling steps of the pipelined CCDFG  $G'$ . Besides, a back edge is generated from the last scheduling step to the first scheduling step. The pipelined CCDFG  $G'$  is formed as a loop. Figure 6.7 shows the edges between scheduling steps in CCDFG  $G'$ .

---

**Algorithm 10** `GenerateFuncEdges` ( $S', E, I, N$ )

---

```

1:  $E' \leftarrow \emptyset$ 
2: /*build the edges between new scheduling steps*/
3: for each consecutive step pair( $S'[i], S'[i + 1]$ ) in  $S'$  do
4:    $e' \leftarrow buildEdge(S'[i], S'[i + 1])$ 
5:    $E' \leftarrow E' \cup e'$ 
6: end for
7: /*build the back edge*/
8:  $s_{src} \leftarrow S'[I - 1]$ ;  $s_{dst} \leftarrow S'[0]$ 
9:  $e_{backedge} \leftarrow buildEdge(s_{src}, s_{dst})$ 
10:  $E' \leftarrow E' \cup e_{backedge}$ 
11: return  $E'$ 

```

---

**Generating Guard Variables.** Note that the pipelined CCDFG  $G'$  must be a

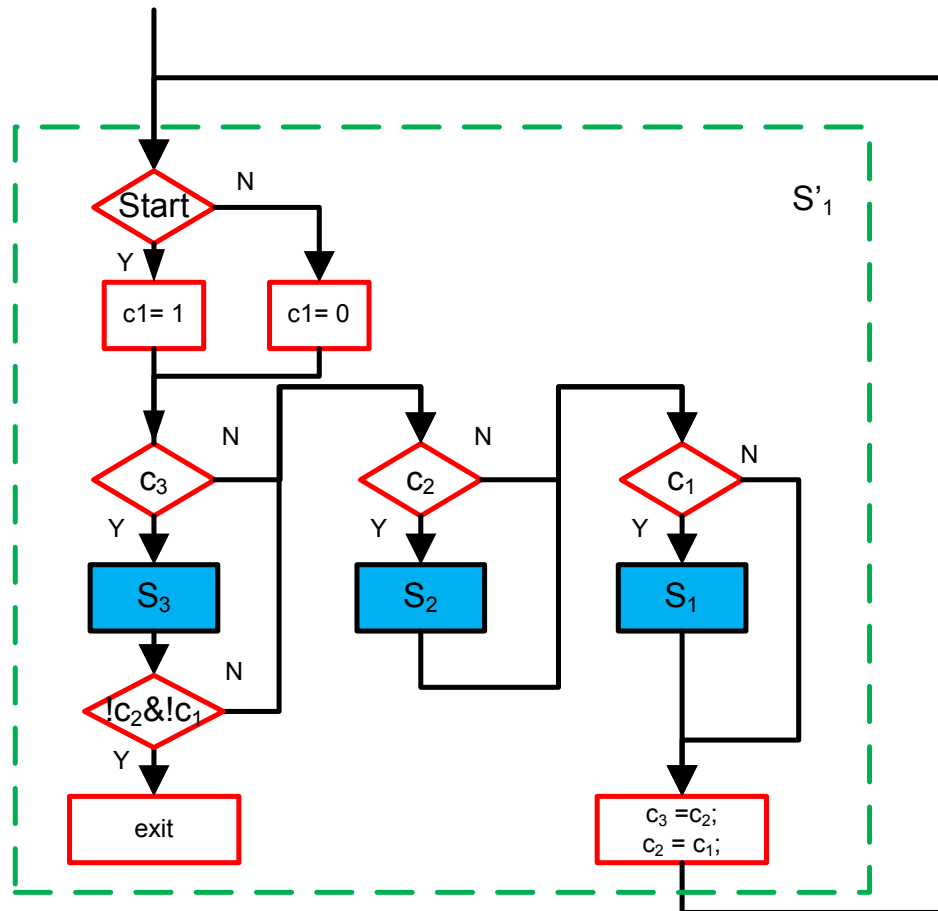


Figure 6.8: Insert guard variables and assignment

loop since it must be able to initiate an arbitrary number of function invocations as determined by the upstream logic. Guard variables guarantee that the execution of this loop corresponding to each function invocation follows the control flow of the original CCDFG  $G$  and terminates properly. Algorithm 11 describes details of guard variable insertion, and Figure 6.8 illustrates it with a simple example. First, subroutine *createGuardVariable* creates guard variables  $c_1, c_2, \dots, c_n$  for all pipeline units. For each scheduling step  $s$  in CCDFG  $G$ , subroutine *insertGuard* inserts a branch operation before entering it. If the guard variable is *true*, this scheduling step is enabled, otherwise it is skipped. After executing one pipeline

unit, we propagate the value of the guard variable to its successor in the sequence. Recall from above that  $G'$  is a loop; One loop iteration executes and finishes all pipeline units simultaneously. The assignment of first guard variable  $c_1$  depends on *start* signal. Guard variables are propagated right before the back edge. The assignment and propagation of guard variables are constructed by subroutine *genVarAssign*. Figure 6.8 shows the guard variables in an example. We need to specially handle the exit of pipelined CCDFG. CCDFG  $G'$  can only exit when  $c_n$  is *true* and  $c_1, \dots, c_{n-1}$  all are *false*, which indicates the pipeline only has one last transaction running and this transaction is going to exit. In pipelined CCDFG, we refine the semantics of *ret* operation. Operation *ret* defines the end of one transaction instead of whole CCDFG, and generates an output if it is not returning void. We introduce a new operation *exit* to denote the termination of the pipelined CCDFG, which is gated by guarded variables. Subroutine *insertExitOp* inserts this gated *exit* operation.

Note that the pipelined CCDFG must permit overlapped execution of all the pipeline stages. If *start* asserts, the first guard variable  $c_1$  is assigned *true* when entering the loop in the pipelined CCDFG. During the execution of the first iteration of the loop body, only those scheduling steps guarded by  $c_1$  are enabled. The other steps are skipped. Consider the example shown in Figure 6.8. In the first loop iteration,  $s_1$  is enabled and  $s_2$  and  $s_3$  are disabled. At the end of the first loop iteration, guard variable  $c_2$  receives the enable token propagated from  $c_1$ ,  $c_3$  remain *false*. In the second iteration,  $c_1$  still remains *true*, because there is a second start request. Both  $s_1$  and  $s_2$  are enabled in the second iteration. This process proceeds until all guard variables  $c_1, \dots, c_3$  are *true*, pipeline enters a pipeline full stage. In pipeline full stage, when a new transaction is started, one early transaction finishes at the same time. When pipeline is in full stage and there is no *start* signal any more, the pipeline starts to flush. Guard variable  $c_1$  is assigned to *false* due to no

start. Thus,  $s_2$  and  $s_3$  are enable and  $s_1$  is disable. The disable token is propagated between guard variables every loop iteration. If all guard variables are *false*, except the last one  $c_3$ , the pipelined CCDFG finishes the execution by executing *exit* operation. We can easily insert bubbles into pipelines by toggling *start* signal.

---

**Algorithm 11 GenerateGuardCond** ( $S, E, M$ )

---

```

1:  $S' \leftarrow S; \quad E' \leftarrow E; \quad M' \leftarrow M$ 
2:  $P = \text{getPipeUnits}()$ 
3:  $C = \text{createGuardVariable}(P)$ 
4: /*generate guard condition for each scheduling steps*/
5: for each pipeline unit  $p$  in  $P$  do
6:   for each scheduling step  $s$  in  $p$  do
7:      $c \leftarrow \text{getGuardVar}(p)$ 
8:      $\langle S', M', E' \rangle = \text{insertGuard}(s, p, C, S', E', M')$ 
9:   end for
10: end for
11: /*generate assignment for guard variables*/
12:  $\langle S', M', E' \rangle \leftarrow \text{genVarAssign}(C, S', M', E')$ 
13: /*insert exit operation*/
14:  $\langle S', E' \rangle \leftarrow \text{insertExitOp}(ms_{ret}, C, S', E')$ 
15: return  $\langle S', M', E' \rangle$ 

```

---

**Implementing Pipeline Forwarding.** The last step in the construction is to implement pipeline forwarding. In function pipelines, the dependencies between transactions are introduced by global or static variables. The forwarding can be implicitly implemented by mapping static/global variables to hardware registers which form feedback paths. In CCDFG, the operations to fetch or store data to registers are represented by *load* and *store*, respectively. Algorithm 12 describes details about constructing forwarding for a pipelined CCDFG. Subroutine

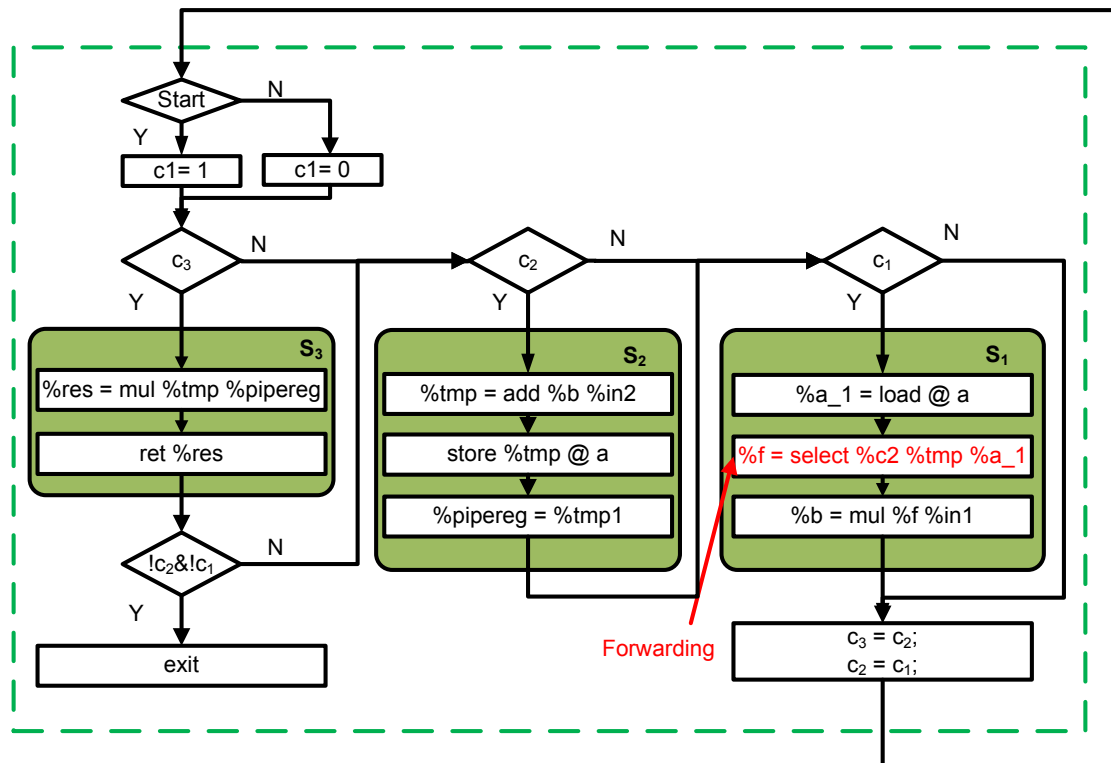


Figure 6.9: Final pipelined CCDFG

*findAllForwarding* returns all pairs of operations which may need forwarding by checking *load* and *store* pairs. However, in order to achieve the best performance, behavioral synthesis may generate a combinational path to forward the data directly. For instance, in the example shown in Figure 6.10, the output of *adder* has been directly forwarded to the next transaction’s multiplier without passing through the register, otherwise the synthesized RTL cannot accept new data every clock cycle. To mimic this combinational path, we make sure there is a valid data path from the forwarding source operation to the destination in the pipelined CCDFG. Absence of such a path is an indication of data hazard. Our checking reports errors. However, the forwarding path may vary depending on bubbles in the pipeline. If the *adder* is disabled due to bubbles, the data forwarded to the *multiplier* by the combinational path is invalid. Instead, the correct data should



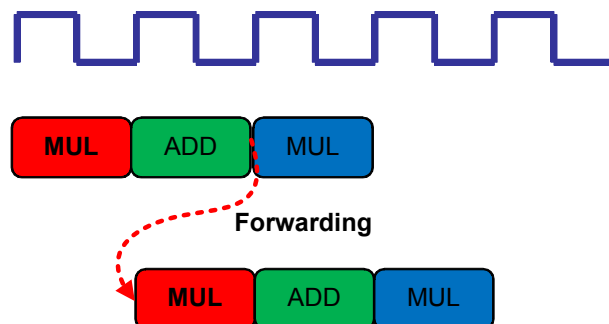


Figure 6.10: Waveform of pipeline forwarding

come from the register. To handle this complication, we determine the forwarding path by checking whether the source operation is enabled. This check can be done by checking its guard variable. We insert a *select* operation to implement the check. Figure 6.9 shows the details about the forwarding between *adder* and *multiplier*.

---

**Algorithm 12** `GenerateFuncForwarding` ( $S, M, E, I$ )

---

```

1:  $D_{lc} \leftarrow findAllForwarding(S, M, E)$ 
2:  $S' \leftarrow S; M' \leftarrow M$ 
3: for each pair  $(o_w, o_r)$  in  $D_{lc}$  do
4:   if  $checkForwarding(o_r, I, S')$  then
5:      $\langle S', M'E' \rangle \leftarrow insertSelect(o_w, o_r, S', E, M')$ 
6:   else
7:     return ERROR
8:   end if
9: end for
10: return  $\langle S', M' \rangle$ 

```

---

### 6.3.2 SEC between CCDFGs and the RTL

Recall from Section 6.2 that handling bubbles is a major hurdle for certifying pipelines. Bubbles in pipelines affect the behavior: (1) the idle operations are disabled; (2) the pipeline forwarding has different paths. These two are modeled in pipelined CCDFG by introducing guard variables. Recall however, that in order to fully check the behavior of pipelines with bubbles, we need to run SEC on all input sequences combinations.

We implement an approach which only runs the check once. We utilize our guard variables to encode all possible input combinations. This introduces the proof obligation that the execution of the transactions already in pipeline do not affect the execution of a new transaction. A new transaction can start at an arbitrary state, with the the pipeline full, empty, or containing bubbles. We model the pipeline at different states by toggling guard variables. For instance, the execution shown in Figure 6.4 (c) can be modeled as assigning  $c_1 = true$ ,  $c_2 = false$ , and  $c_3 = true$ .

Our SEC then has the following three steps:

- Set the pipelined CCDFG to be a symbolic state which starts a new transaction. Setting the pipeline at arbitrary state is done by encoding guard variables. We set  $c_1$  to *true* and assign symbols to  $c_2, \dots, c_n$ ;
- Set the FSM of RTL circuit to the same symbolic state as the CCDFG. The structure of circuit's FSM can be obtained from reports obtained from behavioral synthesis; we analyze these reports to determine the corresponding symbolic states for CCDFG and RTL.
- Feed the same input symbolic data set on CCDFG and RTL, then run dual-rail symbolic simulation between them for a single transaction. The proof obligation is that the output of pipelined CCDFGs and RTL implantations are equivalent.

Before running SEC, we assume the CCDFG and the RTL are equivalent. SEC checks whether the equivalence is still maintained after executed one transaction. The existing SEC approach can be applied directly. Furthermore, because the mapping between CCDFG’s operation and the RTL’s functional units is still maintained, we can apply *cutpoint* to further improve the scalability.

## 6.4 EXPERIMENTAL RESULTS

We have implemented the reference function pipelining transformation on top of our framework for behavioral synthesis certification. SEC is implemented by a cycle-by-cycle dual-rail, word-level symbolic simulation between CCDFG and RTL, with support for cutpoint optimization. We employ CVC3 as the SMT solver. We have applied our tool to a collection of function pipelined designs synthesized by AutoESL. These designs are carefully selected from several different application domains. For instance, *TEA* and *XTEA* are cryptography algorithms, which have complex bitwise operations, such as *shift* and *bitwise OR*. The *FIR* filter is a signal processing design with internal feedback. Behavioral synthesis utilizes forwarding to optimize this feedback path. All loops in these designs have been fully unrolled. The experiments were conducted on a workstation with 3GHz Intel Xeon processor with 2GB memory. We set the running time bound to two hours.

Table 6.1 illustrates our experimental results. We first conducted brute-force SEC between the un-pipelined CCDFG and the RTL on all designs. None of the runs terminated within the time bound. We then conducted brute-force SEC between the pipelined CCDFG and the RTL. Only the run on *FIR* finished while the others timed out. With the *cutpoint* optimization applied, SEC succeeded on all designs with modest time and memory usages. Column *Cuts* shows the number of cutpoints identified for each design. The experiments demonstrate our function pipelining transformation preserved the internal mapping between CCDFG and RTL which effectively enables cutpoints. Our approach has successfully verified

designs with function pipelines involving several thousand lines of synthesized RTL with reasonable time and memory usages.

As discussed in Section 6.2, a naive solution to handle bubbles is to enumerate all possible input combinations to build corresponding CCDFGs. Unfortunately, the number of such CCDFGs is exponential to the number of scheduling steps and task interval. Given a CCDFG with scheduling steps  $N$ , and task interval  $I$ . There exists  $2^{N/I-1}$  pipelined CCDFGs. This naive approach is clearly impractical. For instance, TEA has 43 scheduling steps and task interval is 1, we have to build  $2^{42}$  CCDFGs and run SEC this many times. In our approach, we creatively encode all possible CCDFGs into a single one by assigning symbols to guard variables. We employ SMT solvers to symbolically explore all possible solutions rather than explicit enumeration. This dramatically improve the efficiency and scalability.

Table 6.1: Function pipelining experimental results

Design	RTL #line	App. Domain	Func Info.			Pipeline Info.		Without Opt.			With Opt.		
			Inter-val	Depth	Operations	Forwarding	Pipeline Register	Mem. (MB)	Time (Sec)	Mem. (MB)	Time (Sec)	#Cuts	
FIR	430	Signal processing	1	5	21	1	5	43	34.8	31	11.5	13	
DCT	941	Signal processing	1	4	48	0	1	-	-	135	26.37	32	
CORDIC	1450	Data processing	1	12	170	0	10	-	-	221	38.83	73	
XTEA	1777	Cryptography	1	32	192	0	147	-	-	114	30.57	32	
TEA	2325	Cryptography	1	43	192	0	211	-	-	100	40.39	85	
YUVTORGB	2412	Image processing	1	5	96	0	4	-	-	333	251.62	48	
MemoryOp	4106	Memory operation	2	39	96	1	75	-	-	43	89.53	75	

## Chapter 7

**CONCLUSION AND FUTURE WORK****7.1 SUMMARY OF CONTRIBUTIONS**

Equivalence checking is highly desired to provide confidence in the correctness of behavioral synthesis. This dissertation research has developed a practical and scalable SEC framework for certifying behavioral synthesis flows. This framework successfully addressed the three major challenges: (1) close the significant semantic gap between ESL and RTL, (2) scale to industry designs, (3) verify the correctness of loop and function pipelines.

To address the above challenges, we have introduced CCDFG as the intermediate design representation, which presumes the design’s control and data flow and augments with a schedule. Our SEC algorithm is based on word-level dual-rail symbolic simulation for comparing CCDFG and RTL. The scalability has been dramatically improved by employing SMT solvers as the decision engine. We have developed three effective optimizations targeting different design features. The optimizations exploit the high-level structure of the ESL description to further ameliorate verification complexity. The experimental results have demonstrated that our optimized SEC framework is capable of verifying designs with tens of thousands of lines of RTL synthesized by a state-of-the-art behavioral synthesis tool. We have also developed approaches to certifying behaviorally synthesized loop and function pipelines. The crucial steps here are to develop reference pipelining transformations for loop pipelining and function pipelining. Thus, we can apply SEC between the reference model and synthesized RTL. The key insight is that the

parameterized, synthesis-guided pipelining reference transformations on CCDFG permits comparison with RTL even after mappings with the original sequential specification has been destroyed by loop and function pipelining. The mapping between behavioral level operations and RTL functional units is still preserved; therefore some key optimizations are applicable. To reduce the complexity brought by “bubbles”, which is a specific difficulty in function pipelines, we have encoded all possible bubble insertions into one single CCDFG. Therefore, we can employ SMT solvers to symbolically explore all possible solutions rather than explicit enumeration. The experimental results have shown that our approaches are efficient and scalable to apply on various synthesized pipelines from different application domains.

## 7.2 FUTURE RESEARCH DIRECTIONS

This dissertation has developed an SEC framework for certifying behavioral synthesis flows. However, there are many aspects of behavioral synthesis that have not been explored. This section discusses several future research directions.

### 7.2.1 Hierarchical Function Pipelines

We have presented our approach to certifying function pipelines generated by behavioral synthesis in Chapter 6. One assumption of our approach is that all sub-functions have been fully inlined. However, inlining sub-functions brings two major drawbacks: (1) redundant checks for sub-functions which are invoked multiple times, (2) the complexity of SEC increases exponentially. An ideal solution is to enable compositional reasoning. The modular analysis approach proposed in Section 4.4 is not applicable in function pipelines. The reason is that we represent the pre-certified sub-functions as uninterpreted functions, which are untimed. However, in order to verify the overlapping execution of pipelined sub-functions,

we need timing information. Therefore, how to introduce a timed uninterpreted function model is crucial for certifying hierarchical function pipelines.

### 7.2.2 Verification of Behaviorally Synthesized Interfaces

Behavioral synthesis can automatically translate high-level interfaces into RTL interfaces which have various communication protocols. For instance, given a ESL design specified in a synthesizable subsets of C/C++, which has an output pointer in its parameter list. This pointer can be synthesized into a First In, First Out (FIFO) interface in the RTL. The FIFO interfaces need to follow the corresponding protocols, such as the *write* signal can be asserted until FIFO is not full (*full* signal is low). Therefore, certification of the synthesized interfaces is also a key part to proof the correctness of the synthesized RTL implementation.

### 7.2.3 SEC for Compiler Transformations in Behavioral Synthesis

We proposed an approach to certifying compiler transformations in behavioral synthesis once and for all using theorem proving [59]. The cost of a monolithic proof is mitigated by the reusability of the transformation over different designs. This approach requires a comprehensive knowledge of the algorithms of transformations employed by behavioral synthesis tools. However, they may be not available, especially for those commercial tools. An alternative solution is to utilize SEC to verify the correctness of compiler transformations by checking the equivalence between input and output of each instance. Therefore, we do not need to require internal algorithms in behavioral synthesis. However, because the variable mappings may not be preserved between input and output, how to develop effective optimizations to make this approach scalable is quite challenging.



## REFERENCES

- [1] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *Proceedings of the 8th international SPIN workshop on Model checking of software*, SPIN '01, pages 103–122, New York, NY, USA, 2001. Springer-Verlag New York, Inc.
- [2] C. Barrett, R. Sebastiani, S. Seshia, and C. Tinelli. *Satisfiability Modulo Theories*. IOS Press, 2009.
- [3] C. Barrett and C. Tinelli. Cvc3. In *Proceedings of the 19th international conference on Computer aided verification*, CAV'07, pages 298–302, Berlin, Heidelberg, 2007. Springer-Verlag.
- [4] J. Baumgartner, H. Mony, V. Paruthi, R. Kanzelman, and G. Janssen. Scalable sequential equivalence checking across arbitrary design transformations. In *Proceedings of the 2006 IEEE/ACM International Conference on Computer-Aided Design*. IEEE Computer Society, 2006.
- [5] C. Berman and L. Trevillyan. Functional comparison of logic designs for vlsicircuits. In *International Conference on Computer-Aided Design*, pages 456–459, 1989. USA.
- [6] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without bdds. In *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 193–207, London, UK, 1999. Springer-Verlag.

- [7] D. Brand. Verification of large synthesized designs. In *Proceedings of the 1993 IEEE/ACM international conference on Computer-aided design*, pages 534–537, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.
- [8] R. E. Bryant. Symbolic manipulation of boolean functions using a graphical representation. In *Proceedings of the 22nd Design Automation Conference*, pages 688–694. IEEE Computer Society Press, 1985.
- [9] R. E. Bryant. A methodology for hardware verification based on logic simulation. *Journal of the ACM*, 38(2):299–328, 1991.
- [10] J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In *Proceedings of the 6th International Conference on Computer Aided Verification, CAV '94*, pages 68–80, London, UK, UK, 1994. Springer-Verlag.
- [11] Cadence. *C-to-Silicon Compiler User Guide*, 2012.
- [12] Calypto Design Systems Inc. <http://www.calypto.com>.
- [13] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski. Legup: high-level synthesis for fpga-based processor/accelerator systems. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays, FPGA '11*, pages 33–36, New York, NY, USA, 2011. ACM.
- [14] R. O. Chapman. *Verified high-level synthesis*. PhD thesis, Ithaca, NY, USA, 1994.
- [15] P. Chauhan, D. Goyal, G. Hasteer, A. Mathur, and N. Sharma. Non-cycle-accurate sequential equivalence checking. In *Proceedings of the 46th Annual Design Automation Conference*, pages 460–465, New York, NY, USA, 2009. ACM.

- [16] K.-T. Cheng and V. D. Agrawal. *Unified Methods for VLSI Simulation and Test Generation*. Kluwer Academic Publishers, 1989.
- [17] L. Claesen, M. Genoe, and E. Verlind. Implementation/specification verification by means of SFG-Tracing. In *CHARME*, 1993.
- [18] E. Clarke, D. Kroening, and K. Yorav. Behavioral consistency of c and verilog programs using bounded model checking. In *Proceedings of the 40th annual Design Automation Conference*, pages 368–371, New York, NY, USA, 2003. ACM.
- [19] S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158, New York, NY, USA, 1971. ACM.
- [20] B. Dutertre and L. de Moura. A fast linear-arithmetic solver for dpll(t). In *Proceedings of the 18th international conference on Computer Aided Verification*, pages 81–94, Berlin, Heidelberg, 2006. Springer-Verlag.
- [21] X. Feng, A. J. Hu, and J. Yang. Partitioned model checking from software specifications. In *Proceedings of the 2005 Asia and South Pacific Design Automation Conference, ASP-DAC '05*, pages 583–587, New York, NY, USA, 2005. ACM.
- [22] R. Floyd. Assigning Meanings to Programs. In *Mathematical Aspects of Computer Science, Proc. of Symposia in Applied Mathematics*, 1967.
- [23] Forte Design Systems. *Cynthesizer Manual*, 2012.
- [24] M. Fujita, H. Fujisawa, and N. Kawato. Evaluations and improvements of a boolean comparison program based on binary decision diagrams. In *Proceedings of the 1988 IEEE/ACM International Conference on Computer-Aided Design*, pages 2–5. IEEE Computer Society Press, 1988.

- [25] D. Gajski, N. D. Dutt, A. Wu, and S. Lin. *High Level Synthesis: Introduction to Chip and System Design*. Kluwer Academic Publishers, 1993.
- [26] V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In *Proceedings of the 19th international conference on Computer aided verification*, pages 519–531, Berlin, Heidelberg, 2007. Springer-Verlag.
- [27] E. Giunchiglia and A. Tacchella, editors. *Theory and Applications of Satisfiability Testing*, volume 2919. Springer, 2004.
- [28] M. Gordon, J. Iyoda, S. Owens, and K. Slind. Automatic formal synthesis of hardware from higher order logic. *Electron. Notes Theor. Comput. Sci.*, 145:27–43, Jan. 2006.
- [29] K. Hao, S. Ray, and F. Xie. Equivalence checking for behaviorally synthesized pipelines. In *Proceedings of the 49th Annual Design Automation Conference*, pages 344–349, New York, NY, USA, 2012. ACM.
- [30] K. Hao, F. Xie, S. Ray, and J. Yang. Optimizing equivalence checking for behavioral synthesis. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 1500–1505, 3001 Leuven, Belgium, Belgium, 2010. European Design and Automation Association.
- [31] C. A. R. Hoare. An axiomatic basis for computer programming. volume 12, pages 576–580. ACM, New York, NY, USA, Oct. 1969.
- [32] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
- [33] A. J. Hu. High-level vs. RTL combinational equivalence: An introduction. In *Proceedings of the 2006 IEEE/ACM International Conference on Computer-Aided Design*, pages 274–279. IEEE Computer Society, 2006.

- [34] M. Huth and M. Ryan. *Logic in Computer Science*. Cambridge University Press, 2006.
- [35] S.-W. Jeong, B. Plessier, G. D. Hachtel, and F. Somenzi. Variable ordering and selection for fsm traversal. In *Proceedings of the 1991 IEEE/ACM International Conference on Computer-Aided Design*, pages 476–479. IEEE Computer Society Press, 1991.
- [36] R. Johnson and K. Pingali. Dependence-based program analysis. In *Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation, PLDI '93*, pages 78–89, New York, NY, USA, 1993. ACM.
- [37] R. B. Jones, D. L. Dill, and J. R. Burch. Efficient validity checking for processor verification. In *Proceedings of the 1995 IEEE/ACM international conference on Computer-aided design, ICCAD '95*, pages 2–6, Washington, DC, USA, 1995. IEEE Computer Society.
- [38] D. Kaiss, S. Goldenberg, Z. Hanna, and Z. Khasidashvili. Seqver: A sequential equivalence verifier for hardware designs. In *Proceedings of the 2006 IEEE/ACM International Conference on Computer-Aided Design*, pages 267–273. IEEE Computer Society, 2006.
- [39] M. Kaufmann, P. Manolios, and J. S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, Boston, MA, June 2000.
- [40] H. Kautz and B. Selman. Planning as satisfiability. In *Proceedings of the 10th European conference on Artificial intelligence*, pages 359–363, New York, NY, USA, 1992. John Wiley & Sons, Inc.
- [41] A. Koelbl, J. R. Burch, and C. Pixley. Memory modeling in esl-rtl equivalence

- checking. In *Proceedings of the 44th annual Design Automation Conference*, pages 205–209, New York, NY, USA, 2007. ACM.
- [42] A. Koelbl, R. Jacoby, H. Jain, and C. Pixley. Solver technology for system-level to rtl equivalence checking. In *Proceedings of the Conference on Design, Automation and Test in Europe, DATE '09*, pages 196–201, 3001 Leuven, Belgium, Belgium, 2009. European Design and Automation Association.
- [43] A. Koelbl and C. Pixley. Constructing efficient formal models from high-level descriptions using symbolic simulation. *Int. J. Parallel Program.*, 33(6):645–666, Dec. 2005.
- [44] D. Kroening and E. Clarke. Checking consistency of c and verilog using predicate abstraction and induction. In *Proceedings of the 2004 IEEE/ACM International conference on Computer-aided design*, pages 66–72, Washington, DC, USA, 2004. IEEE Computer Society.
- [45] A. Kuehlmann and F. Krohm. Equivalence checking using cuts and heaps. In *Proceedings of the 34th annual Design Automation Conference, DAC '97*, pages 263–268, New York, NY, USA, 1997. ACM.
- [46] S. Kundu, S. Lerner, and R. Gupta. Validating high-level synthesis. In *Proceedings of the 20th international conference on Computer Aided Verification, CAV '08*, pages 459–472, Berlin, Heidelberg, 2008. Springer-Verlag.
- [47] C. Y. Lee. Binary decision programs. *Bell System Technical Journal*, 38(4):985–999, 1959.
- [48] X. Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, July 2009.
- [49] J. Levitt and K. Olukotun. A scalable formal verification methodology for

- pipelined microprocessors. In *Proceedings of the 33rd annual Design Automation Conference*, DAC '96, pages 558–563, New York, NY, USA, 1996. ACM.
- [50] LLVM Project. The LLVM Compiler Infrastructure. <http://llvm.org>.
- [51] S. Malik, A. Wang, R. K. Brayton, and A. Sangiovanni-Vincentelli. Logic verification using binary decision diagrams in a logic synthesis environment. In *Proceedings of the 1988 IEEE/ACM International Conference on Computer-Aided Design*, pages 6–9. IEEE Computer Society Press, 1988.
- [52] P. Manolios, S. K. Srinivasan, and D. Vroon. Automatic memory reductions for rtl model verification. In *Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design*, pages 786–793, New York, NY, USA, 2006. ACM.
- [53] A. Mathur, M. Fujita, E. Clarke, and P. Urard. Functional equivalence verification tools in high-level synthesis flows. *IEEE Des. Test*, 26(4):88–95, July 2009.
- [54] Mentor Graphics. *Catapult C Reference Manual*, 2011.
- [55] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: engineering an efficient sat solver. In *Proceedings of the 38th annual Design Automation Conference*, pages 530–535, New York, NY, USA, 2001. ACM.
- [56] G.-J. Nam, K. A. Sakallah, and R. A. Rutenbar. Satisfiability-based layout revisited: detailed routing of complex fpgas via search-based boolean sat. In *Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays*, pages 167–175, New York, NY, USA, 1999. ACM.
- [57] G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. Program. Lang. Syst.*, 1(2):245–257, Oct. 1979.

- [58] OCaml. <http://caml.inria.fr>.
- [59] S. Ray, K. Hao, Y. Chen, F. Xie, and J. Yang. Formal verification for high-assurance behavioral synthesis. In *Proceedings of the 7th International Symposium on Automated Technology for Verification and Analysis, ATVA '09*, pages 337–351, Berlin, Heidelberg, 2009. Springer-Verlag.
- [60] P. R. Schaumont. *A Practical Introduction to Hardware/Software Codesign*. Springer, 2010.
- [61] K. Schneider. A verified hardware synthesis of esterel programs. In *Proceedings of the International Workshop on Distributed and Parallel Embedded Systems: Architecture and Design of Distributed Embedded Systems, DIPES '00*, pages 205–214, Deventer, The Netherlands, The Netherlands, 2001. Kluwer, B.V.
- [62] C. J. Seger, R. B. Jones, J. W. O’Leary, T. Melham, M. D. Aagaard, C. Barrett, and D. Syme. An industrially effective environment for formal hardware verification. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 24(9):1381–1405, Nov. 2006.
- [63] O. Shtrichman. Tuning sat checkers for bounded model checking. In *Proceedings of the 12th International Conference on Computer Aided Verification*, pages 480–494, London, UK, 2000. Springer-Verlag.
- [64] J. P. M. Silva. Practical applications of boolean satisfiability. In *Proceedings of the 9th International Workshop on Discrete Event Systems*, pages 74–80. IEEE, 2008.
- [65] J. P. M. Silva and K. A. Sakallah. Grasp: A search algorithm for propositional satisfiability. *IEEE Trans. Computers*, 48(5):506–521, 1999.
- [66] N. Sinha. Symbolic program analysis using term rewriting and generalization. In *Proceedings of the 2008 International Conference on Formal Methods in*



- Computer-Aided Design*, FMCAD '08, pages 19:1–19:9, Piscataway, NJ, USA, 2008. IEEE Press.
- [67] M. Stepp, R. Tate, and S. Lerner. Equality-based translation validator for llvm. In *Proceedings of the 23rd international conference on Computer aided verification*, pages 737–742, Berlin, Heidelberg, 2011. Springer-Verlag.
- [68] G. S. Tseitin. On the complexity of derivation in propositional calculus. *Studies in Constructive Mathematics and Mathematical Logic*, Part II:178–188, 1968.
- [69] M. N. Velev and R. E. Bryant. Verification of pipelined microprocessors by correspondence checking in symbolic ternary simulation. In *Proceedings of the 1998 International Conference on Application of Concurrency to System Design*, CSD '98, pages 200–, Washington, DC, USA, 1998. IEEE Computer Society.
- [70] K. Wakabayashi and H. Tanaka. Global scheduling independent of control dependencies based on condition vectors. In *Proceedings of the 29th ACM/IEEE Design Automation Conference*, DAC '92, pages 112–115, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.
- [71] D. J. Wheeler and R. M. Needham. TEA, a tiny encryption algorithm. In *Fast Software Encryption*, pages 363–366, 1994.
- [72] Xilinx. *AutoESL Reference Manual*, 2011.
- [73] J. Yang and C.-J. H. Seger. Introduction to generalized symbolic trajectory evaluation. *IEEE Trans. Very Large Scale Integr. Syst.*, 11(3):345–353, June 2003.
- [74] J. Zhao, S. Nagarakatte, M. M. Martin, and S. Zdancewic. Formalizing the

llvm intermediate representation for verified program transformations. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 427–440, New York, NY, USA, 2012. ACM.