Certifying Loop Pipelining Transformations in Behavioral Synthesis

by

Disha Puri

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

Doctor of Philosophy
in
Computer Science

Dissertation Committee:
Fei Xie, Chair
Jingke Li
Suresh Singh
Sandip Ray
Fu Li

Portland State University
2016

ABSTRACT

Due to the rapidly increasing complexity in hardware designs and competitive time to market trends in the industry, there is an inherent need to move designs to a higher level of abstraction. Behavioral Synthesis is the process of automatically compiling such Electronic System Level (ESL) designs written in high-level languages such as C, C++ or SystemC into Register-Transfer Level (RTL) implementation in hardware description languages such as Verilog or VHDL. However, the adoption of this flow is dependent on designers' faith in the correctness of behavioral synthesis tools.

Loop pipelining is a critical transformation employed in behavioral synthesis process, and ubiquitous in commercial and academic behavioral synthesis tools. It improves the throughput and reduces the latency of the synthesized hardware. It is complex and error-prone, and a small bug can result in faulty hardware with expensive ramifications. Therefore, it is critical to certify the loop pipelining transformation so that designers can trust the behaviorally synthesized pipelined designs. Certifying a loop pipelining transformation is however, a major research challenge because there is a huge semantic gap between the input sequential design and the output pipelined implementation, making it infeasible to verify their equivalence with automated sequential equivalence checking (SEC) techniques.

Complex loop pipelining transformations can be certified by a combination of theorem proving and SEC: (1) creating a certified pipelining algorithm which

generates a reference pipeline model by exploiting pipeline generation information from the synthesis flow (*e.g.*, the iteration interval of a generated pipeline) and (2) conduct SEC between the synthesized pipeline and this reference model. However, a key and arguably, the most complex component of this approach is the development of a formal, mechanically verifiable loop pipelining algorithm. There has been an attempt to create such a pipelining algorithm but it was not certified. Infact, our experience shows that it was incomplete and error prone. We show how to systematically construct such an algorithm, and carry out its verification using the ACL2 theorem prover. We propose a framework of certified pipelining primitives which we show are essential for designing pipelining algorithms. Using our framework, we build a certified loop pipelining algorithm. We also propose a key invariant in certifying this algorithm, which links sequential loops with their pipelined counterparts. This is unlike other invariants that have been used in proofs of microprocessor pipelines so far.

This dissertation provides a framework for creating certified pipelining algorithms utilizing a mechanical theorem prover. Using this framework, we have developed a certified loop pipelining algorithm. This certified algorithm is essential in the overall approach to certify behaviorally synthesized pipelined designs. We demonstrate the scalability and robustness of our algorithm on ESL designs across various domains.

DEDICATION

ACKNOWLEDGMENTS

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

INTRODUCTION

1.1 DISSERTATION SUMMARY

Developing a certified loop pipelining algorithm is a complex problem. We have developed a certified loop pipelining algorithm for behavioral synthesis by proper application of theorem proving techniques. The result of this dissertation is a framework of certified pipelining primitives which are essential in developing any such pipelining algorithm. We systematically build a loop pipelining algorithm from ground up using these primitives and certify this algorithm.

1.2 MOTIVATION

Behavioral synthesis [15, 4] is the process of synthesizing an Electronic System-level (ESL) specification of a hardware design into a Register-Transfer Level (RTL) implementation. The idea of ESL is to raise the design abstraction by specifying the high-level, functional behavior of the hardware design. Designs are typically specified in a language such as C, C++, or SystemC. The approach is promising since the user is relieved of developing and optimizing low-level implementations. Studies have shown that ESL reduces the design effort by 50% or more while attaining excellent performance results [33]. It has recently received significant attention, as the steady increase in hardware complexity has made it increasingly

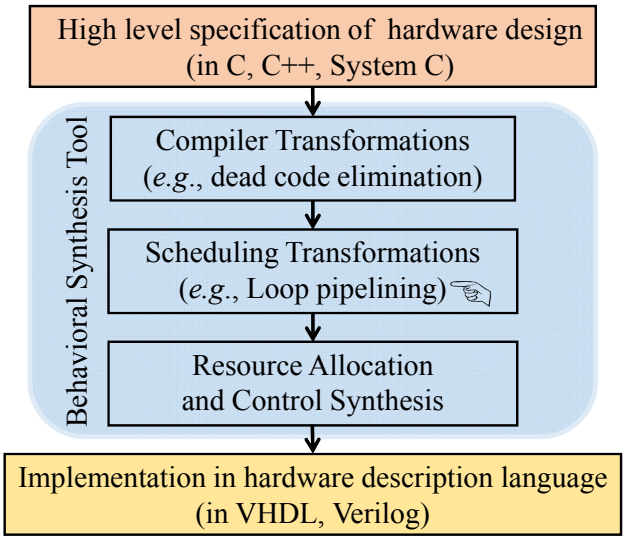


Figure 1.1: Behavioral Synthesis Flow

difficult to design high-quality designs through hand-crafted RTL under aggressive time-to-market schedules. A recent example is VP9 G2 hardware decoder IP developed by Google [2], which has been implemented primarily in standard C++ and synthesized to RTL logic for different target technologies and performance points using Calypto's Calatpult High Level Synthesis tool [8]. Nevertheless, and in spite of availability of several commercial behavioral synthesis tools [7, 48, 19], the adoption of the approach in main-stream hardware development for microprocessor and SoC design companies is dependent on designers' confidence that the synthesized RTL indeed corresponds to the ESL specification.

To satisfy the power and performance demands of modern applications, a behavioral synthesis tool applies hundreds of transformations. As shown in Figure 1.1, a typical behavioral synthesis flow can be roughly divided into three phases: compiler transformations; scheduling transformations and resource allocation and control synthesis. Commercial synthesis tools are highly complex software involving thousands to millions of lines of code; furthermore, they perform aggressive

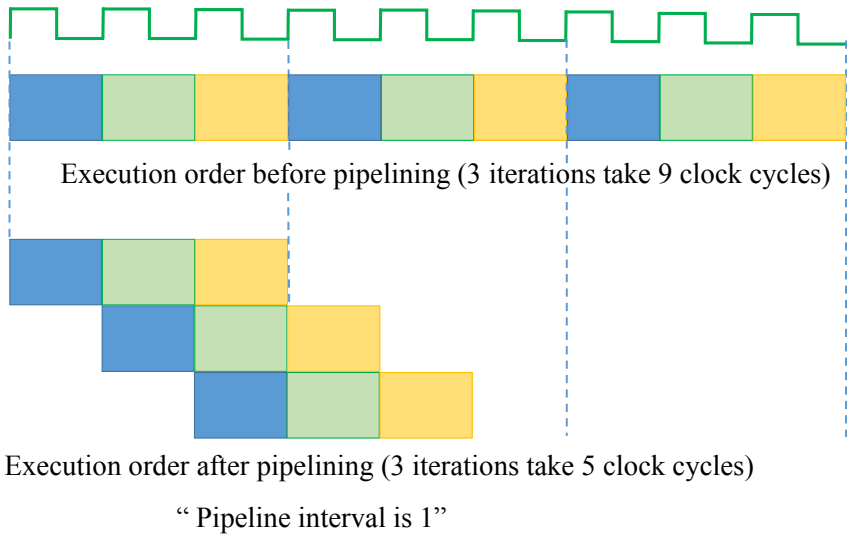


Figure 1.2: Loop Pipelining

optimizations on the design being synthesized to satisfy constraints on power, performance, and area. Tools of such compelxity invariably contain subtle bugs, which undermine the very effectiveness of behavioral synthesis. Consequently, it is critical to develop a methodology for certifying synthesis transformations. Thus it is critical to develop mechanized support for certifying the equivalence between ESL and RTL designs. However, the large difference in abstraction between the two representations makes such certification non-trivial.

Loop pipelining is a critical transformation in behavioral synthesis. The goal of this transformation is to increase throughput and reduce latency of the synthesized hardware by allowing temporal overlap of successive loop iterations. As shown in Figure 1.2, the three iterations of overlapped pipeline structure takes only five clock cycles as opposed to nine clock cycles if executed sequentially. It is performed by most state-of-the-art synthesis tools [48, 19, 9].

Unfortunately, it is also highly complex [61] and error-prone, requiring subtle analysis of invariants to preclude data hazards arising from overlapping control/data flow of executions of successive loop iterations. Furthermore, certifying the result of this transformation is very difficult. In particular, the pipelined output design from the transformation has a markedly different control/data flow structure from the sequential design that is input to the transformation; this makes it hard to find corresponding internal signals to serve as cutpoints, making it hard to compare them through sequential equivalence checking. On the other hand, applying theorem proving to certify the pipelining transformation is clearly cost-prohibitive given the complexity of the implementation; furthermore, most commercial transformation implementations are proprietary, making it infeasible to develop such a framework from a methdological perspective. As a result, hardware designers are vary of using current behavioral synthesis tools as they are often deemed either (a) aggressively optimized but error-prone or (b) reliable but overly conservative, thus often producing circuits of poor quality or performance [20, 40]. Therefore, ensuring correctness of behaviorally synthesized pipeline designs is a critical issue in bringing behavioral synthesis into practice.

An approach for certifying loop pipelining transformations using a combination of SEC and theorem proving techniques has been proposed by Hao et al. [24]. The most critical and complex component of their approach (c.f. Section 2.3) is developing a loop pipelining algorithm with two key properties: (1) it generates a reference pipeline model by exploiting pipeline generation information from the synthesis flow (e.g., the iteration interval of a generated pipeline) and the reference model can be compared with a pipelined RTL implementation using SEC effectively, and (2) it can be mechanically verified to correctly preserve the semantics of sequential (non-pipelined) specification of loop execution. Hao et al. showed

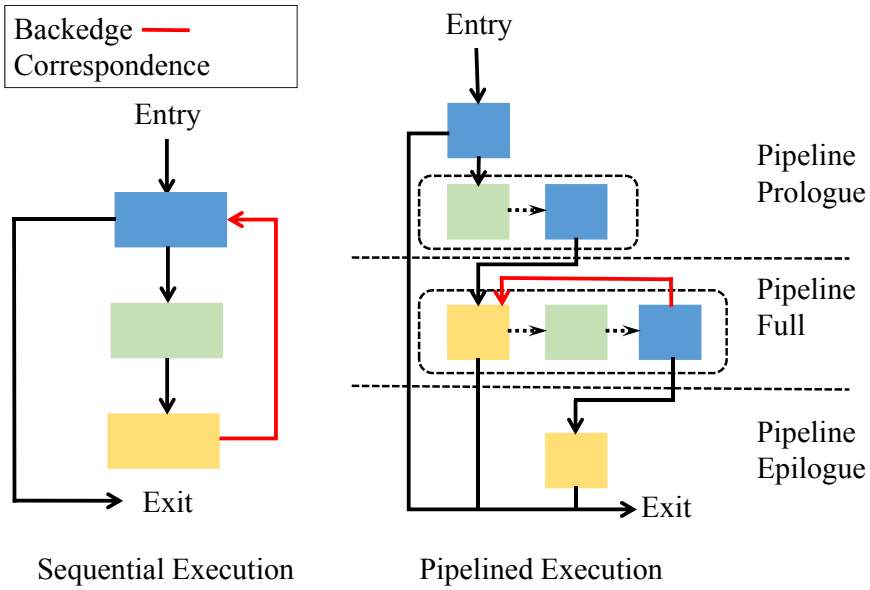


Figure 1.3: Back-edge in Sequential Loop Vs Back-edge in Pipelined Loop

viability of their approach by comparing pipeline generated from their algorithm with RTL implementation using SEC. However, their algorithm was not certified as well as incomplete as explained later in Section 2.3. Certification is a key component without which correctness of behavioral synthesis process for pipelined designs cannot be claimed. Therefore, this dissertation on developing a certified loop pipelining algorithm using our framework of certified primitives is important to facilitating formal verification of behaviorally synthesized pipeline designs.

1.3 PROBLEM STATEMENT

Certifying an algorithm especially as complex as loop pipelining is not easy by any known conventional methods. To develop a certified loop pipelining algorithm in behavioral synthesis, we need to address the following key challenges brought about by the semantic gap between the sequential and pipelined designs.

– *Formalizing an invariant that links loop in a sequential design with loop in the corresponding pipelined design.* As shown in Figure 1.3, a sequential loop executes its iterations in sequence. The previous loop iteration is complete before the next iteration. A pipelined design, however, overlaps the consecutive iterations of a given design based on pipeline interval. As a result, a loop in the pipelined design executes statements from different iterations of the corresponding sequential loop design. Identifying a provable inductive invariant that links the backedge in the sequential loop with the backedge in the pipelined loop is, therefore, a major challenge.

– *Identifying and certifying underlying primitives in a loop pipelining algorithm.* Certifying a loop pipelining algorithm requires a complex invariant to prove that executing a sequential loop is equivalent to executing a pipelined loop. Identifying the pieces which would make this certification managable is a difficult task. We decompose the algorithm into certifiable primitives. We prove that if each primitive maintains an invariant that the execution of the intermediate representation before and after application of the primitive is same, we can prove that the algorithm also maintains the invariant. This approach, however, requires a crisp understanding of the essential steps involved in developing a pipeline loop from a sequential loop. We need to succinctly identify primitives which maintain the given invariant and are also certifiable by theorem proving. Each primitive would require a systematic approach for its proof.

– *Identify and maintain control flow by proper placement of branches.* Branch conditions dictate the control flow. Presence of conditional and unconditional branch instructions in a loop means that the loop is no longer executed in a straight line. At each application of primitive, we need to ensure that this

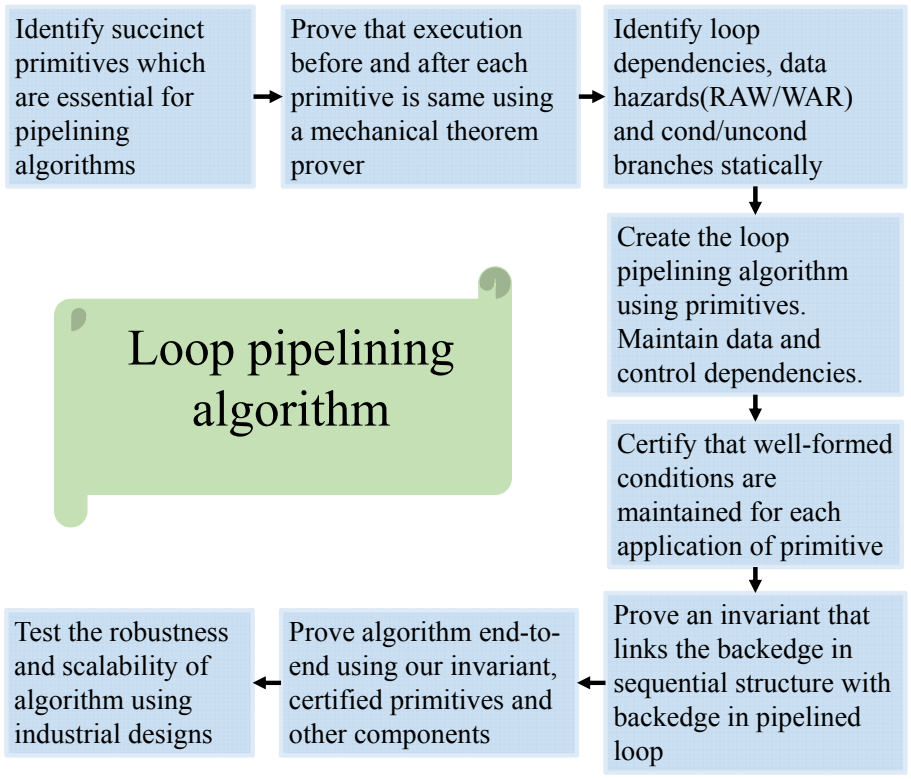


Figure 1.4: Overview of our Approach

control flow is not disturbed or is atleast well accounted for.

– *Certifying the complete loop pipelining algorithm based on certified primitives.* Although, the primitives act as backbone for our algorithm, their certification alone does not automatically certify the entire algorithm. We need to identify the conditions under which a primitive is correct and make sure that every application of the primitive in the algorithm has the required assumptions.

1.4 OVERVIEW OF OUR APPROACH

Our work shows that a certified loop pipelining algorithm can be developed by systematic application of theorem proving techniques.

We have basically divided our approach into eight broad steps as shown in Figure 1.4. The first two steps define our framework of certified primitives which we believe are essential for any certified pipelining algorithm. We have identified these primitives based on a realization that in order to generate a pipelined loop design from a sequential loop design, there are three broad steps: (1) identification and removal of data hazards, (2) overlapping the executions of subsequent iterations after the removal of data hazards, and (3) maintaining the correct control flow to preserve the exit condition and state at the time of exit from the loop. Our primitives are such that they can be applied alone or in combination with other primitives to remove data hazards, reason about branches and to overlap iterations. We certify each primitive by proving that execution before and after each primitive is correct. Certification of each primitive requires separate careful reasoning in a mechanical theorem prover which we describe later in Chapter 6. We have defined the syntax and semantics of intermediate design representation in ACL2 [39, 55]. We have formalized and certified all of our primitives in ACL2 theorem prover.

The next six steps are for creating the certified loop pipelining algorithm using our framework of certified primitives. Certifying an application of a primitive in the context of the algorithm further involves ensuring that addition of any primitive does not alter the underlying assumptions in the syntax, for example, if we assume there are no return statements in a given representation, applying any primitive should also maintain that assumption. We use these primitives as a backbone to build our loop pipelining algorithm with distinct decomposable components one step at a time. Besides primitives, there are also additional components in the algorithm such as for identifying data hazards and for unrolling the loop. Each component satisfies the invariant that execution of intermediate representations before and after the component is same. We elaborate on our approach later in

Chapter 5.

We have also identified a unique invariant which proves that executing overlapped iterations is equivalent to executing sequential iterations. It differs from a typical invariant used for correctness of pipelined systems in that it explicitly specifies the correspondence between the sequential and pipelined programs at each transition. We elaborate on our invariant in Chapter 6. We have proved that our algorithm satisfies this invariant.

We have certified the algorithm end-to-end which means that given a well-formed pipelinable loop (definition explained later in Chapter 5), we show that executing a sequential loop is equivalent to executing the pipelined loop created using our algorithm. We elaborate on the proof in our proof sketch in Chapter 6. Our proof sketch shows that our primitives are sufficient and essential and that we can build a complete certified loop pipelining algorithm from ground up using our framework.

The major **contributions** of our dissertation are:

– Identifying the key provable primitives essential in pipelining algorithms for behavioral synthesis and certifying these primitives in ACL2 theorem prover;

– Formalizing an invariant to link the sequential loop before pipelining with the pipelined loop;

– Developing our own executable loop pipelining algorithm in ACL2 using those primitives and certifying this algorithm using ACL2 theorem prover;

– Testing our certified loop pipelining algorithm on industrial-strength designs

1.5 OUTLINE

The remainder of this dissertation is organized as follows. Chapter 2 provides background on the overall project and explains the context of our theorem proving work. Chapter 3 discusses our formalization of the intermediate representations used in behavioral synthesis. We also discuss the correctness statement for loop pipelining algorithms. Chapter 4 discusses an earlier proposed algorithm and how and why we have chosen a different approach. Chapter 5 discusses our framework and a certified loop pipelining algorithm we have developed using the framework. Chapter 6 provides a proof sketch for our algorithm. Chapter 7 provides evaluation of robustness and scalability of our algorithm on industrial-strength designs. The related work is discussed in Chapter 8. We then conclude with the major contributions of this dissertation and future work in Chapter 9.

Chapter 2

BACKGROUND AND CONTEXT

In this Chapter, we discuss the overall project of verifying behaviorally synthesized designs, and how the certification of loop pipelining fits into this project. The reader interested in a thorough understanding of other components of the project is welcome to review the prior publications [52, 25].

2.1 BEHAVIORAL SYNTHESIS

Behavioral synthesis [44] is an automated compilation process where a behavioral synthesis tool [20, 14, 9] takes an ESL description, together with a library of hardware resources. Analogous to a regular compiler, the tool performs the standard lexical and syntax analysis to generate an intermediate representation (IR). The IR is then subjected to a number of transformations which can be categorized into three phases as shown in Figure 2.1.

– **Compiler Transformations:** These include typical compiler operations, *e.g.,* dead-code elimination, constant propagation, loop unrolling, common subexpression elimination etc. Furthermore, expensive operations (*e.g.,* division) may be replaced with simpler ones (*e.g.,* subtraction). A design may undergo hundreds of compiler transformations.

– **Scheduling Transformations:** Scheduling entails computing for each operation the clock cycle of its execution, accounting for hardware resource constraints

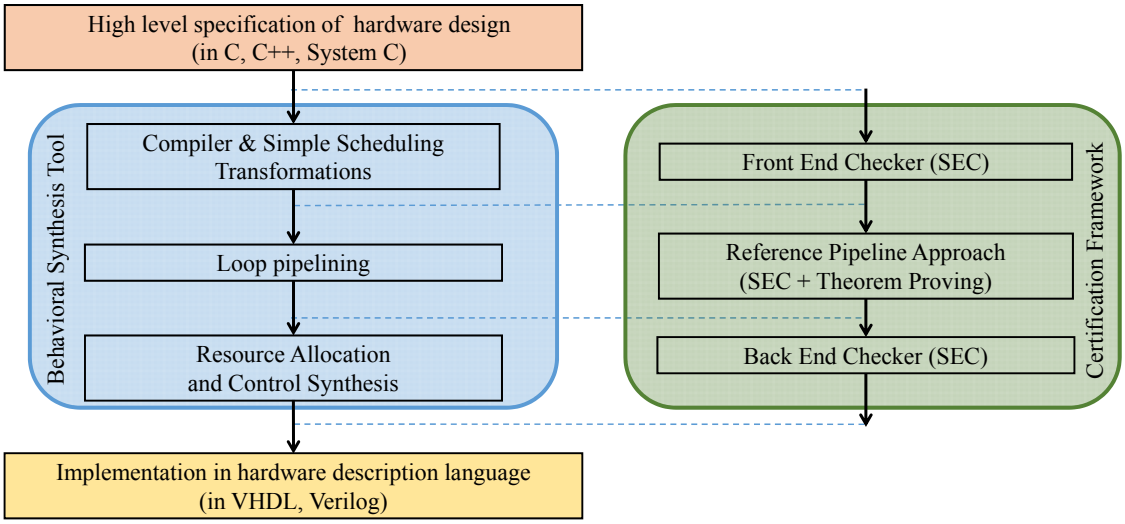


Figure 2.1: Certification Model for Behaviorally Synthesized Pipelines

and control/data dependencies. Loop pipelining, the focus of our dissertation thesis, is a component of this phase.

– **Resource Allocation and Control Synthesis:** This phase involves mapping a hardware resource to each operation (the "+" operation may be mapped to a hardware adder), allocating registers to variables, and generating a controlling finite-state machine to implement the schedule.

After the three phases above, the design can be expressed in RTL. The synthesized RTL may be subjected to further manual tweaks to optimize for area, power, etc. Each of these transformations are non-trivial. The end result is a hardware implemtation which has a huge abstraction gap from the input ESL description.

2.2 OVERALL CERTIFICATION MODEL FOR BEHAVIORALLY SYNTHESIZED PIPELINES

The overall goal of the project is to provide a mechanized framework for certifying hardware designs synthesized from ESL specifications by commercial behavioral synthesis tools. One obvious approach is to apply standard verification techniques (SEC or theorem proving) on the synthesized RTL itself. Unfortunately, such a methodology is not practical. As mentioned earlier, the large gap in abstraction between the ESL and RTL descriptions means that there is little correspondence in internal variables between the two. Consequently, direct SEC between the two reduces to cost-prohibitive computation of input-output equivalence. On the other side, applying theorem proving is also troublesome since extensive manual effort is necessary and this effort needs to be replicated for each different synthesized design. It is also infeasible to directly certify the implementation of the *synthesis tool* via theorem proving. In addition to being highly complex and thus potentially requiring prohibitive effort to formally verify with any theorem prover, the implementations are typically closed-source and closely guarded by EDA vendors and thus out of reach of external automated reasoning communities.

To address this problem, previous work developed two key SEC solutions, which we will refer to below as *Back-end* and *Front-end*. We then discuss the gap between them, which is being filled by theorem proving efforts in this dissertation. The certfication model is illustrated in Figure 2.1.

Back-end SEC: The key insight behind back-end SEC is that automated SEC techniques, while ineffective for directly comparing synthesized RTL with the top-level ESL description, are actually suitable to compare the RTL with the intermediate representation (IR) generated by the tools after the high-level (compiler

and scheduling) transformations have been applied. In particular, operation-to-resource mappings generated by the synthesis tool provide the requisite correspondence between internal variables of the IR and RTL. Furthermore, a key insight is that while the implementations of transformations are unavailable for commercial EDA tools, most tools provide these IRs after each transformation application together with some other auxiliary information. To exploit these, an SEC algorithm was developed between the IR (extracted from synthesis tool flow after these transformations) and RTL [52, 25, 26, 70]. The approach scales to tens of thousands of lines of synthesized RTL.

Front-end SEC: Of course the back-end SEC above is only meaningful if we can certify that the input ESL indeed corresponds to the extracted IR produced after the compiler and scheduling transformations applied in the first two phases of synthesis. To address this, another SEC technique was developed to compare two IRs [67, 69, 68]. The idea then is to obtain the sequence of intermediate representations $\mathrm{IR}_0, \ldots, \mathrm{IR}_n$ generated by the compiler and scheduling transformations, and compare each pair of consecutive IRs with this new algorithm. Then back-end SEC can be used to compare IR_n with the synthesized RTL, completing the flow.

A Methodology Gap: Unfortunately, the front-end SEC algorithm can only compare two IRs that are structurally close. If a transformation significantly transforms the structure of an IR then the heuristics for detecting corresponding variables between the two IRs will not succeed, causing equivalence checking to fail. Unfortunately, loop pipelining falls in the category of transformations that significantly changes the structure of the IR. It is a quintessential transformation that changes the control/data flow and introduces additional control structures (to eliminate hazards). This makes front-end SEC infeasible for its certification.

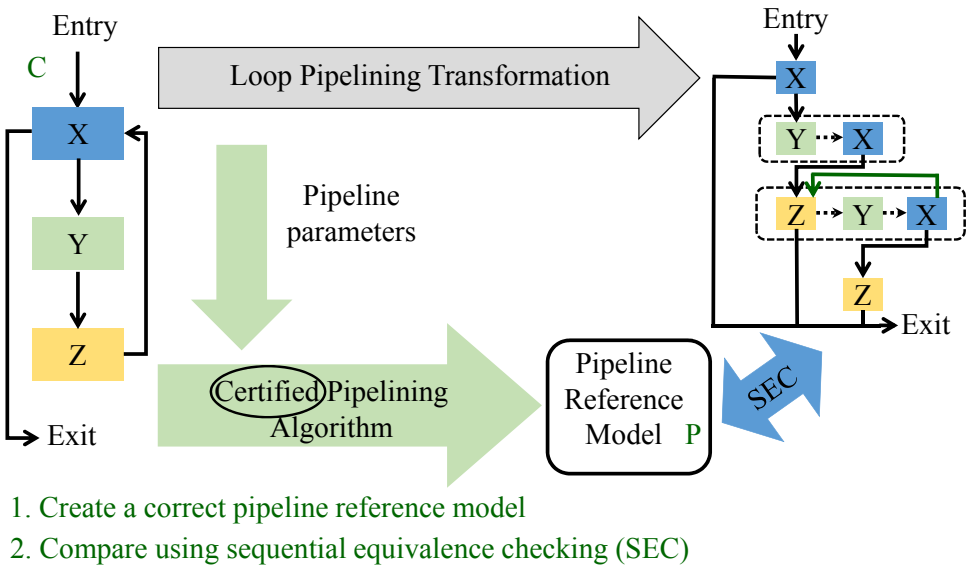


Figure 2.2: Certifying Loop Pipelining Algorithm using SEC and Theorem Proving

Furthermore, most commercial implementations are of course proprietary and consequently not available to us for review; applying theorem proving on those implementations is not viable from a methodology perspective. Thus a specialized approach is warranted for handling its certification.

2.3 A REFERENCE PIPELINE APPROACH

To develop a specialized approach for pipelines, a key observation is that while the transformation *implementation* is inaccessible to us, commercial synthesis tools typically generates a report specifying pipeline parameters (pipeline interval, number of loop iterations pipelined, etc.). The approach (c.f. Figure 2.2) then is to develop an algorithm that takes as inputs these parameters and an IR $\mathcal{C}$ for the design before pipelining, and generates a *reference pipelined IR* $\mathcal{P}$. Note that this algorithm would be much simpler than that employed during synthesis; while the former includes advanced heuristics to *compute* pipeline parameters (like pipeline

interval, number of iterations pipelined etc.), this algorithm would merely use the values provided by its report. To certify a synthesized RTL with pipelines, it is sufficient to (1) check that the *given algorithm* can generate a pipeline $\mathcal{P}$ for the parameters reported by synthesis, (2) use SEC to compare $\mathcal{P}$ with the synthesized RTL, and (3) prove (using theorem proving) the correctness of this algorithm.

A previous work [24] justified the viability of steps 1 and 2 above; such a reference pipeline generation algorithm was developed and used to successfully compare a variety of pipelined designs across various application domains. This suggested that the approach of using a reference implementation is viable for certifying industrial strength behaviorally synthesized pipelines. However, a key (and perhaps the most complicated) component of the approach was missing. The algorithm was not verified (indeed, not implemented in a formal language), rendering the "certification" flow unsound.

The unsoundness mentioned above is not just an academic notion. In fact, merely by going through the formalization process and thinking about necessary invariants, we have already found a bug in the implementation of the algorithm. Thus it is critical to develop a mechanized proof of correctness for this implementation. Unfortunately, it is not easy to verify the original pipeline generation algorithm as written. Its author was an expert in behavioral synthesis but not in program verification or theorem proving; consequently, the algorithm, while simpler than the one implemented in a synthesis tool, was still a highly complex piece of code. In particular, since it was not written with correctness certification in mind, it is difficult to decompose the algorithm into manageable pieces with nice invariants.

One way to address this problem is to "buckle down" and verify the pipeline generation algorithm (and fixing the bugs found in the process). However, a key

insight in our case is that we can get away without verifying such a complex implementation. After all, there is nothing "sacred" about this specific algorithm for pipeline generation: given the steps described above, *any* verifiable pipeline generation algorithm would suffice.[1] Thus the approach of our dissertation can be viewed as a rational deconstruction of the pipeline synthesis algorithm of the previous work. We identify the key invariant that we need to maintain for proving computational equivalence between the pipelined and un-pipelined loops and design an algorithm to explicitly maintain that invariant.

We discuss the previously proposed algorithm in Chapter 4 such that we can draw a comparison and better understand the differences in our implementation of loop pipelining algorithm due to our need to formally certify it.

[1]Note that our algorithm *must* create a pipeline in accordance with the pipeline parameters obtained from the behavioral synthesis tools; otherwise we may fail to certify correct designs. However, in practice, we have not found this to be a problem.

Chapter 3

FORMALIZATION

3.1 INTERMEDIATE REPRESENTATION: CCDFG

In order to formalize and prove the correspondence between pipelined and unpipelined IRs, a first step is to define a formalization of the IRs themselves. We call our formalization of IRs *Clocked Control Data Flow Graph* (CCDFG). An informal description of CCDFG has been provided before [52]. It can be best viewed as a traditional control/data flow graph used by most compilers, augmented with a schedule. Control flow is broken into basic blocks. Instructions are grouped into microsteps which can be executed concurrently. A scheduling step is a group of microsteps which can be executed in a single clock cycle. The state of a CCDFG at a particular microstep is a list of all the variables of a CCDFG with their corresponding values.

The semantics of CCDFG require a formalization of the underlying language used to represent the individual instructions in each scheduling step. The underlying language we use is the LLVM [36]. It is a popular compiler infrastructure for many behavioral synthesis tools [14, 9] and includes an assembly language front-end. We currently support only a subset of LLVM operations which are required to handle all the designs we have seen. Instructions supported include assignment, load, store, bounded arithmetic, bit vectors, arrays, and pointer manipulation instructions. Note that the reasoning involved in creating a pipelined CCDFG does not involve the exact syntax of any operation. We are merely concerned with a

way to find the variables which are read and written at each step. Increasing the operations database in our algorithm is expected to increase the time taken to prove certain primitives as much more analysis needs to be done. However, it would not affect the logical reasoning of the primitives, the overall algorithm and the proof. We define the syntax of each type of statement by defining an ACL2 predicate. For example, in our syntax, an assignment statement can be expressed as a list of a variable and an expression.

```
(defun assignment-statement-p (x)
  (and (equal (len x) 1)
       (and (equal (len (car x)) 2)
            (first (car x)) (symbolp (first (car x)))
            (expression-p (second (car x))))))
```

An expression can further be of multiple types, load expression (loading the value of a variable from memory), add expression (addition of two variables), xor expression (xor of two variables) etc., where each expression includes the operation applied to the appropriate number of arguments.

We provide semantics to these instructions through a state-based operational formalization as is common with ACL2 [45]. We define the notion of a CCDFG state, which includes the states of the variables, memory, pointers, etc. Then we define the semantics of each instruction by specifying how it changes the state. Thus, for an assignment statement we will have a function execute-assignment that specifies the effect of executing the assignment statement on a CCDFG state.

```
(defun add-expression-p (x)
  (and (equal (len x) 3)
       (equal (first x)  'add)
       (variable-or-numberp (second x))
```

```
      (variable-or-numberp (third x))))

(defun expression-p (x)
  (and (consp x)
       (or (load-expression-p x)
        (add-expression-p x)
        (xor-expression-p x)...)))
```

Defining the semantics of most supported statements is straightforward, with one exception. The exception is the so-called "ϕ-construct" available in LLVM [1]. A ϕ-construct is a list of ϕ-statements. A ϕ-statement is $v := \phi[\sigma, bb1][\tau, bb2]$, where v is a variable, σ and τ are expressions, and $bb1$ and $bb2$ are basic blocks: if it is reached from $bb1$ then it is the same as the assignment statement $v := \sigma$; if reached from $bb2$, it is the same as $v := \tau$; the meaning is undefined otherwise. The construct is complex since the effect of executing this statement on a CCDFG state s depends not only on the state s but also on how s is reached by the control flow. Unfortunately, ϕ-statements are required in loop designs — they are used to evaluate the value of loop carried dependencies. Consequently, the complexity induced by this instruction cannot be avoided.

```
(defun phi-expression-p (x)
  (and (consp x) (equal (len x) 1)
       (consp (car x)) (> (len (car x)) 2)
       (equal (caar x) 'phi) (phi-l (cdr (car x)))))

(defun phi-statement-p (x)
  (and (consp x) (equal (len x) 2)
       (symbolp (first x)) (first x)
       (phi-expression-p (cdr x))))
```

Here `phi-1` recognizes an expression of the form `((E0 b) (E1 b-prime))` where `E0` and `E1` are expressions and `b` and `b-prime` are symbols representing basic blocks. Thus in ACL2, the ϕ-statement looks like `(v (phi ((E0 b) (E1 b-prime))))`. Finally, the execution semantics requires the additional parameter `prev-bb` to track the previous basic block.

```
(defun choose (choices prev-bb)
  (if (or (equal (nth 1 (first choices)) prev-bb)
          (equal (symbol-name (nth 1 (first choices))) prev-bb))
      (nth 0 (first choices))
    (nth 0 (second choices))))

(defun evaluate-val (val bindings)
  (if (symbolp val)
      (cdr (assoc-equal val bindings))
    val))

(defun execute-phi (stmt init-state prev-bb)
  (let* ((expr (cdr stmt))
         (var (first stmt))
         (val (evaluate-val (choose (cdr (car expr)) prev-bb)
                                    (car init-state))))
    (list (replace-var var val (variables-of init-state))
          (memory-of init-state)
          (pointers-of init-state))))
```

The *init-state* represents the state of a CCDFG before executing ϕ-statement. The function *variables-of* is used to get a list of all the variables of *init-state* with

their corresponding values. *replace-var* replaces the values of the variable var to val in the list of those variables.

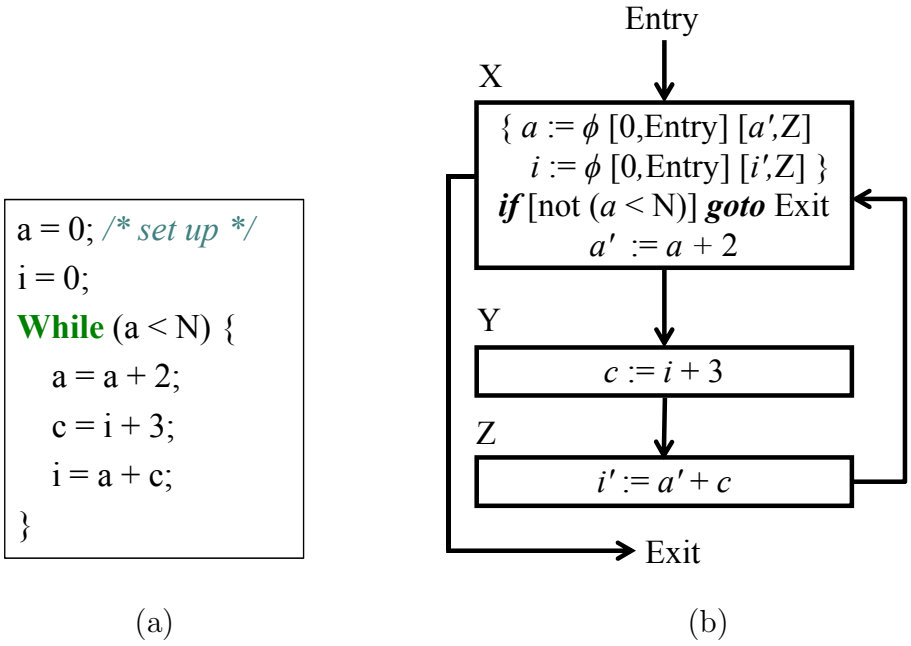
3.2 CORRECTNESS OF LOOP PIPELINING

For the purposes of this paper, a *pipelinable loop* is a loop with the following restrictions [24]:

1. no nested loop;
2. only one *Entry* and one *Exit* block; and
3. no branching between the scheduling steps.

A well-formed pipelinable loop is expected to have only one conditional branch and one unconditional branch. Unconditional branch is at the end of the loop dictating the back edge which enforces that the loop CCDFG is executed again from the first step. Conditional branch ensures that depending on the current value of the exit condition variable, a loop can exit if required. Other intermediate branches have already been handled by compiler and scheduling transformations prior to the pipelining transformation so we need not consider them in our reasoning. There is one ϕ-construct in the first scheduling step which handles the value assigned to loop carried variables depending on whether we are entering loop for the first time or not.

These restrictions are not just meant to simplify the problem, but reflect the kind of loops that can be actually pipelined during behavioral synthesis. For instance, synthesis tools typically require inner loops to have been fully unrolled (perhaps by a previous compiler transformation) in order to pipeline the outer loop.



(a) (b)

Figure 3.1: (a) Loop in C (b) Loop CCDFG before pipelining

Figure 3.1(a) illustrates the C code (ESL description) for a loop. The C code does not have a schedule or the concept of a clock cycle. Figure 3.1(b) shows CCDFG of the sequential loop just before loop pipelining. The loop has three scheduling steps: X, Y and Z. The scheduling step before the loop is *Entry* and after the loop is *Exit*. The edges in the CCDFG indicate the control flow. Note that the sequential CCDFG has Static Single Assignment (SSA) structure , as a result variable a and i are not assigned more than once and we require the quoted variables a' and i'. Note that there is a ϕ-statement in the first scheduling step of the loop. This ϕ-statement accounts for those variables whose values are dependent on the variables evaluated in a previous iteration.

Behavioral synthesis tools use complicated heuristics and aggressive scheduling strategies to find an optimized pipeline interval (clock cycles after which a new iteration can be started such that there are no data hazards). One iteration of

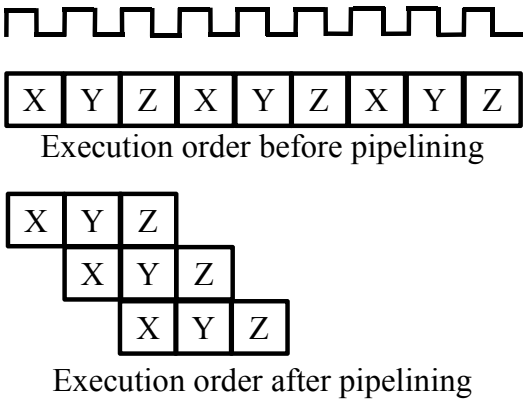


Figure 3.2: Pipelining increases throughput

the sequential design takes three clock cycles. Observe in Figure 3.2 that with the pipeline interval of one, the three iterations of the pipelined loop take five clock cycles as opposed to nine clock cycles in the sequential loop. Loop pipelining reduces the number of clock cycles required to execute the loop, hence this transformation is used by synthesis tools to increase throughput and reduce overall latency.

The main lemma involved in the correspondence proof between the sequential and pipelined CCDFG can paraphrased in English as follows.

> If the pipeline generation succeeds without error, executing the pipelined CCDFG loop for k iterations generates the same state of the relevant variables as executing the sequential CCDFG for some k' iterations. The explicit value of k' is given by the term `(+ (- k 1) (ceil m pp-interval))`.

The theorem can be stated in ACL2 as follows.[1]

[1]The theorem mentioned in the paper does not contain all the hypotheses. Please refer to our proof scripts for final form of this theorem.

```
(defthm correctness-statement-key-lemma
    (implies (and (posp k)
                  (posp pp-interval)
                  (posp m)
                  (equal pp-ccdfg
                     (superstep-construction pre loop pp-interval m))
                  (not (equal pp-ccdfg "error"))))
    (equal (in-order (get-real (run-ccdfg (first pp-ccdfg)
                                          (second pp-ccdfg)
                                          (third pp-ccdfg)
                                          k init-state prev)))
             (in-order (run-ccdfg pre loop nil
                                  (+ (- k 1) (ceil m pp-interval))
                                  init-state prev)))))
```

The theorem involves several ACL2 functions, *e.g.*, `get-real`, `superstep-construction`, etc. For details, one can refer to our proof-scripts. We provide a brief, informal description of some of the critical functions in the theorem. Two key functions that appear in the theorem above are `superstep-construction` and `run-ccdfg`.

The function `run-ccdfg` runs a CCDFG including a pipelinable loop in three parts, first the prologue before the loop, next the loop itself, and finally the epilogue past the loop.[2]

This function is defined as follows, where `prefix` determines the previous

[2]Of course one can have the standard function `run` that executes the entire CCDFG rather than in parts. However, for reasons that will be clear when we define the invariant, in our case it is easier to do most of the work with the execution in three parts and then assemble them into a final theorem about the CCDFG run in the end.

scheduling step of the iteration (required to resolve ϕ-statements).

```
(defun run-ccdfg (pre loop post iterations init-state prev)
  (let* ((state1 (run-block-set pre init-state nil prev))
         (state2 (run-blocks-iters loop state1 iterations (prefix loop)))
         (state3 (run-block-set post state2 nil (prefix post)))
    state3))
```

The function `superstep-construction` combines the scheduling steps of successive iterations to create the "scheduling supersteps" of pipelined CCDFG. If there are data-hazards and pipelined CCDFG cannot be generated as per the pp-interval given, the function generates an "error".

Finally, the function `get-real` removes from the pipelined CCDFG state, all auxiliary variables introduced by the pipeline generation algorithm itself, leaving only the variables that correspond to the sequential CCDFG,[3] and `in-order` normalizes "sorts" the components in a CCDFG state in a normal form so that the sequential and pipelined CCDFG states can be compared with `equal`.

Correctness Statement: Let L be a loop in CCDFG C, and let L_α be the pipelined implementation generated by a pipeline algorithm using pipeline parameters α. Let V be the set of variables in L, and U be the set of all variables in C. Suppose we execute L and L_α from CCDFG states s and s' respectively, such that for each variable $v \in V$, the value of v in s is the same as that in s', and suppose that the state on termination are f and f' respectively. Then (1) for any $v \in V$, the value of v in f is the same as that in f', and (2) for any $v \in (U \backslash V)$, the value of v in f' is the same as that in s'.

[3]The algorithm has to introduce new variables in order to eliminate hazards. One consequence of this is that the new variables so introduced must not conflict with any variable subsequently used in the CCDFG. Since we do not have a way to ensure generation of fresh variables, this constraint has to be imposed in the hypothesis.

Remark: Condition (2) ensures that variables in C that are not part of the loop are not affected by L_α. The value of any new variables introduced by the algorithm in f' are irrelevant since they are not accessed subsequently.

Chapter 4

RESEARCH CHALLENGES

4.1 CHALLENGES ASSOCIATED WITH FORMAL REASONING

To understand the complexities involved in mechanical certification of an algorithm that was not designed originally with certification in mind, we need to re-visit the general approach to applying formal reasoning on software programs. The typical approach is to break the program into a number of pieces, prove key lemmas characterizing the role of each piece, and then chain these lemmas together into a proof of the correctness of the entire program. Crucial to this approach, however, is the requirement that each program piece can be characterized by a succinct invariant that can be easily verified. However, in a program not developed with reasoning in mind, optimizations typically destroy the structural disciplines and modularity of the individual program pieces. This makes it difficult to identify and isolate the components that actually maintain succinct, interesting invariants.

For instance, to prove the correctness statement in the previous algorithm, we want to prove that the complete algorithm follows the invariant that the execution of input CCDFG is equal to execution of the output CCDFG. Since the algorithm is composed of four concrete steps – generate scheduling steps, add shadow register, add edges and data propagation, we intuitively expect the individual steps or at least a combination of steps in sequence to follow this invariant. However, since the algorithm has not been designed keeping theorem proving in mind, that is not the case. For example, if we consider the first step of the proposed algorithm

– **generating new scheduling steps** by overlapping executions of an unrolled loop, we know that the execution of the sequential scheduling steps is not the same as the execution of new scheduling steps unless we prove that there are no data hazards. But, data hazards are not completely eliminated till the last step of the algorithm. Note, that the complete algorithm does follow the invariant as expected, but reasoning about the structure of the complete algorithm at once is not easy.

Our first approach was to certify their implementation as it is using theorem proving. But, our experience was that it is a difficult approach, one that we need not endure. In general, in order to certify such an arbitrary implementation, one has to either (1) restructure the implementation into one that is more disciplined, and prove the equivalence between the two, or (2) come up with very complex invariants that essentially comprehend how invariants from each individual piece are conflated together in the implementation. Both approaches require extensive human interaction, resulting in the proverbial euphemism of proofs of programs being orders of magnitude more complex than the programs themselves [45].

In our work, however, we can "get away" without verifying the specific implementation while still being able to certify the design generated by behavioral synthesis without loss of fidelity. The key observation, as above, is that it is sufficient to develop *any* certifiable algorithm that generates a pipelined CCDFG from a sequential implementation which can be effectively applied with SEC. In particular, any certifiable algorithm that has the same input-output characteristic as the proposed algorithm is sufficient. Thus, our dissertation is on identifying certifiable primitives and invariants of a loop pipelining transformation and developing a pipeline generation algorithm using those primitives, achieving the dual goal of mechanical reasoning of the algorithm and amenability of the resulting reference

model to SEC.

4.2 IMPORTANCE OF USING FORMAL METHODS FOR CHECKING CORRECTNESS

Formally certifying an algorithm gives confidence that the pipelined design is indeed correct. We can claim that if a pipeline loop is created, then there are no additional data hazards which have not been accounted for. Also, since our final theorem proves that executing a sequential loop is same as executing the pipelined loop generated from our algorithm, we can confidently say that our algorithm is complete and data and control flows are well-maintained.

Note that our framework is independent of the inner workings of a specific tool, and can be applied to certify designs synthesized by different tools from a broad class of ESL descriptions. Also, the approach produces a certified reference flow, which makes explicit generic invariants that must be preserved by different transformations. Checking correctness using formal methods prompted us to address the issues lacking in the previous algorithm. To ensure that control flow is maintained, we had to deal with branches. The previous algorithm introduces the concept of Exit edges but does not explain/implement them. The previous authors checked the output of their algorithm with RTL under the assumption that the loop never exits, hence they did not face any issue while testing. However, removing a conditional branch in a loop and furthermore, adding the conditional branch back in the middle of a pipelined loop requires complex reasoning which we manage using one of our primitives, explained in Chapter 5.

Also, the invariant that data flow is maintained at each step enabled us to find a bug in the previous algorithm. The previous algorithm moves a statement to make sure one particular data hazard is removed, but in doing so they move the

statement across a conditional branch statement. Our primitves ensure that such a move is not possible. We have restructred the data propagation step so that instead of going across a conditional branch in the same iteration, the movement of step is now to the previous iteration, explained in Chapter 5.

Chapter 5

OUR APPROACH

As mentined earlier, one of the most complex requirement of verifying behaviorally synthesized pipelined designs is a certified loop pipelining algorithm which can generate a pipeline reference model for industrial strength designs. This pipeline reference model must have a similar structure to the pipelined RTL generated by behavioral synthesis tools such that they can be compared using SEC.

Pipeline synthesis is based on the key observation that execution of successive iterations can be overlapped without affecting execution as long as data and control dependecies are correctly maintained. Thus, the three main activities of a pipeline synthesis algorithm are to (1) identify and remove possible hazards (2) overlap the successive iterations according to the pipeline interval, and (3) ensure proper placement of conditional and unconditional branches. In our case, the identification of data hazards is simplified since the synthesis tool provides a pipeline interval. If we can use this pipeline interval to build our design, then the pipeline reference model is comparable to RTL in abstraction. Thus, instead of *discovering* a pipeline interval ourselves by analyzing read and write variables of every design so that no hazard is introduced, we reuse the provided interval. We have developed a framework of five certified pipelining primitives which allows us, among other things, to prevent possible data hazards. Our framework also provides a primitive to overlap successive iterations and a provision to add and remove branches when required while still maintaining the control flow. We now discuss the framework

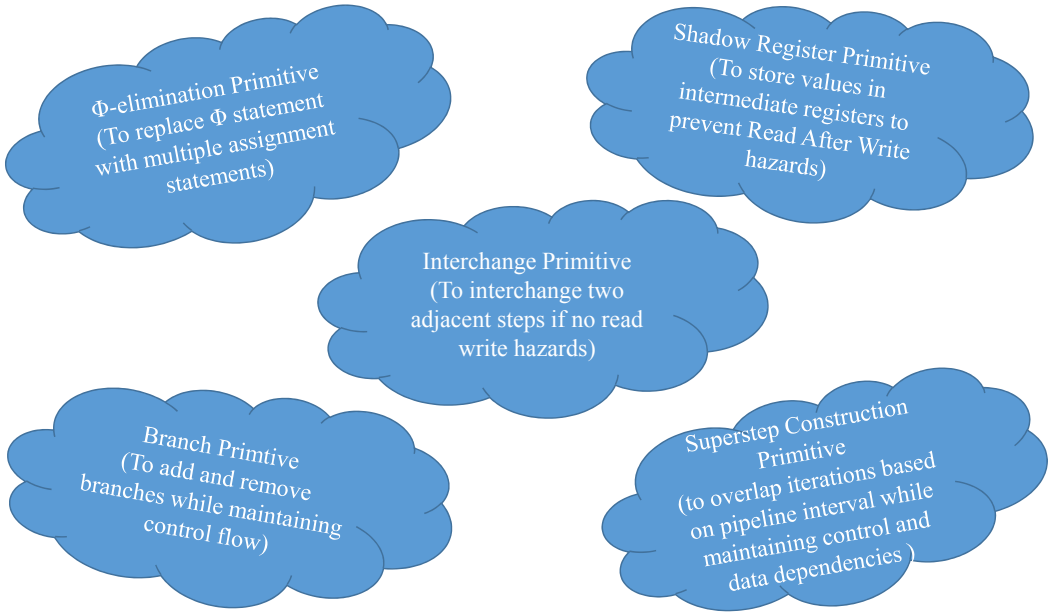


Figure 5.1: Our framework of Certified Primitives

as shown in Figure 5.1 in detail.

5.1 FRAMEWORK OF PROVABLE PIPELINING PRIMITIVES

We believe that the following primitives are necessary and essential in creating any pipelining algorithm in behavioral synthesis.

ϕ-elimination primitive – A ϕ-statement is "`v = phi [`σ `X] [`τ `Y]`", where `v` is a variable, σ and τ are expressions, and `X` and `Y` are basic blocks: while execution, if the ϕ-statement is reached from `X` then it is the same as the assignment statement `v =` σ; if reached from `Y`, it is the same as `v =` τ; the meaning is undefined otherwise. Reasoning about the ϕ-statement is complex since after its execution from a state, say s, the state reached depends not only on the state s but also on previous basic blocks in the execution history. However, we must handle it since it is used extensively in loops to perform different actions depending

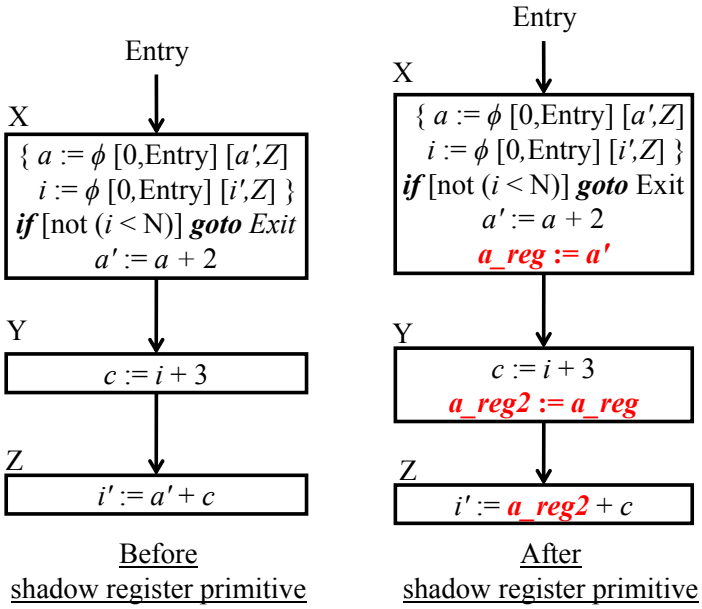


Figure 5.2: Shadow Register Primitive

on whether the loop body is executed the first time. One of the key steps in loop pipelining is, therefore, ϕ-elimination *i.e.*, replacing ϕ-statement with appropriate assignment statements when the previous basic block is explicitely known.

Shadow register primitive – We define a shadow register microstep as simply an assignment statement with symbol expression (x) assigned to a new value (x_reg). We call all the new introduced variables as shadow registers. Intuitively, it is correct that in a sequence of steps, if we assign a variable to a shadow register and replace all occurences of x with x_reg till the next write of x, we should not have made any difference in the execution. Also, since we are not changing the value of x itself, the state after end of execution for both CCDFGs as far as real variables are concerned (all variables excluding all shadow registers) is same. In Figure 5.2, if we assign a shadow register a_reg value of a' at the end of X block, shadow register a_reg2 value of a_reg in Y and replace the read occurence of a'

in Z with a_reg2, the sequential execution remains same. But, because of the addition of these shadow registers, the value of a' is stored in a new temporary variable in every new scheduling step which prevents data hazards.

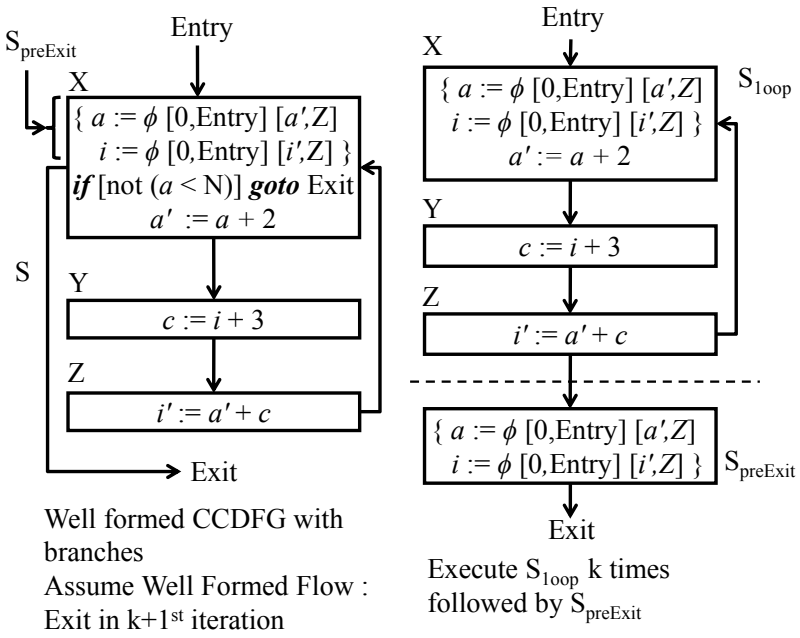


Figure 5.3: Branch Primitive

Branch primitive – Branch instructions are required to determine the control flow. However, reasoning about execution of branch instructions in a loop everytime we apply a primitive can make proof very complex. We note that if we specifically assume that the exit condition becomes true after completing k iterations, then we can remove the conditional branch. To understand the branch primitive (c.f. Figure 5.3), let's assume there is a conditional branch in the sequential loop structure S, which points to either the next microstep in sequence or exits the loop by branching to the scheduling step $Exit$. Let $S_{preExit}$ be the collection of microsteps before this branch in S and let S_{loop} be the corresponding CCDFG loop without the conditional branch. The conditional branch primitive allows us

to replace S with S_{loop} followed by $S_{preExit}$. Similarly, the primitive also allows us to introduce an exit conditional branch by replacing S_{loop} followed by $S_{preExit}$ with S. Note that since k can take any value $k \geq 0$, we are not compromising on the correctness statement. It can be proved that executing S k times such that it exits in the $(k+1)$ st iteration is same as executing S_{loop} k times followed by $S_{preExit}$.

Interchange primitive – Let m and n be two adjacent scheduling steps (or in general, any collection of microsteps) in a CCDFG where both m and n do not have any microsteps containing branch statements. Also, there are no read write hazards between m and n. By read write hazards, we mean that m does not read or write any variable which is written in n and vice versa. Then, the interchange primitive allows us to interchange the order of m and n in the given CCDFG. Note that under the given assumptions, if initial state is the same, then the state reached after executing m followed by n is same as the state reached after executing n followed by m.

Superstep construction primitive – This operation entails combining the scheduling steps of the successive iterations, forming scheduling "supersteps" that act as scheduling steps for the pipelined implementation. Supersteps must account for read-after-write hazards, i.e, if a variable is written in a scheduling step X and read subsequently in Z then Z cannot be in a superstep that precedes X in the control/data flow. Note that we implement data forwarding (forward value of data within a single clock cycle); thus X and Z can be in a single superstep.

5.2 OUR LOOP PIPELINING ALGORITHM

Given a sequential loop S in CCDFG C and pipeline interval I, we can create a pipelined loop P using Algorithm 1. Note that every step of the algorithm is build from ground up using our framework of provable primitives such that the

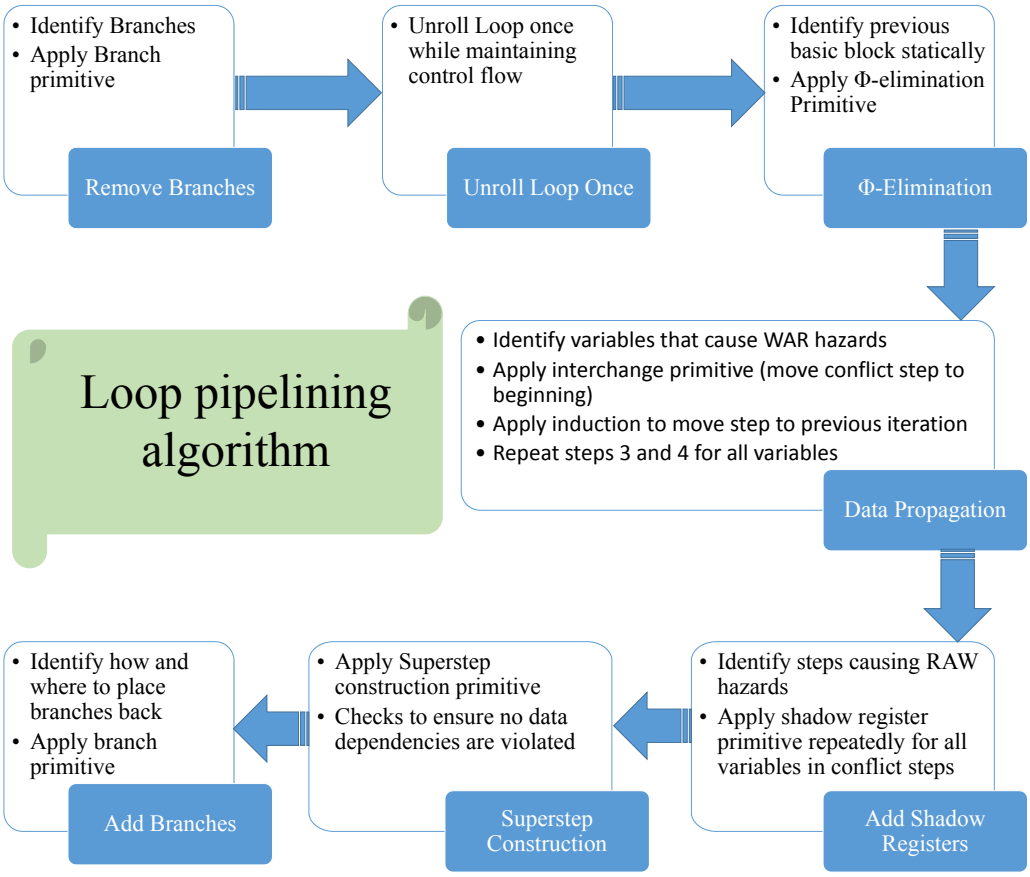


Figure 5.4: Our Loop Pipelining Algorithm (built using primitives)

algorithm can be certified by theorem proving. A quick overview of primitives used in the algorithm at each step are shown in Figure 5.4.

Algorithm 1 Pipelining Algorithm

1: **procedure** PIPELINELOOP(S, I)
2: $S_1 \leftarrow RemoveBranches(S)$
3: $S_2 \leftarrow UnrollLoopOnce(S_1)$
4: $S_3 \leftarrow \phi - Elimination(S_2)$.
5: $S_4 \leftarrow DataPropagation(S_3, I)$.
6: $S_5 \leftarrow GenerateShadowRegisters(S_4, I)$.
7: $S_6 \leftarrow SuperstepConstruction(S_5, I)$.
8: $P \leftarrow AddBranches(S_6)$
9: **return** (P).
10: **end procedure**

Now, we describe the steps to convert a sequential loop CCDFG (c.f. Figure 5.5) to a pipelined loop CCDFG in detail:

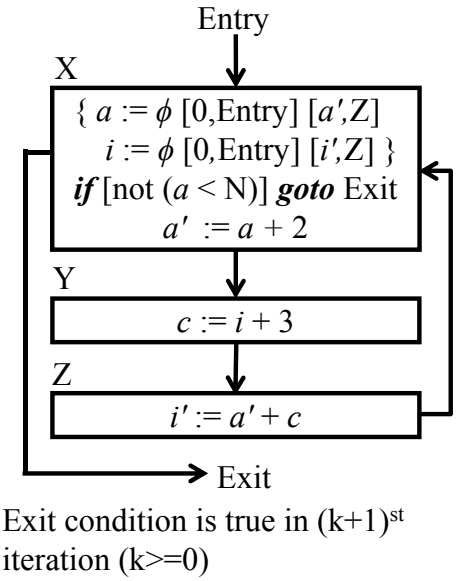


Figure 5.5: Sequential CCDFG with conditional branch

Remove Branches: We apply the branch primitive on S (c.f. Figure 5.5) to remove the conditional and unconditional branch by explicitly defining the control flow in S. The output is a sequence of two CCDFG's S_{loop} and $S_{preExit}$ connected through an edge as shown in Figure 5.6. Note, that S_{loop} does not contain the conditional branch originally present in S. Executing S such that S exits in the $(k+1)$st iteration is same as executing S_{loop} k times followed by $S_{preExit}$. This is possible because the input CCDFG has only one conditional and one unconditional branch as per our definition of pipelinable well formed CCDFG's.

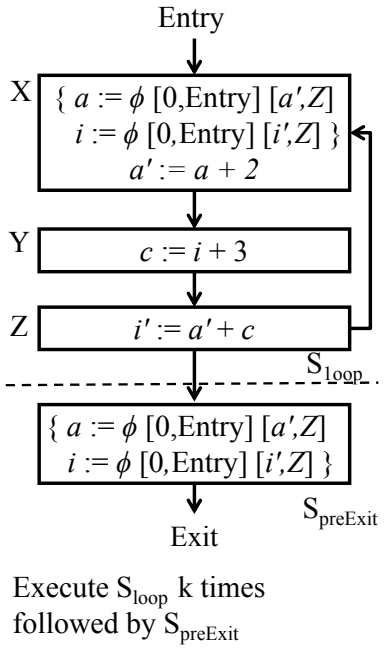


Figure 5.6: Sequential CCDFG without conditional branch. Note the addition of $S_{preExit}$ to explicitely define the control flow

Unroll Loop Once: We have already established that the first iteration behaves differently than the rest of the iterations due to ϕ-construct. So, in this particular step, we simply unroll the loop S_{loop} once. This step does not use any primitive. It is an intuitively correct step, although we also formally verify it

using induction in the final proof. We call the first iteration S_{pre} as shown in Figure 5.7(a).

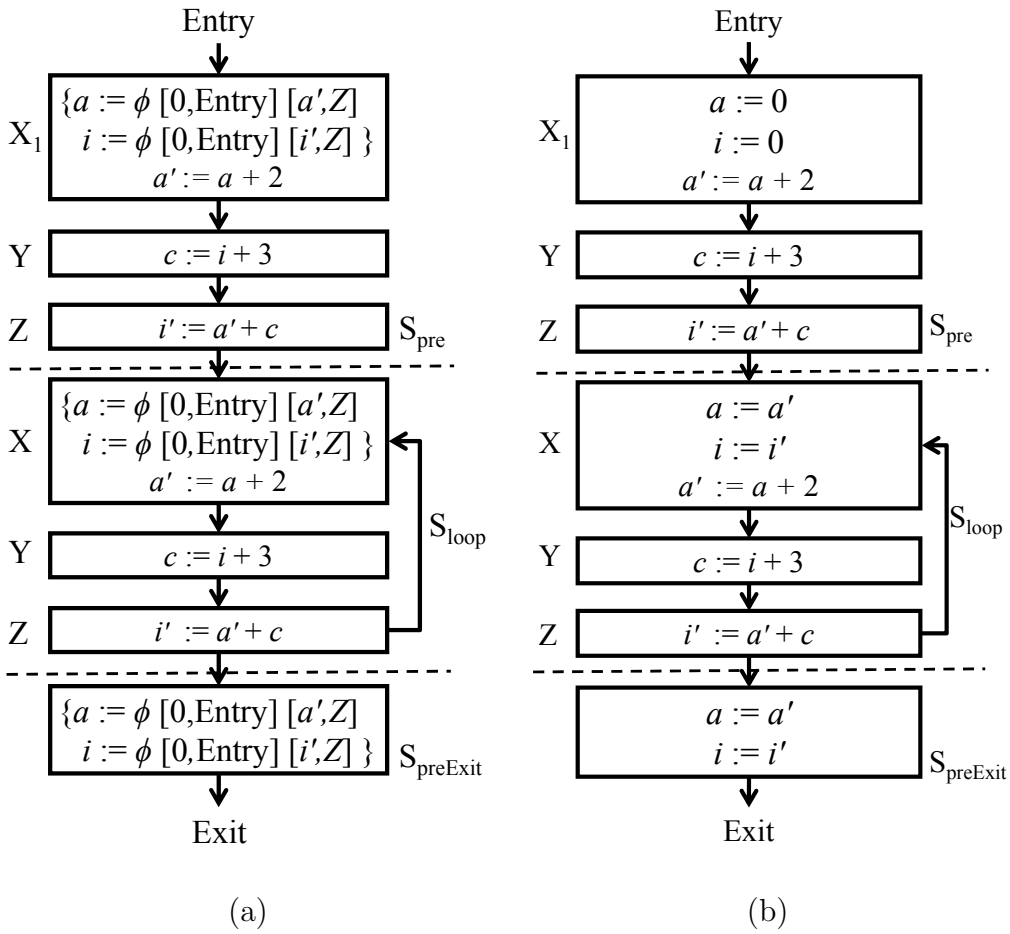


Figure 5.7: (a) Unrolling the loop once to separate the first iteration (b) After ϕ-removal transformation

ϕ-elimination: We apply the ϕ-elimination primitive on S_{pre}, S_{loop} and $S_{preExit}$ to return a CCDFG in which all the ϕ-statements have been replaced with their corresponding assignment statements. Figure 5.7(b) shows the CCDFG after applying the ϕ-elimination primitive. Note that ϕ-construct is only in the first scheduling step of any iteration, so the remaining scheduling steps are the same in all the

Algorithm 2 Data propagation

```
1: procedure DataPropogration(L)
2:     msteps ← GetLoopCarriedDependencies(L)
3:     for each mstep in msteps do
4:         if CheckConflict(L, mstep, N, I) ≠ 0 then
5:             L ← RelocateMStep(L, mstep)
6:         end if
7:     end for
8:     return (L)
9: end procedure
```

iterations.

Data propagation: Algorithm 2 describes how to compute candidates for data propagation across pipeline iterations. It is a critical step in removing data hazards. We want to make sure that when we pipeline a loop, we do not read a variable which has not yet been written. A critical observation is that data propagation is required only for loop carried dependencies. *GetLoopCarriedDependencies* identifies the microsteps where loop carried dependencies are being read. Then, $CheckConflict$ checks whether there would be a conflict when we pipeline the loop. Conflict occurs when the value being read in a microstep is not yet written in the pipelined loop execution. If so, *RelocateMSteps* works in two steps. It first relocates the microstep which reads the variable in an iteration to the starting of S_{loop}. This step can be proved by the interchange primitive since we have already established that the value has not been written yet so there are no read write hazards in between. In the next step, we relocate the microstep to the end of S_{loop}. Note, to maintain the invariant that executing CCDFG before and after this relocation is the same, we need to add the microstep at the end of S_{pre} as well and remove it from $S_{preExit}$. This step ensures that any variable which is being read has already been written. Note that in order to maintain the invariant, only those microsteps can be propagated which exist in $S_{preExit}$, which means only

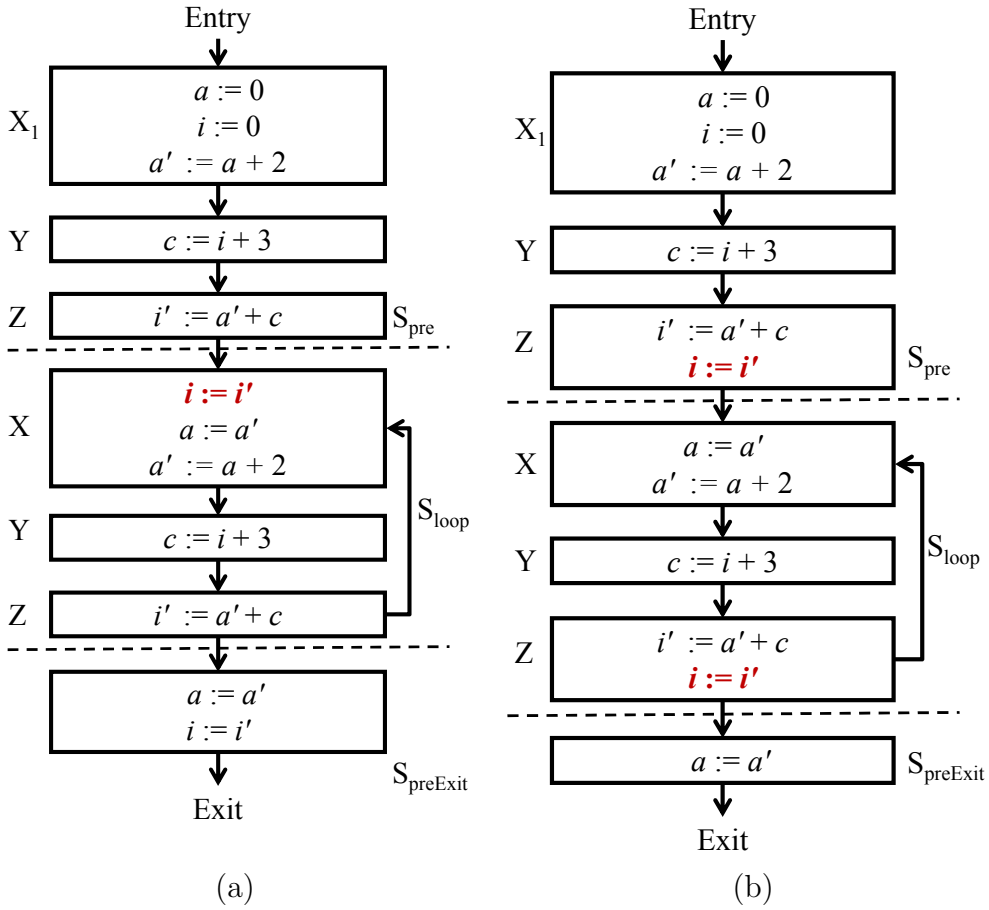


Figure 5.8: (a) Data propagation - first step (b) Data Propagation - second step

those steps which occur before the conditional branch in original CCDFG can be relocated. This ensures that our algorithm does not have the bug which the previously proposed algorithm had. In Figure 5.7(b) we found that the loop carried dependency i' in X would create a conflict when we would move X before Z while pipelining. So, first we relocate the microstep $i := i'$ to the beginning of S_{loop} using interchange primitive in Figure 5.8(a). Then, we move the microstep to end of S_{pre} and S_{loop} and remove the microstep from $S_{preExit}$ in Figure 5.8(b). Note that this preserves execution as explained more in Chapter 6. This step needs to be repeated for every variable found using *GetLoopCarriedDependencies*.

Generate shadow registers: Algorithm 3 inserts shadow registers to prevent variables from being overwritten before being read.

Algorithm 3 Generate shadow registers

```
1: procedure GenerateShadowRegisters(L, I)
2:     V ← GetAllVariables(L).
3:     for each v in V do
4:         w_v ← WriteVariable(v, L).
5:         r_v ← LastReadVariable(v, L).
6:         if RequireShadowRegister(r_v, w_v, I) ≠ 0 then
7:             L ← AddShadowRegister(w_v, L).
8:         end if
9:     end for
10:    return (L).
11: end procedure
```

We first compute all program variables that may be overwritten before being read, which means these are the variables that require shadow registers. To find such variables, *GetAllVariables* first gets a set of all variables. Then, for each variable, we compare the distance (the number of scheduling steps) between the write of the variable w_v (*WriteVariable*) and the last read of the variable r_v (*LastReadVariable*) in an iteration; if the distance is greater than I (pipeline interval), the variable is assigned the new data value of the next iteration before the current iteration's value has been fully consumed; this warrants insertion of shadow registers in every scheduling step between the r_v and w_v. The value is propagated every clock cycle following the CCDFG data flow. We apply the shadow register primitive on the microstep which writes the variable (*AddShadowRegister*). We assign that variable to a new temporary variable called shadow register in every

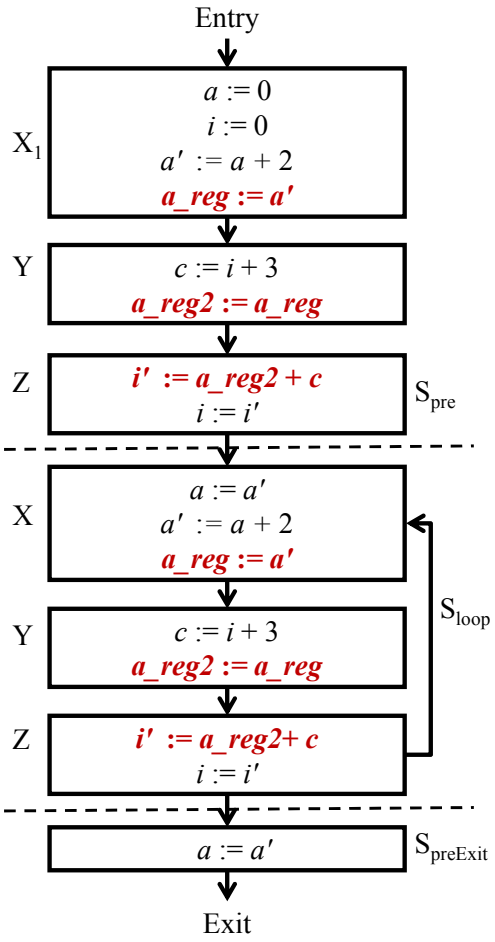


Figure 5.9: After Shadow Register

new scheduling step and replace all subsequent reads of that variable with the shadow register till its next write. In Figure 5.9, we introduce a shadow register a_reg in X and a_reg2 in Y. This step is also repeated for all the variables found using $GetAllVariables$.

Superstep construction: Now that we have removed the data hazards, we can successfully pipeline the loop using the pipeline interval I. We combine the scheduling steps of the successive iterations, forming scheduling "supersteps" that act as scheduling steps for the pipelined implementation. Supersteps must account

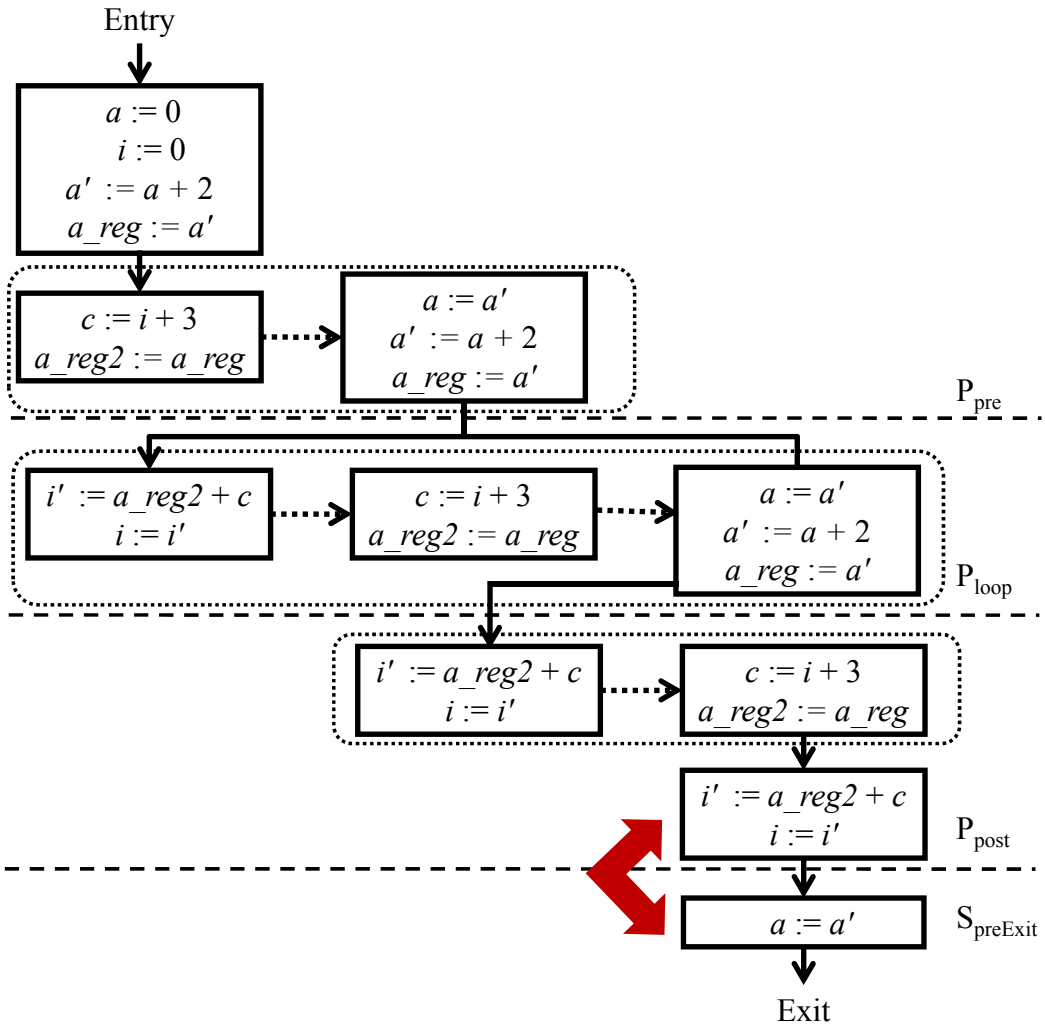


Figure 5.10: After superstep construction

for read-after-write hazards, i.e., if a variable is written in a scheduling step s and read subsequently in s' then s' cannot be in a superstep that precedes s in the control/data flow. A scheduling step is allowed to move up another scheduling step only if there are no intermediate read and write conflicts. Note that we implement data forwarding; thus s and s' can be in a single scheduling superstep. Superstep construction on S_{pre} and S_{loop} creates a CCDFG with three parts: prologue P_{pre},

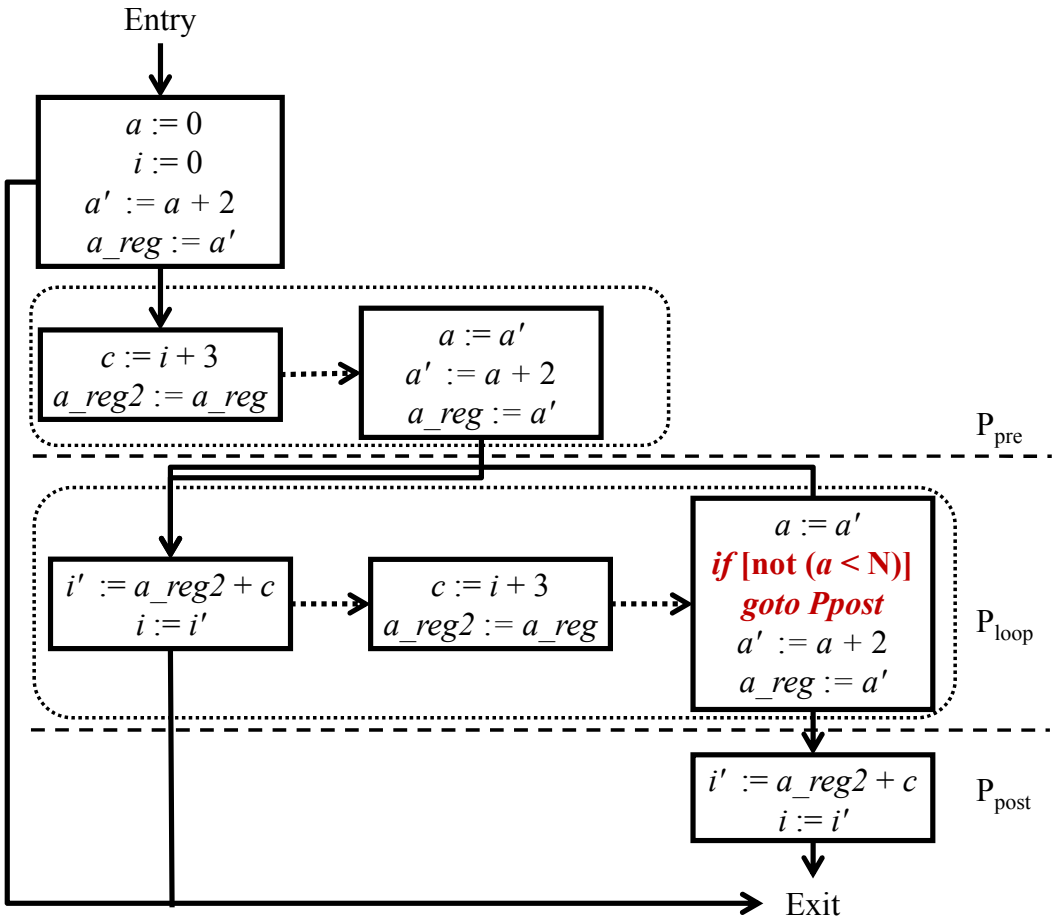


Figure 5.11: Final Pipelined CCDFG

P_{loop} which is the full pipeline stage and epilogue P_{post} as shown in Figure 5.10. We will later prove using our invariant that executing P_{pre} followed by k iterations of P_{loop} followed by P_{post} is equivalent to executing S_{pre} followed by x iterations of S_{loop}, where value of x is determined based on value of k, pipeline interval I and number of scheduling steps in S.

Add Branches: To add the branches back, we use the a combination of interchange primitive and reverse of Branch primitive. Note in Figure 5.10, if there are no read write hazards in between the last scheduling step Z of P_{post} and

$S_{preExit}$, we can interchange them using interchange primtive. Now recall from the branch primitive that if there is a loop structure S_{loop} with a conditional branch, then executing S_{loop} such that it exits in the (k+1)st iteration is same as executing S_{loop} without the conditional branch followed by only those steps from S_{loop} which occur before the branch $S_{preExit}$. Now, we apply the reverse of branch primitive here. P_{loop} in Figure 5.10 is a loop structure without a conditional branch, followed by a collection of microsteps $P_{preExit}$ (here, a collection of Z, Y and $S_{preExit}$). Then, we can add an exit conditional branch in P_{loop} after the microsteps $P_{preExit}$. This branch points to the next scheduling step after the loop P_{post} if the exit condition is true. We can add the conditional and the unconditional branch as shown in Figure 5.11.

We now have the final pipelined loop structure. We describe a proof sketch for the primitives and the algorithm in the next chapter.

Chapter 6

PROOF SKETCH

Certification of our loop pipelining algorithm naturally requires a certification of each of our primitives. In addition, we need to ensure that every time a primitive needs to be applied, the conditions under which the primitive can be applied are maintained. We discuss both aspects below.

6.1 CORRECTNESS OF PRIMITIVES

We must prove that applying a particular primitive is correct, *i.e.*, maintaining a certain invariant. This is proven without considering how it is applied in the context of a pipeline synthesis algorithm. We give an outline of the proof to justify that the primitives are correct.

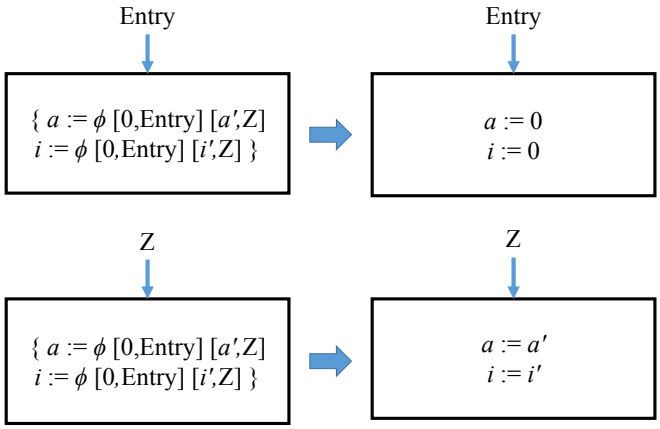


Figure 6.1: Correctness of ϕ-elimination primitive

ϕ-elimination primitive: We prove that the execution of a ϕ-construct is the same as executing the corresponding assignment statements assuming that we already know the previous basic block for the ϕ-construct. In essence, we prove that the algorithm correctly resolves the ϕ to create multiple assignment statements. We induct along the length of each sub-microstep of ϕ-construct and relate it to one corresponding assignment statement as shown in Figure 6.1.

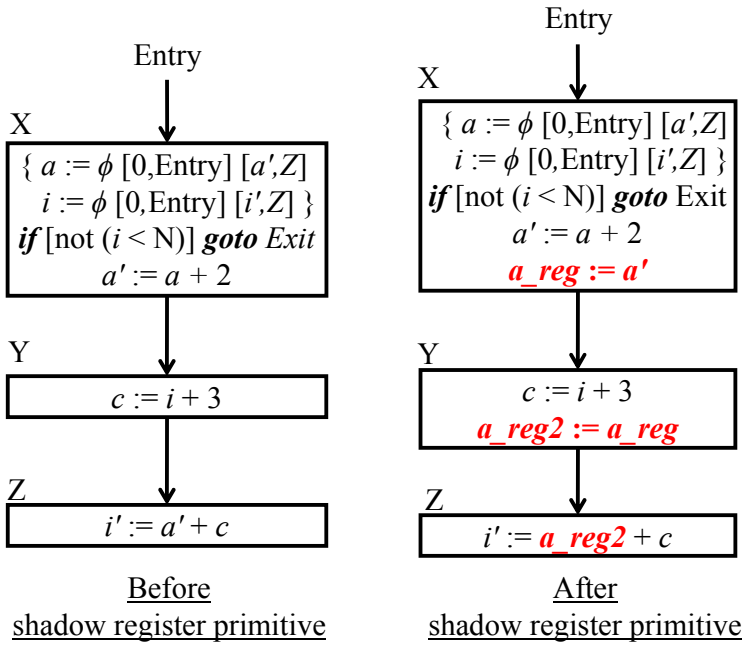


Figure 6.2: Shadow Register Primitive

Shadow register primitive: We prove that adding a shadow register microstep, $a\_reg = a'$ (as shown in Figure 6.2), does not change the value of any variable in the state except the shadow variable. In essence, we prove that if a variable is not written in a microstep, then its value in the state before and after executing that microstep is same. Also, we prove that after executing the shadow register microstep, value of a_reg in the state is equal to value of a'. Furthermore,

since now the value of a_reg is equal to value of a', we prove that executing a statement which reads a' has the same effect on the state as executing a statement which reads a_reg till the next write of a'. This needs to be done for all types of statements *e.g.,* assignment statements (with different types of operations like load, add, mul, getelementptr *e.t.c.*), store statement, branch statement etc. We determine the variables read and written in a statement by analyzing the statements. Note that a_reg is a new variable which is neither written nor read in the given statements.

Interchange primitive: We prove that we can interchange any two adjacent microsteps (excluding branch microsteps) which do not have read-write conflict. We prove that given an initial state, the state after executing microsteps m and n is the same as the state after executing n then m if m and n have no read-write conflict. Suppose, the state after executing m and n is s_1 and that after executing n and m is s_2. We prove that for any variable x, its value remains same in s_1 and s_2. After normalizing the states, we can prove that s_1 is equal to s_2, i.e., the states are the same after executing the two microsteps in a sequence or in an interchanged order. Again, reasoning about read and write of statements involves reasoning about execution semantics of all types of microsteps present in the language which is not trivial.

Branch primitive: We prove that executing S k times such that it exits in the $(k+1)$ st iteration is same as executing S_{loop} k times followed by $S_{preExit}$. (c.f Figure 6.3). We need to define a notion of a well-formed-flow to ensure that we can show that the branch does not exit in the first k iterations. We also need a way to track the backedge along the unconditional branch and ensure that it points back to the beginning of loop S.

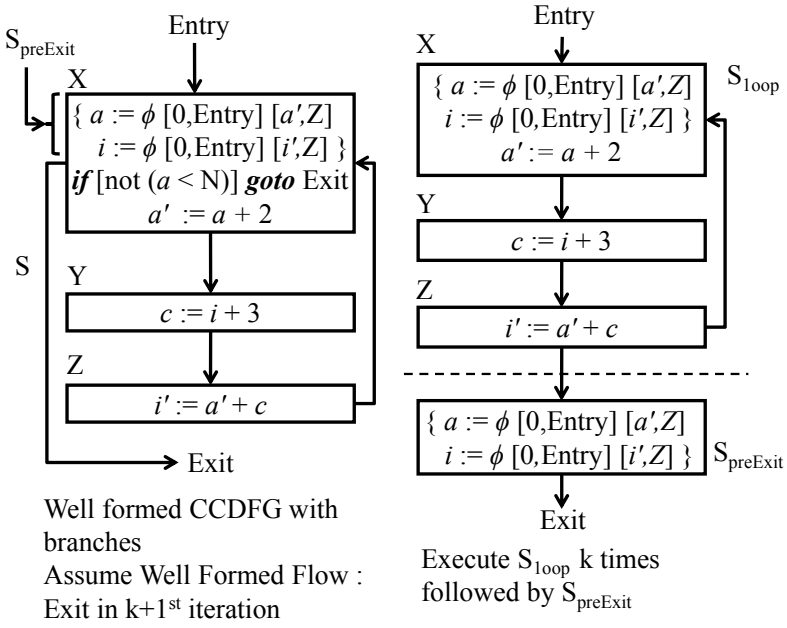


Figure 6.3: Branch Primitive

Superstep construction primitive: This primitive is for overlapping iterations while maintaining data and control dependencies. It is built on interchange primitive but while interchange primitive handles only two adjacent microsteps, superstep construction moves around scheduling steps with multiple microsteps. The interchange primitive is extended by non-trivial induction along the length of the scheduling steps to achieve the desired result. Superstep construction primitive is proved using the interchange primitive and our key invariant described in detail below.

6.2 KEY INVARIANT ON CORRESPONDENCE BETWEEN BACK-EDGES OF SEQUENTIAL AND PIPELINED LOOPS

Our key invariant defines a "correspondence relation" between the back-edges of the sequential and pipelined CCDFGs. The relation can be informally paraphrased

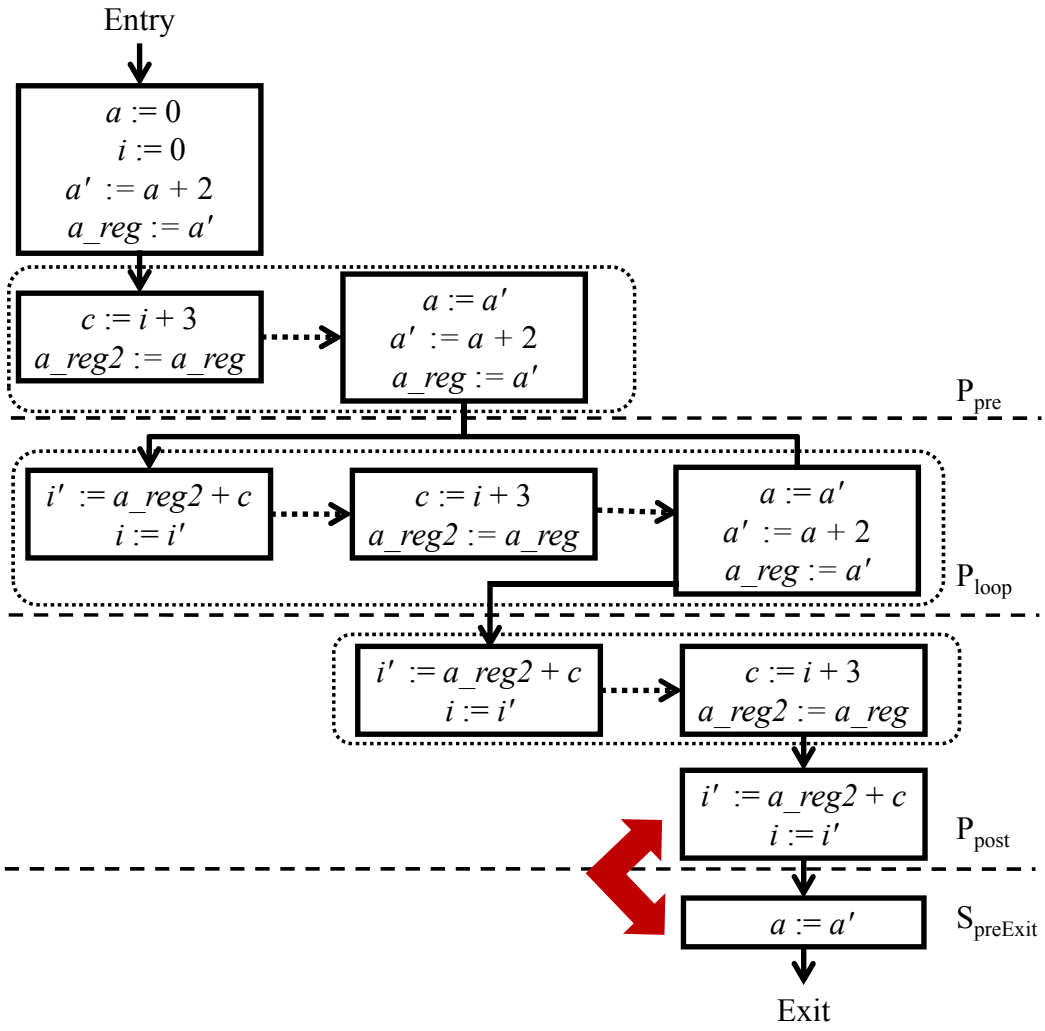


Figure 6.4: Superstep Construction

as follows [50].

> Let S be a sequential loop and P be the pipelined loop generated from our algorithm. The pipelined loop after superstep construction consists of three stages before $S_{preExit}$ as depicted in Figure 6.4: prologue P_{pre}, full stage P_{loop}, and epilogue P_{post}. Let s_l be any state of P poised to execute P_{loop}, and let k be any number such that the loop of P is

> not exited in k iterations from s_l. Then executing P_{pre} followed by k iterations of P_{loop} is equivalent to executing first iteration of S, say S_1 followed by $(k-1)$ iterations of S together with a collection of "partially completed" iterations of S.[1]

The partially completed iterations can be determined by the length of the first iteration in P_{pre} and the pipeline interval. Suppose the length of the first iteration in P_{pre} is `m` and the pipeline interval is `i`. Note that we can calculate the value of `m` based on the number of scheduling steps in a CCDFG and the pipeline interval. The partially completed iterations mean m scheduling steps of S followed by $(m-i)$ scheduling steps of S, by $(m-2i)$ scheduling steps of S, etc. while $(m-ni)$ is positive.

In our example, `m` is 2 and `i` is 1. The invariant implies that starting from the same initial state, executing P_{pre} and `k` iterations of P_{loop} is the same as executing k iterations of S, followed by $m = 2$ scheduling steps of S, followed by $(m-i) = 1$ scheduling steps of S.

As is standard with proofs involving invariants, there are two obligations to prove the correctness, *viz.*, that it is indeed an invariant, and that its invariance is sufficient to imply the desired correctness theorem. Here we give a sense of our envisioned proof.

The proof of invariance of this predicate is, of course, the main "work horse" in this exercise. The proof depends on our interchange primitive which in turn is based on a fundamental idea for pipelining, *viz.*, commutability of independent instructions.

---

[1]The formalization actually characterizes each incomplete iteration, *e.g.*, if the pipeline includes d iterations and successive iterations are introduced in consecutive clock cycles, then the i-th iteration has $i-1$ incomplete scheduling steps.

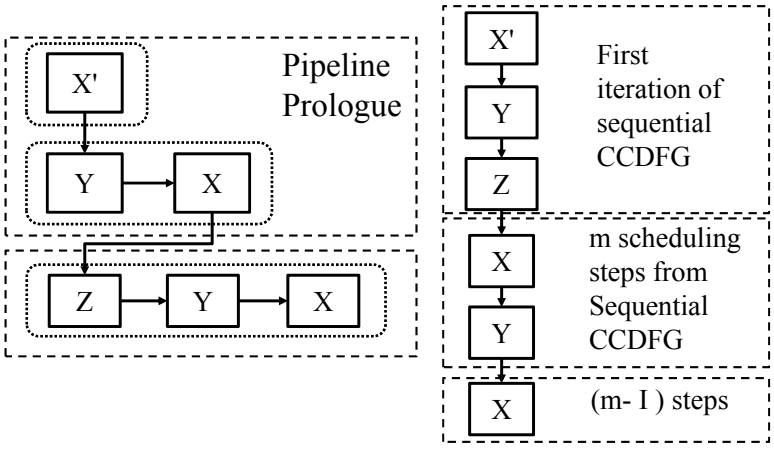


Figure 6.5: Invariant base case where k = 1

> Suppose that the set of variables written and read by two consecutive operations a and b is disjoint. Then executing a followed by b generates the same result as executing b followed by a.

If we view the scheduling steps in Figure 6.5 as arranged in a matrix, then the sequential execution proceeds column-wise along the matrix while the pipelined execution proceeds row-wise. Thus the core proof obligation involves the following two proof requirements.

– Our pipelining algorithm correctly combines the "appropriate" scheduling supersteps which do not have read-write hazards.

– Given that there are no read-write hazards at appropriate places, executing scheduling steps row-wise is same as executing those scheduling steps column-wise in the pipelined CCDFG. This requires the use of interchange primitive.

Although these requirements justify that our correspondence relation is an invariant, they are used somewhat differently in the base case (when the number of iterations k of the pipelined loop is 1) and inductive step (assume the invariant

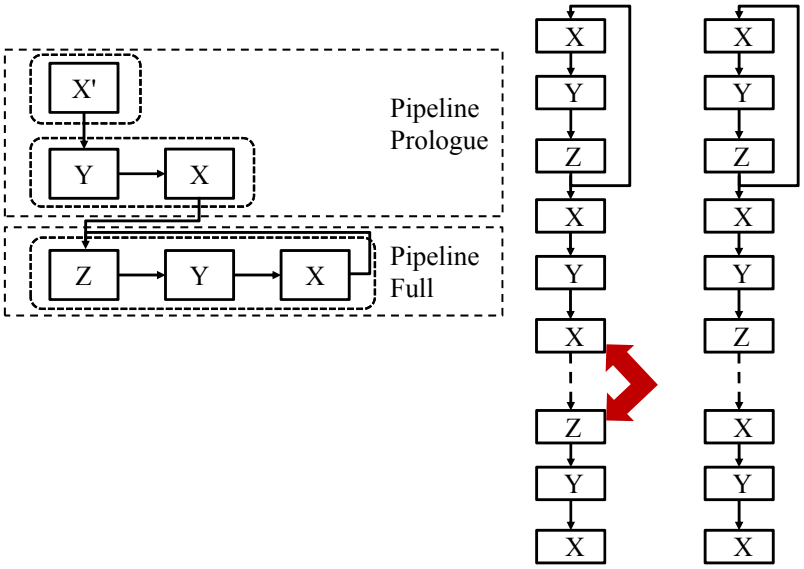


Figure 6.6: Invariant Inductive Step

holds for k iterations of the pipeline and prove that it holds for $(k+1)$ iterations). Their usage is pictorially shown in Figures 6.5 and 6.6. For invariant base case where k is equal to 1, we commute operations in the loop prologue of the pipeline (which corresponds to the first iteration after unrolling) with the loop body. We prove that executing pipeline prologue and one pipeline full stage is the same as executing S_{pre} followed by a sequence of partially completed sequential loop CCDFG. For the inductive step we work with two consecutive iterations of the loop. Assuming that invariant is true for k steps, we prove that executing one pipeline full stage on both sides gives us $(k+1)$ iterations of sequantial loop CCDFG followed by partially completed sequences as expected.

Our invariant is defined specifically to make the proof sufficiency straightforward. Equivalence of CCDFG states of P and S follows from the invariant by noting that the epilogue P_{post} exactly constitutes the incomplete scheduling steps of S specified by the invariant (cf. Figure 6.7).

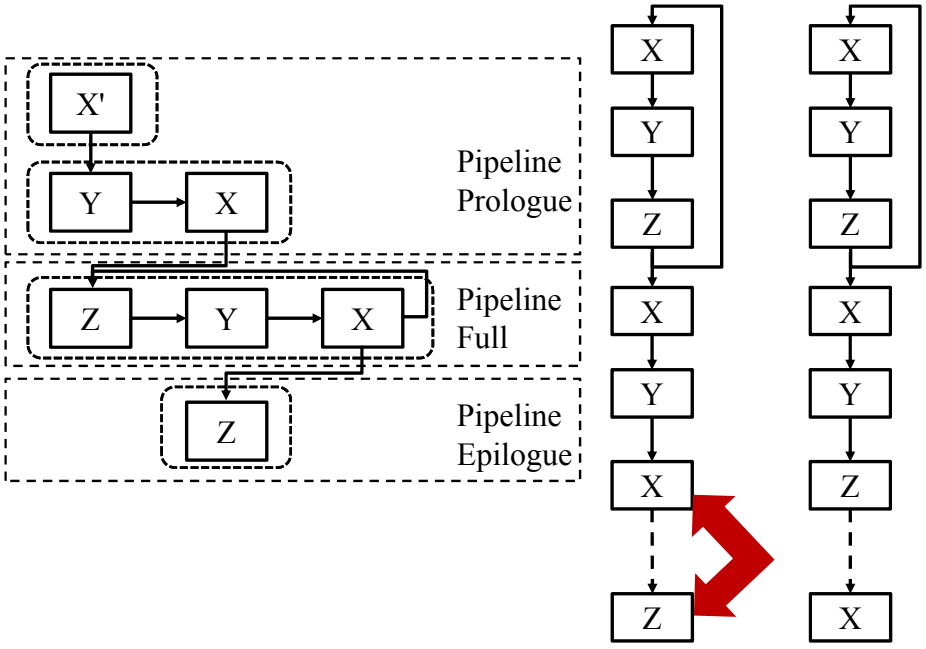


Figure 6.7: Correctness of invariant implies the correctness statement

Our invariant is very different from a typical invariant used in the verification of pipelined machines (*e.g.*, for microprocessor pipelines). We make explicit the correspondence with the sequential execution. The key requirement from a pipeline invariant, *viz.*, hazard freedom, is left implicit and arises indirectly as a proof obligation for invariance of this predicate. Most microprocessor pipeline verification work went the other way. For instance, Sawada and Hunt's invariant [57], expressed through an intermediate structure called MAETT, "tracks" the instructions as they pass through different pipeline stages to ensure that hazards are not introduced. One difference in our case is that we are not working with a concrete pipeline with a fixed set of operations but an algorithm that generates pipelines with an arbitrary sequence of scheduling steps; a construction like MAETT is thus not directly applicable. However, there is a deeper reason for defining our invariant

the way we did. Suppose we simply unroll the loop in the sequential design three times, and then use a technique similar to MAETT to track scheduling steps in this "unrolled loop body" in the pipeline execution. Unfortunately, this does not work, because of the back edge. There is no direct correlation between this edge and any edge in the sequential loop. In fact, it is interesting to observe what its introduction achieves: completion of one scheduling step in each of the three partially executed, overlapping loop iterations. This suggests that the invariant must explicitly capture the state of the executions that have been partially completed during each iteration of the pipeline (*ie*, each traversal of the back edge).

6.3 CORRECTNESS OF OUR ALGORITHM

The algorithm is essentially built from ground-up using primitives as shown in Chapter 5. However, apart from proving correctness of each primitive and our key invariant, we also need to ensure that the primitive is applied by our algorithm properly, *i.e.*, the environment assumptions on which the **correctness of primitive** depends are maintained appropriately by the algorithm at the point where the primitive is applied. The correctness of each primitive discussed above, entails a so-called "assume-guarantee" reasoning: the primitive is guaranteed to maintain the desired invariant if and only if it is applied under certain well-formed conditions. To use these correctness statements to verify the algorithm, we must therefore prove that the algorithm applies each primitive appropriately, maintaining the well-formedness condition required for the correctness of the primitive. Note that verifying this requires an inductive proof relating the states of the CCDFG C' generated after the application of the transformation with the original CCDFG C. Note that the induction is non-trivial because transformations have significant "global" effect on a CCDFG. These include one or more of the following:

1. Replacing one microstep of C with more than one microsteps in C' (*e.g.*, ϕ-elimination), or

2. Interchanging scheduling steps (*e.g.*, interchange), or

3. Changing the variable being read or written in several microsteps (*e.g.*, shadow register)

The final theorem can be written in ACL2 as

```
(defthm run-random-final-theorem
 (implies (and (well-formed-ccdfg c)
               (well-formed-flow-ccdfg bb sub-bb loc c
                                       ccdfg-state prev no_seq_steps)
               (equal (list pre loop post) (final-pp c prev interval m)))
               (not (equal (final-pp c prev interval m) "error"))
     (equal (get-final-real-state
                 (run-ccdfg-random 0 0 0 c ccdfg-state prev no_seq_steps))
            (get-final-real-state
                 (run-ccdfg-random 0 0 0 (append pre loop post)
                                         ccdfg-state prev no_pp_steps)))))
```

The theorem involves several ACL2 functions, *e.g.*, `well-formed-ccdfg`, `run-ccdfg-random`, etc. Please refer our proof scripts for details. Here, we provide a quick overview of some of the critical functions in the theorem below.
c here refers to a sequential loop. We have defined a function `well-formed-ccdfg` which imposes restrictions on the structure of the types of loops which can be pipelined. It ensures that we have only one conditional branch in c which either points to the next microstep or points to `Exit` somewhere outside c. It also ensures that we have only one unconditional branch at the end of the loop which points

back to the beginning of the loop. Moreover, there is only one relevant ϕ-branch which is required to handle variables which change value depending on whether previous basic block is outside the loop or inside. It is required to handle loop carried dependencies as explained before. Also, there is only one place of entry and one place of exit in the loop. Note that these restrictions are seen in behavioral synthesis tools as well since branches inside/outside the loop have already been taken care of before this step using other compiler and scheduling transformations. When we reach the pipelining stage, we have a `well-defined-ccdfg` loop structure dictated by one conditional and one unconditional branch.

We have also defined a function `well-formed-flow-ccdfg`, which states that starting from the initial basic block `bb`, sub-basic-block `sub-bb` and location `loc` of $(0, 0, 0)$ and an initial state `ccdfg-state`, if we encounter a branch within m number of steps, then we do not exit i.e., the exit condition variable is not true. Also, it ensures that the next statement after branch is the next mircostep in order such that we execute the microsteps in order for m number of steps.

The function `run-ccdfg-random` executes a CCDFG starting from the initial basic block `bb`, sub-basic-block `sub-bb` and location `loc` of $(0, 0, 0)$ and an initial state `ccdfg-state`.

The function `final-pp` applies the pipeline generation algorithm to create the list of `pre`, `loop` and `post` as expected.

Finally, the function `get-final-real-state` removes from the CCDFG state, all auxiliary variables introduced by the pipeline generation algorithm itself, leaving only the variables that correspond to the sequential CCDFG. Recall that the algorithm has to introduce new variables in order to eliminate hazards. One consequence of this is that the new variables so introduced must not conflict with any variable subsequently used in the CCDFG. Since we do not have a way to ensure

generation of fresh variables, this constraint has to be imposed in the hypothesis. Also, this function normalizes "sorts" the components in a CCDFG state in a normal form so that the sequential and pipelined CCDFG states can be compared with `equal`.

Following is an English paraphrase of the theorem.

> If the pipeline generation succeeds without error, executing the pipelined CCDFG (a combination of *pre*, *loop* and *post*) for *no_pp_steps* generates the same state of the relevant variables as executing the sequential CCDFG c for *no_seq_steps*.

$$\begin{aligned} no\_seq\_steps &= lenS_1 + (lenS * (\lceil \frac{m}{interval} \rceil - 1)) + lenS_{preExit} \\ no\_pp\_steps &= lenP_{pre} + (lenP_{loop} * k) + lenP_{post} \end{aligned} \tag{6.1}$$

We can take each stage one by one to understand the complexity involved in verifying the algorithm as a whole, over and above the verification of individual primitives.

1. **Remove Branches**: In the *RemoveBranches* stage, which is the first stage of pipelining algorithm, we have to create a correspondence between randomly executing a CCDFG with branches using basic-block, sub-basic-block and location with executing a CCDFG in sequence without a conditional and unconditional branch as shown in Figure 6.8. This is similar to branch primitive, but since we have a non-streamlined run on one side and a steamlined sequential run on the other side, there are theorems involved with finding the next microstep randomly and proving that it is same as the next microstep in streamlined order. We also need to prove that the applicaton of branch

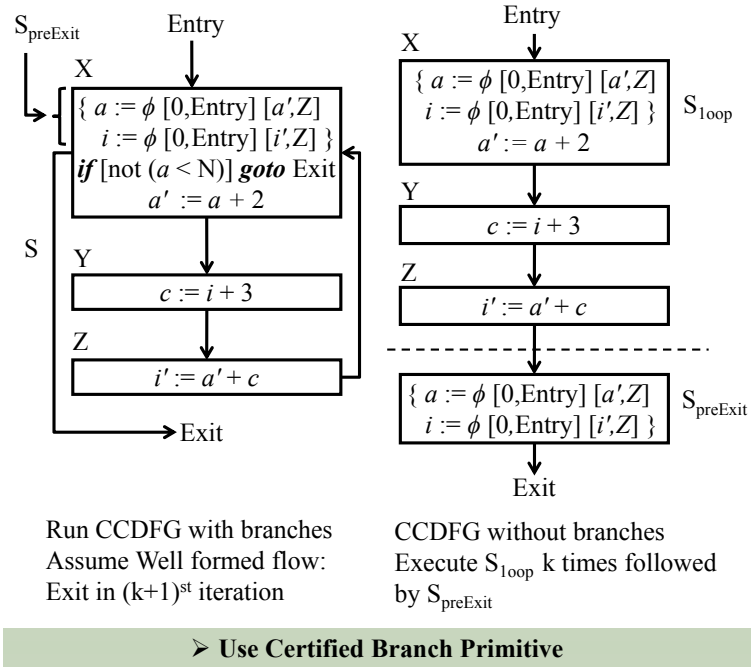


Figure 6.8: Proof Sketch for Remove Branches Stage

primitive is correct. After this step and for all the subsequent steps, we need to show that there are no relevant branches in CCDFG.

2. **Unroll Loop Once**: This step unrolls the loop by one as explained before. The proof uses induction along the number of iterations.

3. **ϕ-to-assign**: In the ϕ-to-assign stage, we replace one microstep of C with more than one microsteps in C' as shown in Figure 6.9. In addition to inductively reasoning about application of a primitive in entire CCDFG, we also have to ensure that addition of new microsteps does not affect the basic structure of the CCDFG. These well-formed-conditions need to be maintained at each step to ensure that the primitives can be applied.

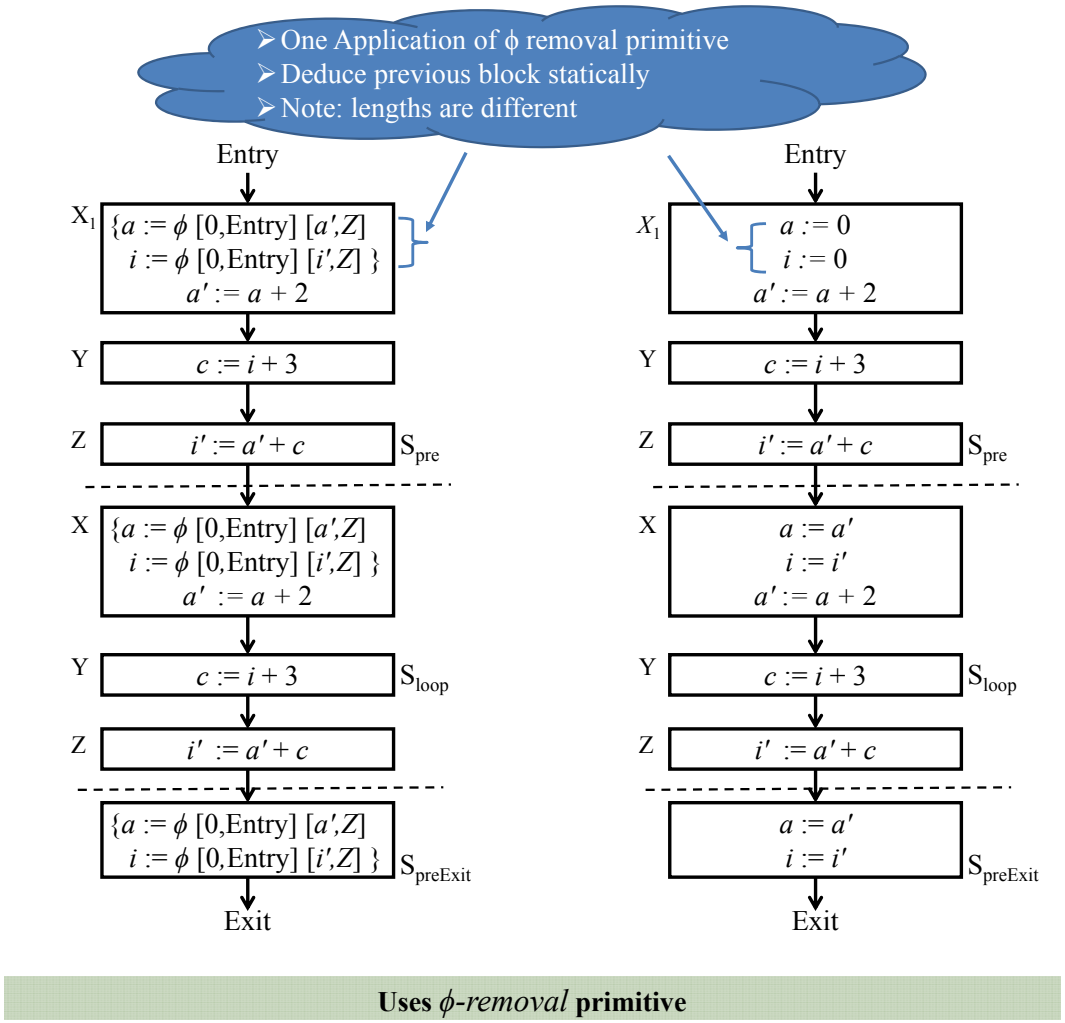


Figure 6.9: Proof Sketch for ϕ-to-assign Step

The upshot is that an inductive theorem relating C and C' must be strong enough to comprehend the global effects. For instance, an inductive statement showing the correctness of ϕ-elimination must account for the fact that the number of microsteps of C is different from that of C'. Thus an execution of C for n microsteps must correspond to an execution of C' for a different number m of microsteps, where the number m is a function of n and the

structures of C and C'; the statement of the correctness of ϕ-elimination must characterize the value of m precisely, perhaps defining functions that statically and symbolically execute C and C', in order to be provable by induction. Furthermore the functions so introduced for static symbolic execution must themselves be proven correct.

4. **Data Propagation** : In the data propagation stage, the first step involves identifying the appropriate statements that cause conflict and applying interchange primitive multiple times to move the microstep to the beginning of the loop. We need to make sure that the conditions under which interchange primitive can be further applied are maintained after each application.

   The second step involves moving the microstep into the previous iteration. It requies removing the microstep, referred as $mstep$ from beginning of S_{loop} and adding it to end of S_{loop}. Also, $mstep$ is added in S_{pre} and removed from $S_{preExit}$. The proof of this step requires non-trivial induction as explained in Figures 6.10 and 6.11. These stages need to be repeated for as many variables as are in conflict.

5. **Shadow-register**: Recall that shadow register step adds many more new statements to assign temporary values to new shadow variables. The addition of new microsteps means that in addition to inductively reasoning about application of a primitive in entire CCDFG, we also have to ensure that basic structure of the CCDFG is maintained. Moreover, we need to reason about read and write of variables across a number of microsteps. The proof is analogous to the proof of shadow-register primitive. However, the primitive is applied multiple times based on the variabes which are causing conflict. This gets tricky as after application of one primitive, there are new variables

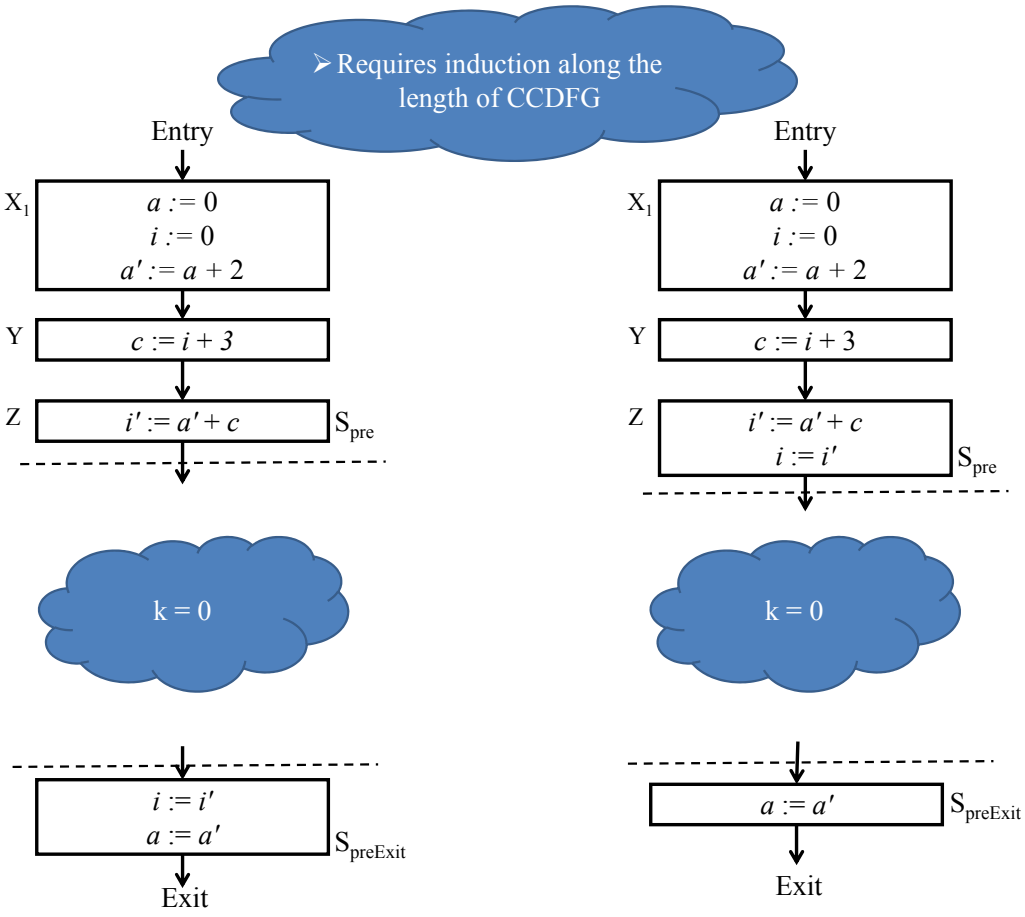


Figure 6.10: Proof Sketch for Data Propagation Step Base Case

introduced and we can only claim that the relevant variables have same value.

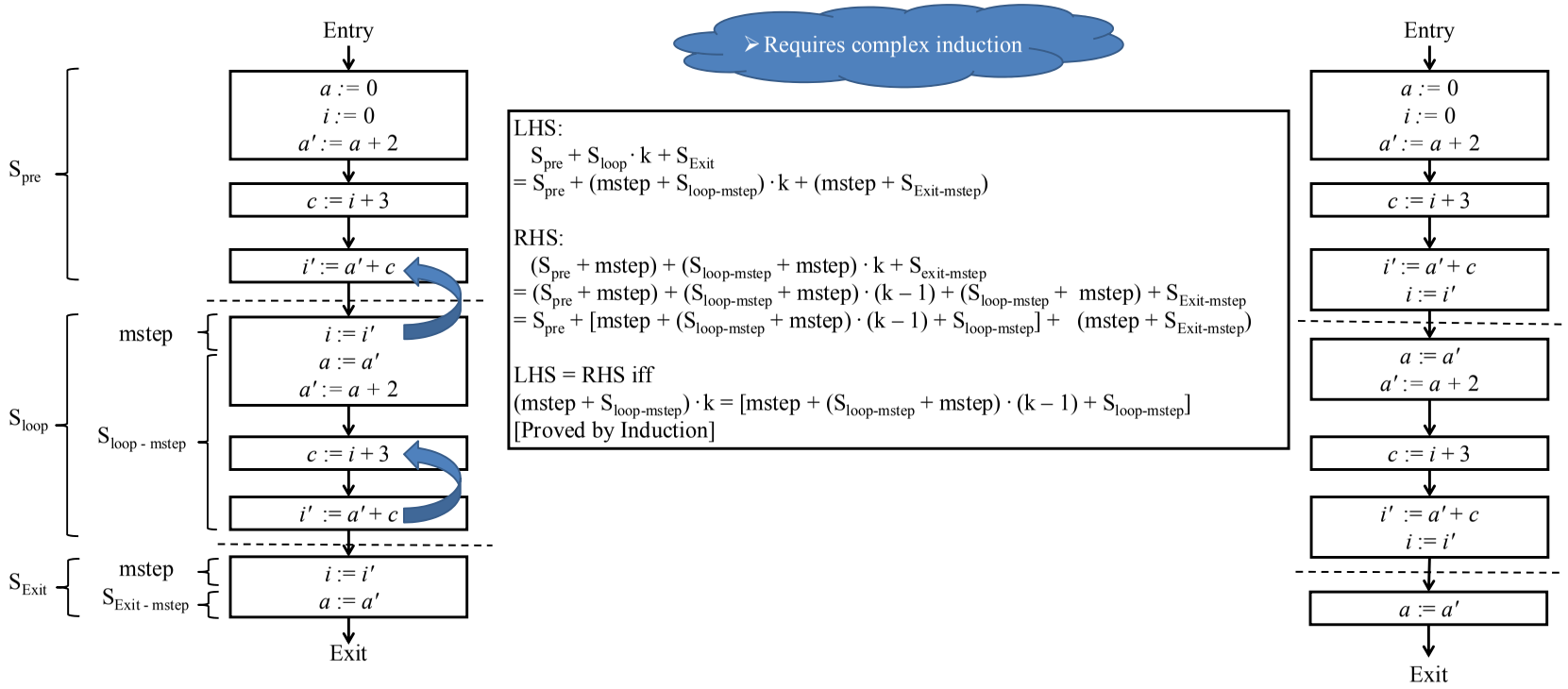


Figure 6.11: Proof Sketch for Data Propagation Step

6. **Superstep-construction**: This step requies proof of invariant and multiple applications of interchange primitive as explained earlier.

7. **AddBranches**: The proof required is the reverse of branch-primitive. However, a key requirement is that branch-primitive can be applied only when we have a `well-formed-ccdfg`, so we need to ensure that the structure of the *loop* before adding branches is such that the final *loop* in the pipelined CCDFG is indeed a `well-formed-ccdfg`.

6.4 LESSONS FROM PREVIOUS FALSE STARTS

Before we came up with our approach of building a pipelining algorithm using a framework of certified pipelining primitives, we tried a few other intuitive approaches. From each false start, we were able to learn something valuable.

In our initial approach we had decided to simplify the problem by ignoring the back edge and proving the correspondence between an unrolled loop and the pipeline. Only after substantially completing this proof and in attempting to extend it to the pipeline with the back edge did we realize that the extension does not work. So, we came up with a key invariant to deal with this problem.

Also, we attempted initially to stick to the previously proposed algorithm and try to prove that the execution of the input is equal to the execution of the output for the complete algorithm. To do that, we need to claim that the output pipeline does not introduce any data hazards. Hazard freedom entails showing the following. "Suppose a variable v is written by a scheduling step S and read subsequently by a scheduling step S' in the sequential CCDFG. Then in the pipelined CCDFG, there is no scheduling step P that writes v and is executed between S and S'." Originally, we defined this notion directly for each variable, *viz.*, with a function

that statically analyzes the CCDFG to identify the range of scheduling steps between a write and subsequent read of each variable. However, this does not work. For example, proving this property for variable x may require a similar property to hold for another variable y (perhaps because x is assigned an expression involving y). But the range of scheduling steps in which x and y are read and written are different, and the extension of the property to all the variables cannot be easily specified by an invariant for any specific scheduling step. When we realized the challenges involved in proving the complete algorithm, it led us to propose our framework of pipelining primitives. Also, our current approach succinctly captures an "on-track property", *viz.*, that the state after k pipeline iterations is equivalent to partial execution of a certain number of iterations in the sequential CCDFG (in addition to completion of k' iterations) which avoids this problem and can indeed be specified as an invariant.

Chapter 7

VIABILITY OF OUR APPROACH

As mentioned earlier, the viability of this approach was tested in [26]. They used a pipelining algorithm to generate a pipeline reference model and compared their pipelined implementation with pipelined RTL using SEC to justify their approach.

If we replace their algorithm with our certified algorithm and still produce the same pipelined implementation with same shadow registers and data propagations, we can claim that our algorithm is also suited for certifying behaviorally synthesized designs.

7.1 EXPERIMENTAL RESULTS

We ran our algorithm on industrial strength pipelined designs synthesized by AutoESL (c.f Figure 7.1).

Design	RTL Lines #	App. Domain	Loop Interval	Loop Depth	No. of operations	Pipeline Register
MemoryOp	291	Memory Operation	1	4	18	2
TEA	383	Cryptography	1	4	28	2
XTEA	483	Cryptography	1	4	37	1
SmithWater	517	Data Processing	2	3	73	0

Figure 7.1: Behaviorally synthesized pipelined designs tested using our algorithm

We have successfully tested the pipeline reference model generated by our certified algorithm with the pipelined reference model generated by the previous algorithm. The test designs are non-trivial to pipeline with varied pipeline intervals and depths and require data forwarding and use of temporary shadow registers to remove data hazards.

7.2 WALK THROUGH OF OUR APPROACH FOR AN INDUSTRIAL STRENGTH DESIGN

To understand our approach, we can go over the steps of one particular industrial strength design.

```
void encrypt (uint32_t* v, uint32_t* k)
{
        uint32_t v0=v[0], v1=v[1], sum=0, i;    /* set up */
        uint32_t delta=0x9e3779b9;              /* a key schedule constant */
        uint32_t k0=k[0], k1=k[1], k2=k[2], k3=k[3]; /* cache key */
        for (i=0; i < 32; i++) { /* basic cycle start */
                sum += delta;
                v0 += ((v1<<4) + k0) ^ (v1 + sum) ^ ((v1>>5) + k1);
                v1 += ((v0<<4) + k2) ^ (v0 + sum) ^ ((v0>>5) + k3);
        } /* end cycle */
        v[0]=v0;
        v[1]=v1;
}
```

Figure 7.2: TEA: C code

Tiny Encryption Algorithm (TEA) [65] is a cryptography design. It is a block

cipher notable for its simplicity of description and implementation with a few lines of code as shown in Figure 7.2. TEA operates on two 32-bit unsigned integers (could be derived from a 64-bit data block) and uses a 128-bit key. It has a simple key usage, mixing all of the key material in exactly the same way for each cycle. Different multiples of a magic constant are used to prevent simple attacks based on the symmetry of the rounds. The magic constant, 2654435769 or 9E3779B916 is chosen to be $232/\phi$, where ϕ is the golden ratio".

The C code is converted to an Intermediate representation IR and undergoes compiler transformations. If we only consider the loop CCDFG, ignoring the paraphernalia before and after this, we have a loop sequential CCDFG just before the loop pipelining transformtion needs to be applied as shown in Figure 7.3. Now, we show how we apply our algorithm to derive a pipelined loop structure from this sequential CCDFG.

Recall that the first step of the algorithm is to remove branches. Assuming that the loop exits in the (k + 1)st iteration, we separate S_{loop} and $S_{preExit}$ as we explained earlier in Chapter 5. We now have a CCDFG as shown in Figure 7.4.

Next, we unroll the loop once to separate the first iteration from the rest. Recall that this step is important so that we can statically determine how to resolve the ϕ-construct. The unrolled loop structure is shown in Figure 7.5.

Next, we resolve the ϕ-construct and replace it with appropriate assignment statements as explained in ϕ-removal step in Chapter 5. Note that the first iteration of the loop has the previous basic block as $Entry$ so ϕ-construct resolution is different than those in other iterations where previous basic block is Z. The CCDFG after this step is as shown in Figure 7.6.

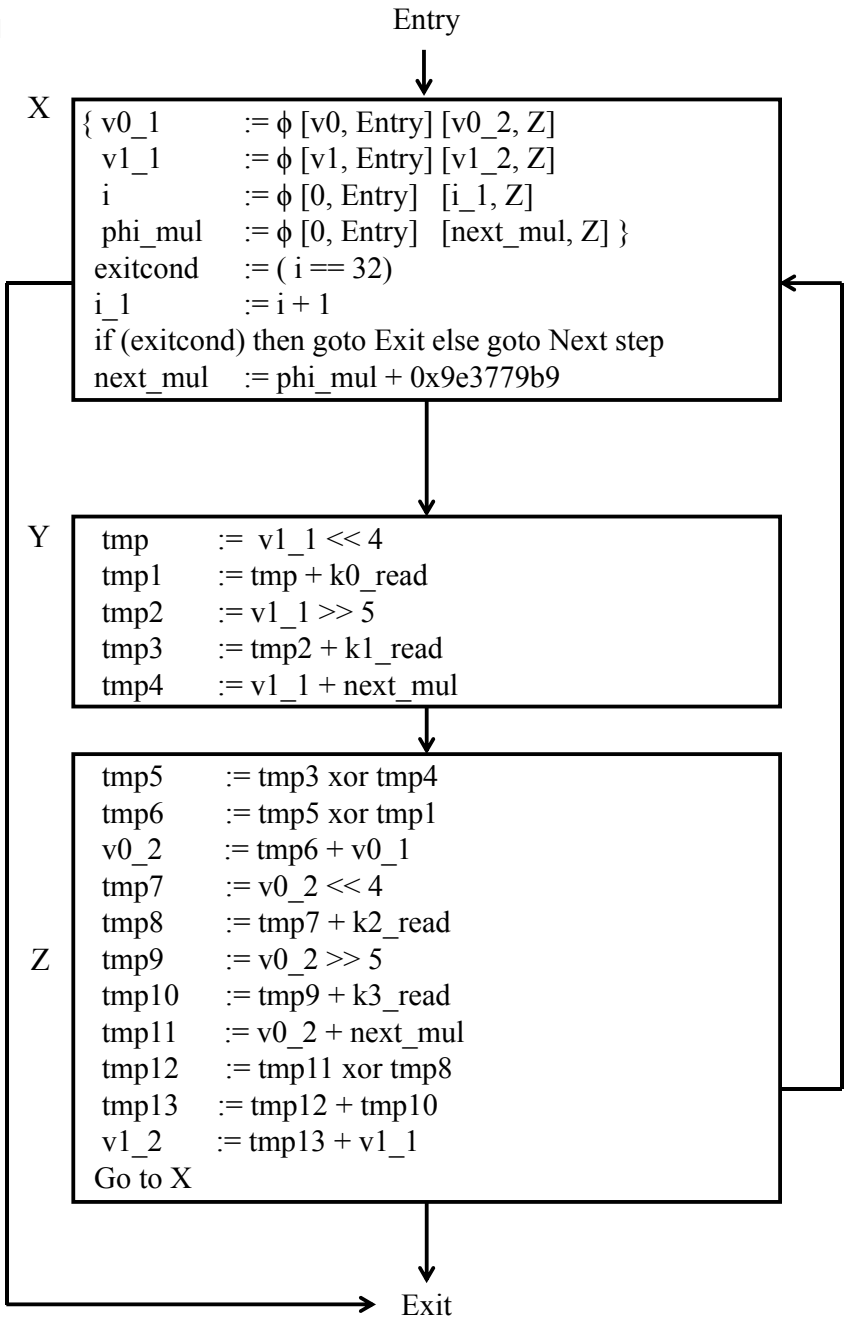


Figure 7.3: TEA: Sequential Loop CCDFG

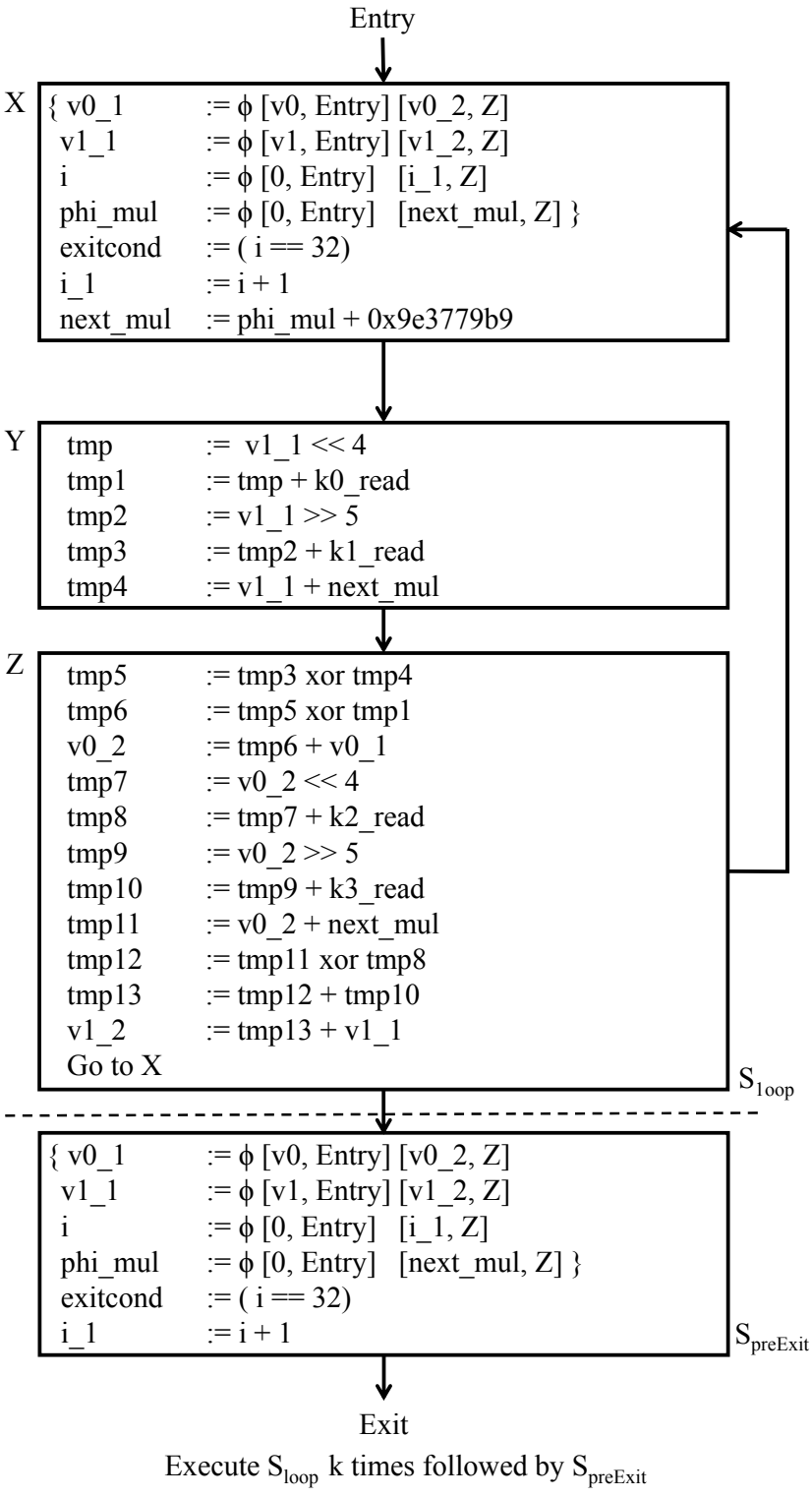


Figure 7.4: TEA: After Removing Branches

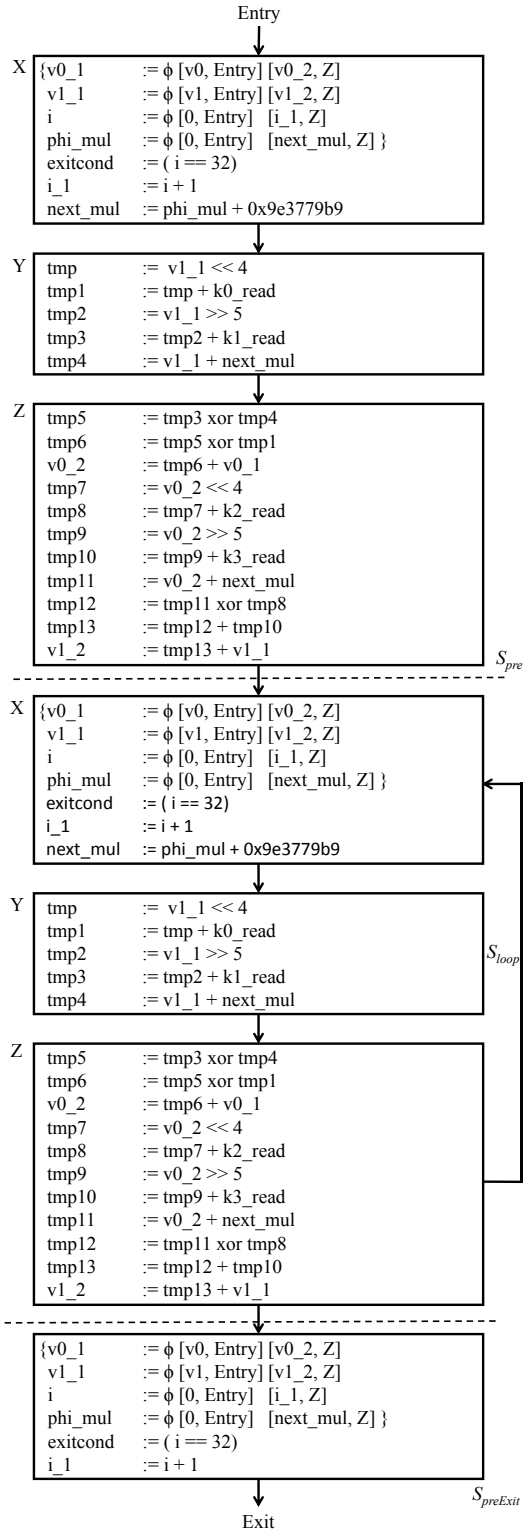


Figure 7.5: TEA: After Unrolling Loop Once

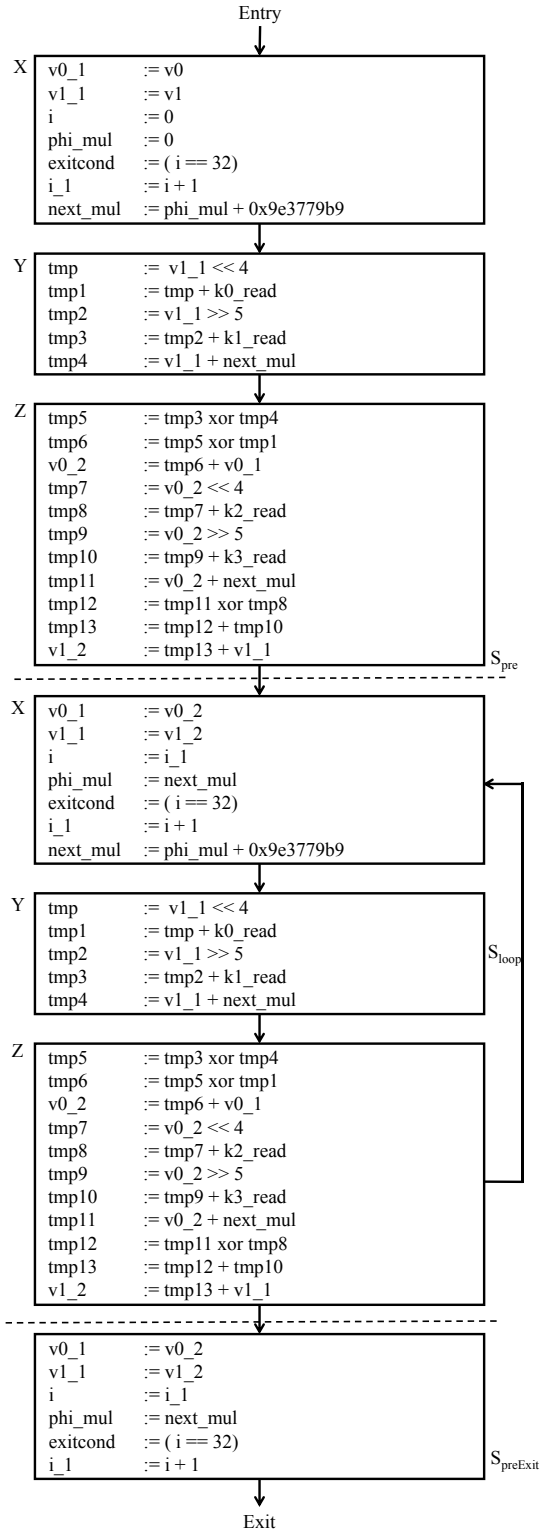


Figure 7.6: TEA: After ϕ-removal

Now, we have to prepare this CCDFG so that iterations can be overlapped. So, we need to remove the data hazards. We first identify the variables/microsteps which cause Write After Read (WAR) hazards. We note that values of v0_1 and v1_1 are read in X and written in Z of S_{loop}. If we overlap the iterations, then X of second iteration will read the outdated values of these variables before they have had a chance to be updated by the Z of the first iteration, thus causing data hazards.

To overcome this, recall that we have data propagation step in Chapter 5 as next step of our algorithm. We implement the first step for $v0\_1 := v0\_2$ and move it to the beginning of the X block in S_{loop} and $S_{preExit}$ as shown in Figure 7.7. Since, we are already at the beginning of S_{loop} here, nothing needs to be done. Recall, this step may need multiple applictions of interchange primitive.

In the second step, we add this microstep to Z of S_{pre} and Z of S_{loop} and remove this mstep from X of S_{loop} and X of $S_{preExit}$ as shown in Figure 7.8. The motivation is to move the microstep to the previous iteration such that the value of any variable is overwritten only when the previous value has already been correctly read. The justification of this step is as explained in Chapter 6 using smart restructing of CCDFG to ease the complexity and application of multiple interchange primitives.

We apply both the steps of data propagation primitive for the second mstep as well, as shown $v1\_1 := v1\_2$ in Figures 7.9 and 7.10.

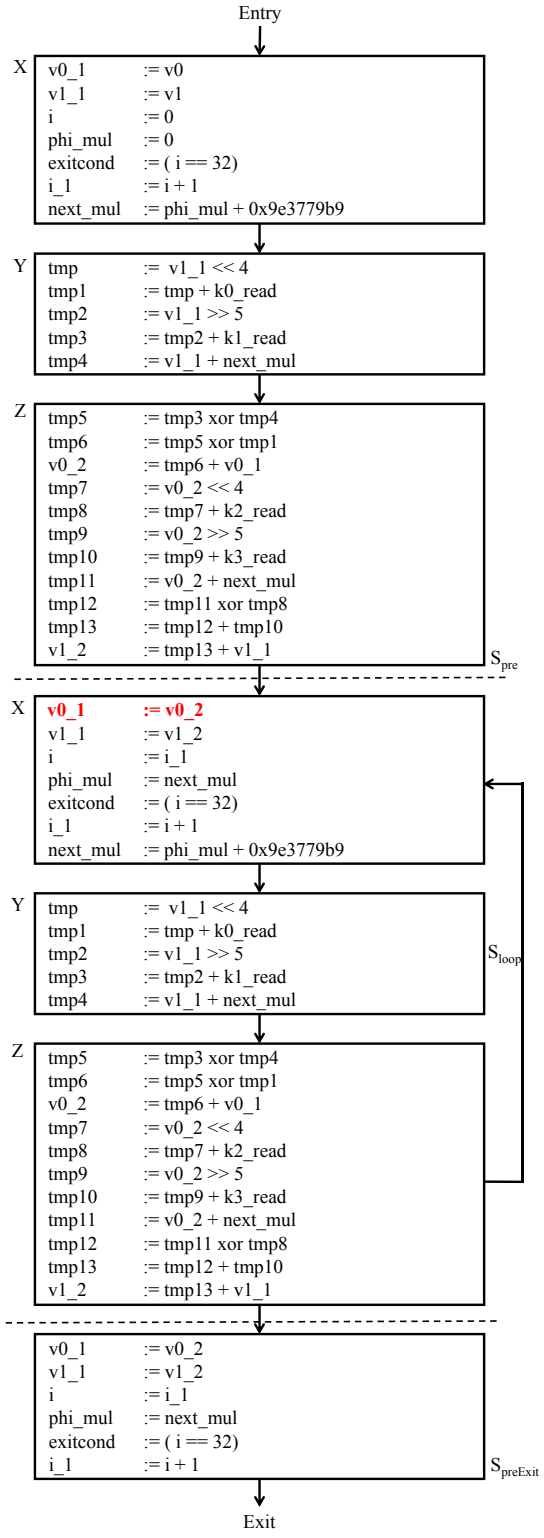


Figure 7.7: TEA: After Data Propagation First Step for $v0\_1 := v0\_2$

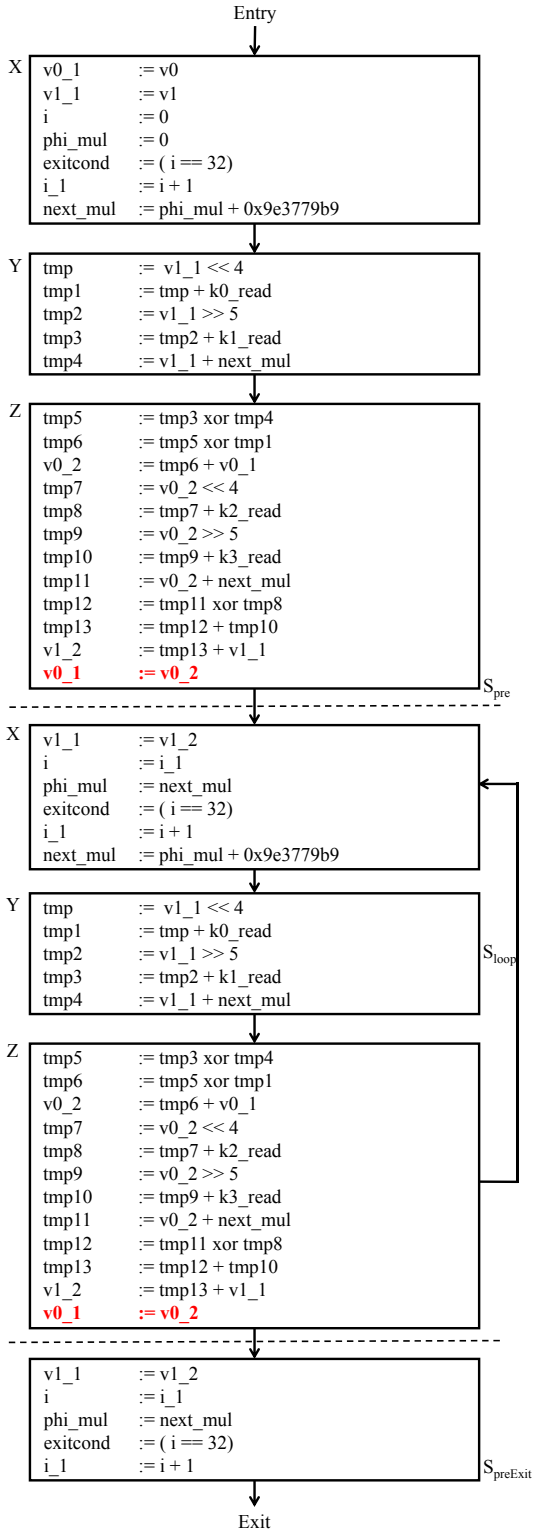


Figure 7.8: TEA: After Data Propagation Second Step for $v0\_1 := v0\_2$

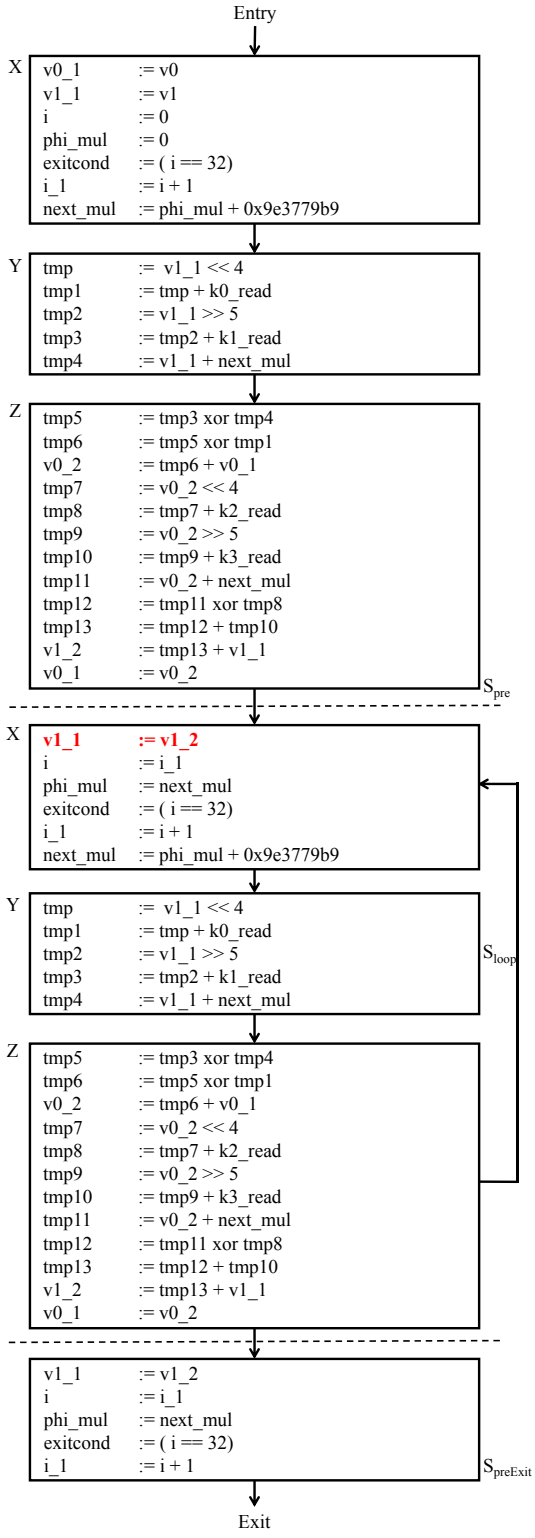


Figure 7.9: TEA: After Data Propagation First Step for $v1\_1 := v1\_2$

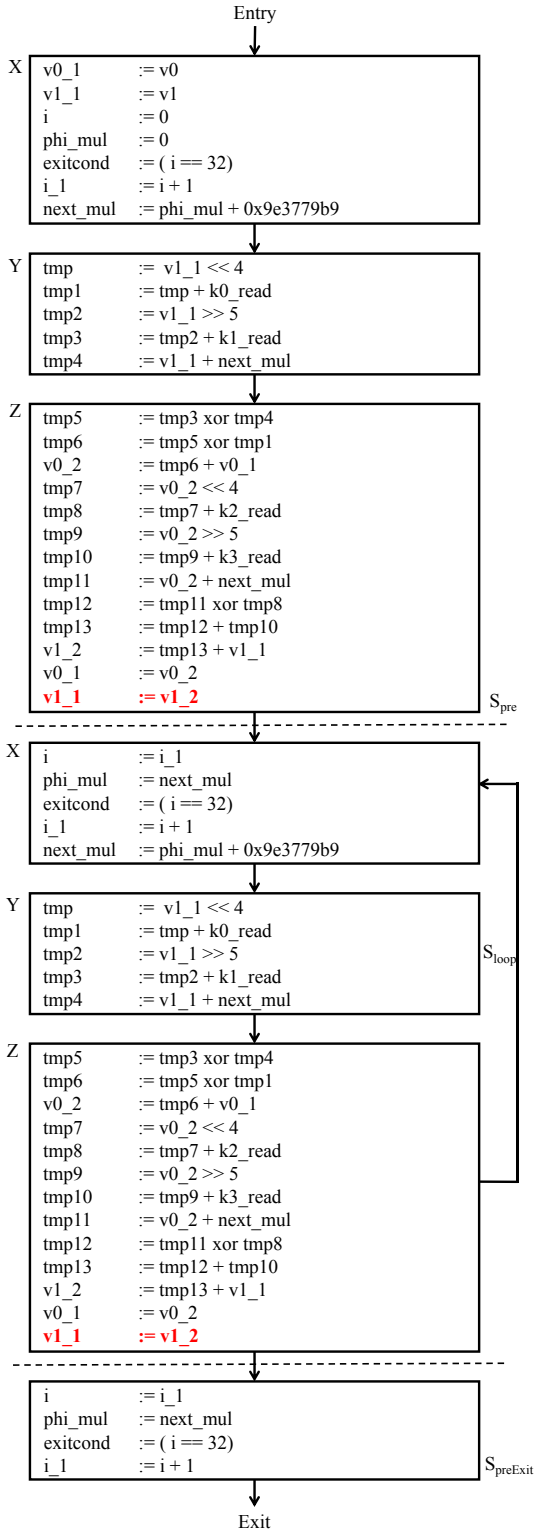


Figure 7.10: TEA: After Data Propagation Second Step for $v1\_1 := v1\_2$

After we have removed the potential WAR hazards which can stall the pipeline, we need to remove the RAW hazards as well. We check the variables which can cause data hazards by measuring the read and write distance between variables and compare it to pipeline interval. For example, in Figure 7.10, we write *next_mul* in X while we read *next_mul* in both Y and Z. If we overlap the iterations as it is, then the X of second iteration occurs before Z of first iteration and we overwrite the value in X before Z has a chance to read it. Note that we know this as our algorithm calculates the longest read and write distance of every variable in an iteration. Here the distance for *next_mul* is 2 scheduling steps, while the pipeline interval is 1. So, we know that this variable will cause a hazard when we pipeline. For all other variables, the distance is either 1 or 0 which is less than the pipeline interval so we know that they are safe.

We store the value of the variable in temporary variables called shadow registers, here we store the value in next_reg, for second scheduling step, we store in next_reg2 and we read from these shadow registers so that the original value is unaffected and can be read as required. The new CCDFG with temporary variables is shown in Figure 7.11.

Now, we can overlap the iterations as shown in Figure 7.12. We call this step - superstep construction.

Now, we need to add the branches back which is the final step of our algorithm. We first interchange the *S_preExit* with *P_post*. Since we have already removed the potential data hazards, we know that we can apply interchange primitives to achieve this step as shown in Figure 7.13. Then, we apply the branch primitive and put the branch back to get the final pipelined structure as shown in Figure 7.14.

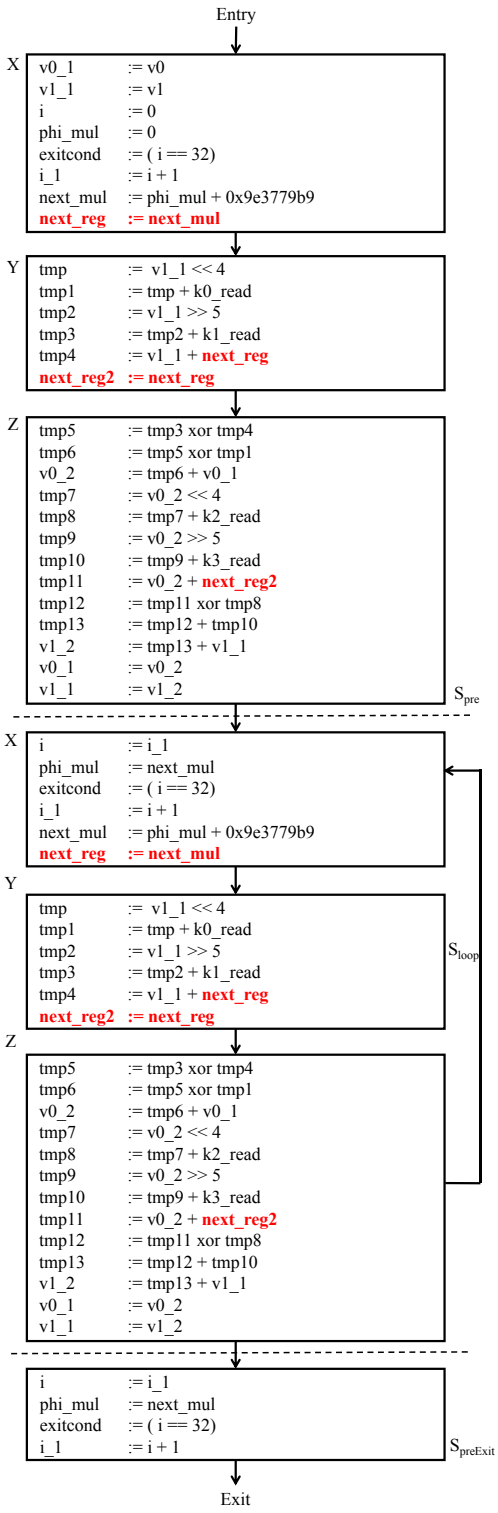


Figure 7.11: TEA: After Adding Shadow Registers

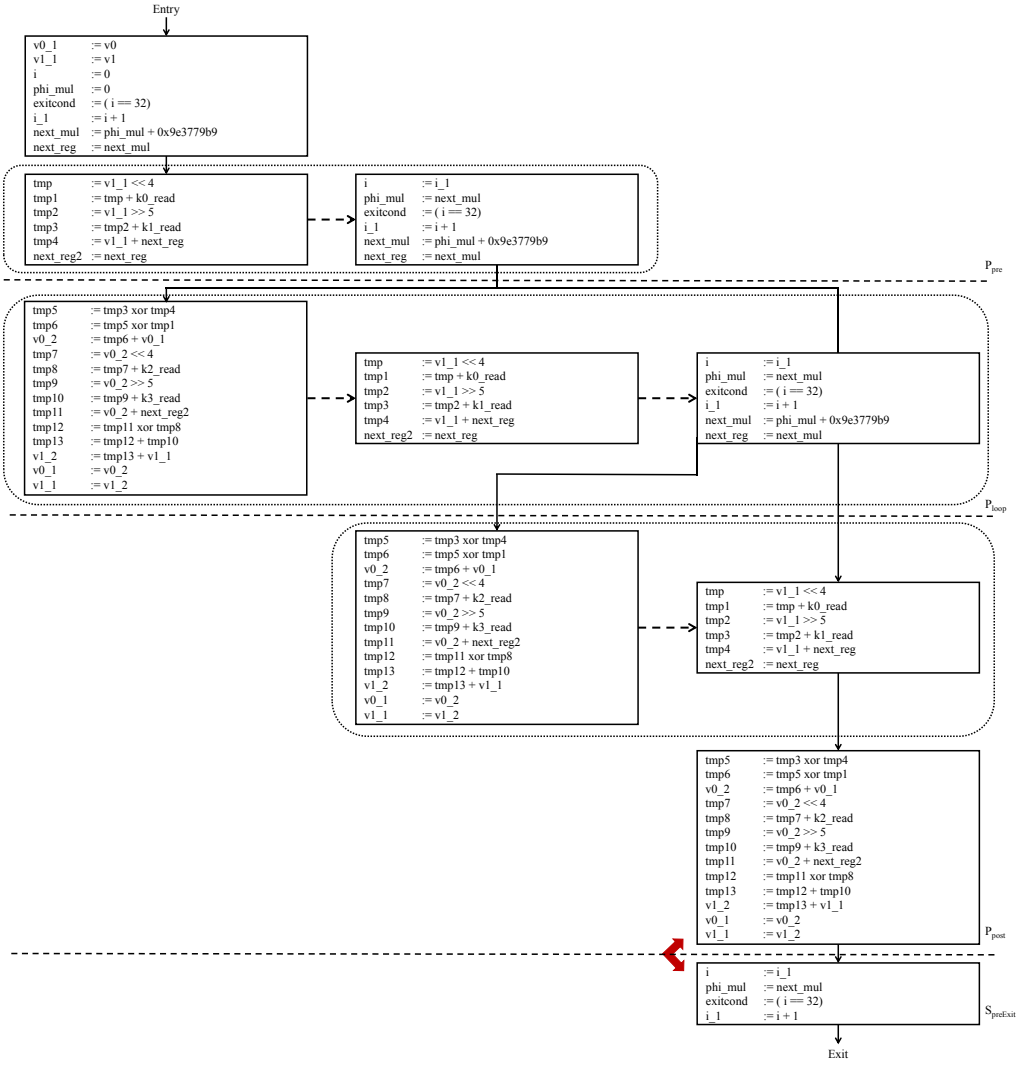


Figure 7.12: TEA: After Superstep Construction

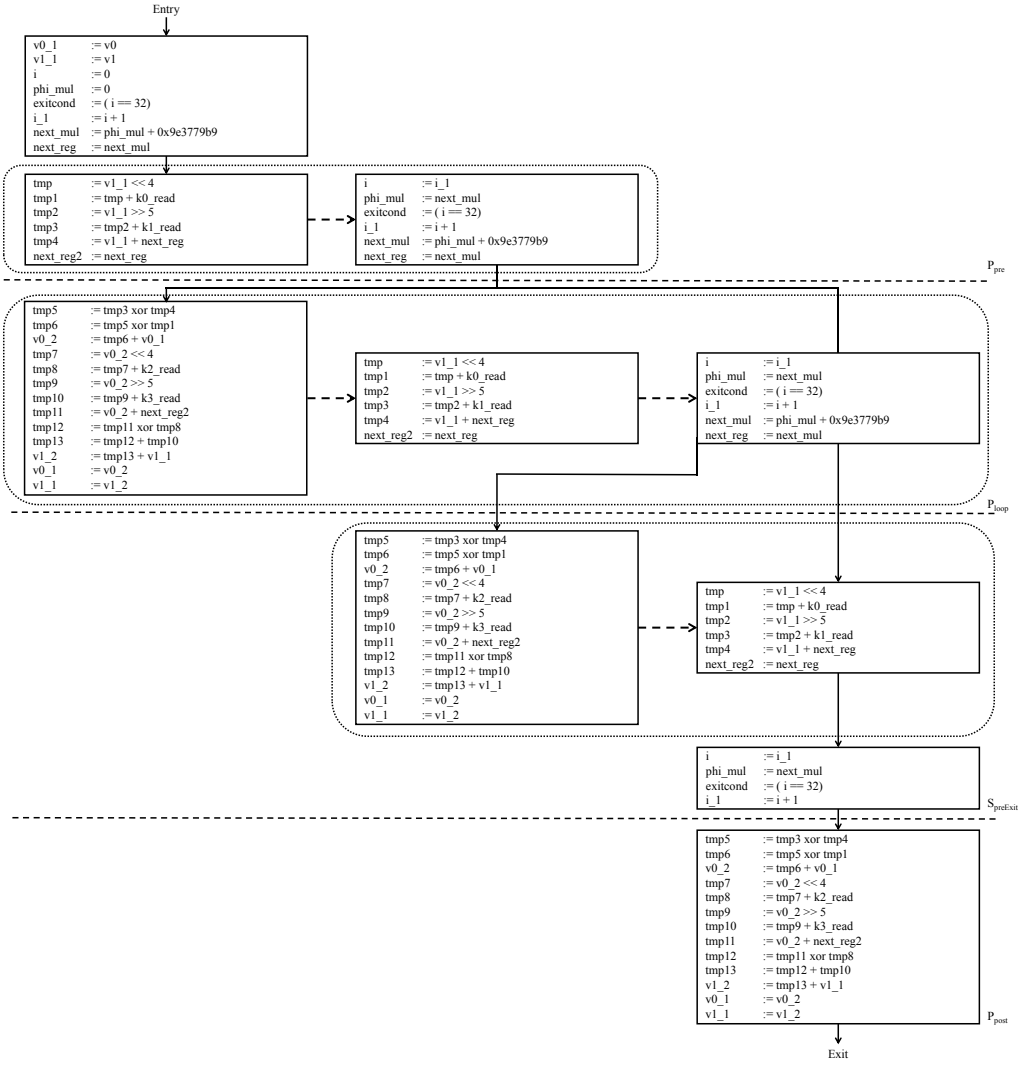


Figure 7.13: TEA: After Interchanging Post with preExit

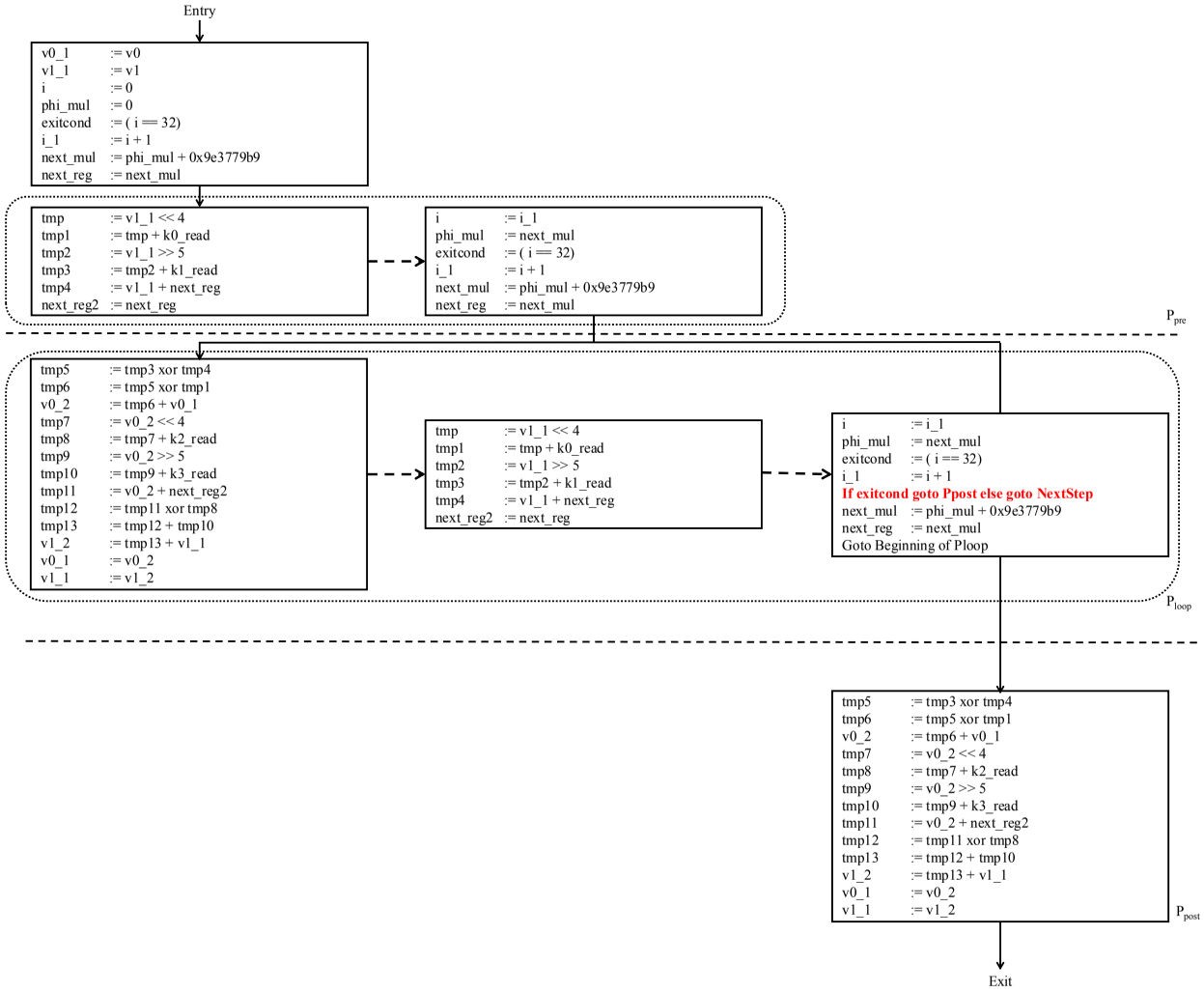


Figure 7.14: TEA: Pipelined CCDFG after adding branches back

As is evident from this example, we have a methodical way of dealing with data hazards and ensuring that we can get a smooth pipeline structure. We have shown that our approach works by testing it on industrial strength designs. The systematic approach to creating a pipelined structure has enabled us to succintly decompose our algorithm into certified primitives and thus certify the overall algorithm.

Chapter 8

RELATED WORK AND NOVELTY OF OUR APPROACH

Besides behaviorally synthesized pipelines, there are mainly two other kinds of pipelines, hardware pipelines and software pipelines.

8.1 HARDWARE PIPELINES AND THEIR VERIFICATION

Hardware pipelining [28] is of two types: *Instruction pipelining* where there is a continuous, overlapped movement of instructions to the processor or *Arithmetic pipelining* where different stages of an arithmetic operation are handled along the stages of a pipeline. An Instruction pipeline has five stages: Fetch, Decode, Execute, Memory Access and Write Back. Without any pipelining, a processor gets the first instruction from memory, undergoes arithmetic operations and then sends it back to memory before starting any new instruction. While pipeline is fetching the instruction, Arithmetic and Logic Unit (ALU) of processor is idle. Pipelining allows the fetching of instructions to be continuous. The next instructions can be fetched even while the processor is performing arithmetic operations, holding them in a buffer close to the processor until each instruction operation can be performed. It reduces the processing time. However, there may be hardware conflicts (structural hazards), data dependencies (data hazards) or hazards that come from branch, jump and other control flow changes (control hazards). These prevent the pipeline from running at full speed. These issues can and are successfully dealt with. But, detecting and avoiding these hazards leads to a considerable increase

in hardware complexity.

There has been a significant amount of work in formal or semi-formal verification of processor (hardware) pipelines. A theorem prover, PVS [49] was successfully used for verification of a simple pipelined processor [16]. Sawada and Hunt [57] presented a technique that models the trace of executed instructions using a table-based representation called a MAETT. These approaches require involvement of user to a great degree, especially in control dominated designs. Hosabettu [30] proposed to build the proof of correctness of pipelined microprocessors by constructing the abstraction function using completion functions. Burch and Dill [6] presented a technique to verify the correctness of the implementation model of a pipelined processor against its instruction-set architecture (ISA) model based on quantifier-free logic of equality with uninterpreted functions. The technique has been extended to handle more complex pipelined architectures by several researchers [59, 63, 64, 46, 62, 60]. ARM2 pipelined processor was verified [32] using abstract state machine. Levitt and Olukotun [43] created a verification method to merge repeatedly last two stages of a pipeline into one, called unpipelining, to ultimately create a sequential verision. Aagaard et al. [3] presented a framework for microprocessor correctness statements about safety that is independent of implementation representation. Out of order pipelines have been verified by combining model checking for the verification of the pipeline control, and theorem proving for the verification of the pipeline functionality [34] .

All the above techniques attempt to formally verify the implementation of pipelined processors by comparing the pipelined implementation with its sequential (ISA) specification model, or by deriving the sequential model from the implementation. There are significant differences in goals and techniques between these efforts and ours. Microprocessor pipelines include optimized (hand-crafted) control

and forwarding logics, but have a static set of operations based on the instruction set. Behaviorally synthesized loop pipelines tend to be deep with a high complexity at each stage, but control and forwarding logics are more standardized since they are automatically synthesized. Furthermore, microprocessor pipeline verification is focused on one (hand-crafted) pipeline implementation, while our work focuses on verifying an *algorithm that generates pipelines*. As explained earlier in Chapter 6 that our invariant is very different from a typical invariant used in the verification of pipelined machines (*e.g.*, for microprocessor pipelines). We make explicit the correspondence with the sequential execution. The key requirement from a pipeline invariant, *viz.*, hazard freedom, is left implicit and arises indirectly as a proof obligation for invariance of this predicate.

8.2 SOFTWARE PIPELINES AND THEIR VERIFICATION

Software pipelining is a form of out of order execution. It is performed by compiler rather than a processor. Behaviorally synthesized loop pipelines are similar in reasoning to software loop pipelines except that since behavioral synthesis is automatic, it is much more streamlined than software pipelines.

In [42], Pnueli and Leviathan present a validator to verify software pipeline in IA-64 architecture [18] (Intel's architecture specifically designed with keeping complexities of software pipelining in mind and to provide additional support for it). It uses rotating register file and predicate registers for its verification. Using symbolic evaluation, the validator generates a set of verification conditions that are discharged by a theorem prover. Kundu et al. [41] use parameterized translation validation to verify software pipelines. They use code motion (rewrite of original loop by validating each rewrite step). Our understanding of hazards and reasoning behind pipelining algorithm is very closely related to recent work on

verification of software pipelines. In particular, Tristan and Leroy [61] present a verified translation validator for software loop pipelines. The loop pipelines in behavioral synthesis considered in this paper are close in structure to software loop pipelines, although our formalization (*e.g.*, CCDFG) has different semantics from the Control Flow Graphs they use, reflecting the difference between eventual targets of compilation (*viz.*, hardware vs. software). However, the fundamental difference is in the approach taken to actually certify the pipelines. Tristan and Leroy's approach decomposes the certification problem into two parts, a "dynamic" part that is certified on a case-by-case basis and a "static" part that is certified in the Coq theorem prover [5] once and for all. The theorem proven by Coq is informally paraphrased as follows:

> Suppose the pipelining algorithm generates a pipeline $\mathcal{P}$ from a sequential design $\mathcal{S}$. Suppose symbolic simulation of $\mathcal{S}$ and $\mathcal{P}$ verifies certain "dynamic" verification conditions (VCs). Then $\mathcal{S}$ and $\mathcal{P}$ are indeed semantically equivalent.

Thus for any pipeline instance $\mathcal{P}$ generated by their algorithm, symbolic simulation is executed between $\mathcal{P}$ and $\mathcal{S}$ to certify that $\mathcal{P}$ is indeed a correct pipelined implementation of $\mathcal{S}$. The dynamic VCs checked by symbolic simulation essentially certify that the pipeline generation did not overlook any hazards.

This is where our work differs from theirs. Our work is expected to provide a single theorem certifying the correctness of the reference pipelined implementation, without requiring further runtime hazard check. Furthermore, their correspondence theorem relates the pipelined implementation with a sequential design with a (bounded) unrolled loop, while our approach certifies the correspondence between the actual Control Flow Graph (CFG) and the pipelined implementation. Indeed, Tristan and Leroy remark that the mechanization of the correspondence

between the CFG and unrolled loop is "infuriatingly difficult". We speculate this is so because they focus on verifying the correspondence between the unrolled loop and the pipeline. In our experience, attempting the formal correspondence between the unrolled sequential loop and pipelined design is indeed difficult since there is no formal way to connect to back edge of the loop with any of the edges in the pipeline. We believe that reconciling this problem and developing a fully certified pipeline generation algorithm would require backtracking from the correspondence with an unrolled loop (and hence translation validation) to a more complex invariant like ours. Of course we must note that we can "afford" to develop a fully certified algorithm in our approach since the pipelines are simpler (cf. Chapter 3); achieving this for arbitrary software pipeline may require further more subtle invariants.

8.3 VERIFICATION OF BEHAVIORALLY SYNTHESIZED DESIGNS

A lot of research has been done in verification of behaviorally synthesized designs. Matsumoto et al. [47] compare two similar C-based hardware descriptions. To verify large C descriptions efficiently, they rely on scanning for textual differences to reduce problem complexity, then enumerate execution paths and apply symbolic simulation and word-level uninterpreted functions. Bounded model checking is used if the software is abritrary. If the software is arbitrary high-level code, then full formal verification is undecidable, but bounded-length verification is possible using symbolic execution. Kroening, Clarke and Yorav [12] apply BMC (Bounded Model Checking) to both a circuit and a C program. Their tool covers arbitrary designs. However, this method shows only the absence of inconsistencies up to a given bound. Furthermore, the number of paths is very high. In order to avoid the state space explosion problem of full formal verification, Jain, Kroening, and

Clarke [11] introduce predicate abstraction for hardware implementations against software specifications. This approach can greatly reduce the size of the state space and verify certain properties for large circuits. The strength of that work is powerful abstraction techniques that reduce the complexity of the software specifications. However, such abstraction techniques can be too coarse, and finding good predicates is highly challenging.

Initially, high level synthesis verification focused on behavioral VHDL [10] and translation from VHDL to dependence flow graphs [35] was verified by structural induction based on CSP semantics [29]. Bisimulation has been proposed as a solution to validate behaviorally synthesized designs [40]. Their approach is implemented for the Spark synthesis tool [20]. However, their approach is not scalable and we handle more complex industrial strength designs. Recently, HOL [22]has been used to synthesize hardware from formal languages automatically. A certified hardware synthesis from programs in Esterel, a synchronous design language, has been also been developed [58] in which a variant of Esterel was embedded in HOL.

There has been much research on sequential equivalence checking (SEC) between RTL and gate-level hardware designs [56, 37]. Research has also be done on combinational equivalence checking between high-level designs in software-like languages (e.g., SystemC) and RTL-level designs [31]. There has also been effort for SEC between software specifications and hardware implementations [66] .

8.4 USE OF THEOREM PROVERS IN HARDWARE VERIFICATION

Thorem provers are widely used for hardware verification. HOL theorem prover [21] has been used in several well-documented projects [13, 23]. ACL2 is also used a lot

in hardware verification [17, 38, 39, 27, 51, 54, 53]. Our project is however somewhat different from the traditional applications of theorem provers. First, since an over-arching goal is to exploit automatic decision procedures, we use theorem proving primarily to complement automated tools. Second, we eschew theorem proving on inherently complex or low-level implementations. Third, interactive theorem proving is acceptable for one-time use, in certification of a transformation, but not as part of a methodology that requires ongoing use in certification of each design. The constraints are imposed by the the environment in which we envision our framework being deployed: it may not be possible to have a dedicated team of experts doing theorem proving as full-time jobs. Finally, the loop pipelining transformation we verify are proprietary to the synthesis tools. Therefore, our approach is targeting verification of transformations which are closed-source (and exceedingly complex), thus making traditional program verification techniques unusable. Our approach shows a novel way in which theorem proving can be applied even under those constraints, in concert with automated SEC.

In addition to technical contributions, we see our work as providing an important methodological contribution enabling use of theorem proving in situations where one needs to certify the result of an implementation on which theorem proving cannot be directly applied either because it is closed-source or because it is highly complex: (1) create a reference implementation, perhaps using as much information as available from the actual implementation, in our case information about pipeline intervals, (2) certify this simpler reference implementation with theorem proving, and (3) develop an SEC framework to compare the result of the reference implementation with that of the actual implementation. In addition to making theorem proving applicable on industrial flows without requiring us to certify industial implementations with their full complexity, this approach permits

adjusting the algorithm (within limits) to suit mechanical reasoning while still affording comparison with actual synthesized artifacts. We have made liberal use of this "luxury", *e.g.*, we have been continually redefining our superstep construction function to facilitate proof of key structural lemmas of the invariant before settling on the final version. We believe similar approach is applicable in other contexts and may provide effective use of theorem proving within industrial verification flows.

Chapter 9

CONSLUSION AND FUTURE WORK

9.1 SUMMARY

Our dissertation is on developing a framework of certified pipelining primitives for building certified pipelining algorithms. We build a loop pipelining algorithm using this framework and certify it using ACL2 theorem prover. We have formalized the syntax and semantics of our intermediate representation (CCDFG) in ACL2. We have successfully identified and formalized a framework of succinct and provable primitives essential for loop pipelining algorithms. These primitives include ϕ-elimination, shadow-register, interchange, branch and superstep-construction primitive. We have formalize a key invariant, unlike used before for any microprocessor pipeline verification, required for the correspondence between the sequential loop with the backedge and the pipelined loop with the backedge. We have proved that the corresponding relation is true for our algorithm and we have proved the implication chain from this relation to the correctness statement for our algorithm. Using these certified primitives as building blocks and our key invariant, we have formalized and certified a loop pipelining algorithm. We have proved that each component of our algorithm described in Chapter 5 maintains the invariant that the execution of CCDFGs before and after that component is same. Even though each component essentially decomposes into proving that our primitive is correct, we still have to prove that every application of our primitives maintains certain assumptions and does not disrupt the certification flow. Also,

we have proved by induction that applying a primitive in the context of a CCDFG is correct.

Our current ACL2 script has 296 definitions and 1012 lemmas, including many lemmas about structural properties of CCDFGs (but not counting those from the false starts).

Since, we have a certified loop pipelining algorithm, we can confidently say that there are no data hazards and executing a sequential loop is same as executing a pipelined loop created using our algorithm. We have tested the pipeline reference model created using our algorithm on a variety of designs across different application domains. This shows that our algorithm is practical and can be used for industrial strength designs with tens of thousands of RTL.

With this dissertation, we have made the following major contributions:

– *Developed a framework of succinct certified primitives essential to build pipelining algorithms* : Our primitives are essential for developing certified loop pipelining algorithm in behavioral synthesis. This framework can also be extended to certify other pipelining algorithms such as function pipelines.

– *Designed and certified a reference loop pipelining algorithm* : We utilize our framework of certified primitives as backbone to build our certified loop pipelining algorithm. Since a primitive can only be applied under certain conditions, when certifying the algorithm, we prove that every application of our primitive is under correct conditions and certain assumptions are maintained after the application of a primitive. We also formalize and certify a key invariant for the correspondence between the sequential and pipelined CCDFGs and propose an algorithm for handling branch conditions in pipelines.

– *Evaluated our algorithm on industrial-strength designs* : We test our algorithm

on a variety of designs across different application domains. If our algorithm can generate a pipeline reference model for a design, we can compare it to the pipelined RTL generated by behavioral synthesis tools using SEC. If the SEC passes, we certify the application of loop pipelining transformation is correct. We show that our algorithm can discharge industrial-strength designs.

9.2 NEXT STEPS

Our dissertation shows that it is possible to develop and certify an industrial-strength loop pipelining algorithm if we can decompose it into succint certifiable primitives. We have already identified and certified these primitives. Our algorithm has components which can identify data hazards based on the given pipeline interval. Then we use our certified primitives to remove those data hazards and create a pipelined implementation.

Function pipelining algorithms also have the same type of data hazards as we have mentioned in loop pipelining algorithms. However, while loop pipelines have a fixed pipeline interval which is known at compile time, function pipelines have a variable pipeline interval for every iteration. So, instead of identifying data hazards at once for every iteration, we would have to call those functions for each iteration. After we have identified the data hazards, we can use our certified primitives to remove those data hazards. We believe that if we can modify the algorithm to identify data hazards, then we can conveniently reuse our certified primitives to certify behaviorally synthesized function pipelines as well.

REFERENCES

[1] Phi Operator LLVM reference manual. http://llvm.org/releases/2.9/docs/LangRef.html#i_phi. Accessed: September 29, 2016.

[2] WebM. VP9 Video Hardware RTL. http://www.webmproject.org/hardware/vp9/. Accessed: September 11, 2016.

[3] Mark D. Aagaard, Byron Cook, Nancy A. Day, and Robert B. Jones. *A Framework for Microprocessor Correctness Statements*, pages 433–448. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.

[4] Reinaldo A. Bergamaschi. *Behavioral Synthesis: an Overview*, pages 103–131. Springer US, Boston, MA, 2007.

[5] Yves Bertot, Pierre Castran, Grard (informaticien) Huet, and Christine Paulin-Mohring. *Interactive theorem proving and program development : Coq'Art : the calculus of inductive constructions*. Texts in theoretical computer science. Springer, Berlin, New York, 2004. Donnes complmentaires http://coq.inria.fr.

[6] J. R. Burch and D. L. Dill. Automatic Verification of Pipelined Microprocessor Control. In D. L. Dill, editor, *Proceedings of the 6th International Conference on Computer-Aided Verification (CAV 1994)*, volume 818 of *LNCS*, pages 68–80. Springer-Verlag, 1994.

[7] Cadence. *C-to-Silicon Reference Manual*, 2011.

[8] Calypto. *Catapult Reference Manual*, 2014.

[9] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Tomasz Czajkowski, Stephen D. Brown, and Jason H. Anderson. Legup: An open-source high-level synthesis tool for fpga-based processor/accelerator systems. *ACM Trans. Embed. Comput. Syst.*, 13(2):24:1–24:27, September 2013.

[10] R.O Chapman. *Verified high-level synthesis*. PhD thesis, Portland State Univeristy, 1994.

[11] Edmund Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. Satabs: Sat-based predicate abstraction for ansi-c. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'05, pages 570–574, Berlin, Heidelberg, 2005. Springer-Verlag.

[12] Edmund M. Clarke, Daniel Kroening, and Karen Yorav. Behavioral consistency of c and verilog programs using bounded model checking. In *DAC*, pages 368–371. ACM, 2003.

[13] Avra Cohn. *The Notion of Proof in Hardware Verification*, pages 359–374. Springer Netherlands, Dordrecht, 1993.

[14] J. Cong, Y. Fan, G. Han, W. Jiang, and Z. Zhang. Platform-based behavior-level and system-level synthesis. In *2006 IEEE International SoC Conference*, pages 199–202. IEEE, 2006.

[15] Philippe Coussy, Daniel D. Gajski, Michael Meredith, and Andrs Takach. An introduction to high-level synthesis. *IEEE Design & Test of Computers*, 26(4):8–17, 2009.

[16] David Cyrluk. Microprocessor verification in pvs - a methodology and simple example. Technical report, 1994.

[17] DavidRussinoff and Matt Kaufmann and Eric Smith and Robert Sumners. Formal Verification of Floating-Point RTL at AMD Using the ACL2 Theorem Prover, 2014.

[18] Gautam Doshi. Understanding the IA-64 architecture. Technical report, August 1999.

[19] Tom Feist. White paper: Vivado design suite. Technical report, Xilinx, June 2012.

[20] D. Gajski, N. D. Dutt, A. Wu, and S. Lin. *High Level Synthesis: Introduction to Chip and System Design.* Kluwer Academic Publishers, 1993.

[21] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic.* Cambridge University Press, New York, NY, USA, 1993.

[22] Mike Gordon, Juliano Iyoda, Scott Owens, and Konrad Slind. Automatic formal synthesis of hardware from higher order logic. *Electron. Notes Theor. Comput. Sci.*, 145:27–43, January 2006.

[23] Brian T. Graham. *The SECD Microprocessor: A Verification Case Study.* Kluwer Academic Publishers, Boston, MA, 2012.

[24] K. Hao, S. Ray, and F. Xie. Equivalence Checking for Behaviorally Synthesized Pipelines. In P. Groeneveld, D. Sciuto, and S. Hassoun, editors, 49*th International ACM/EDAC/IEEE Design Automation Conference (DAC 2012)*, pages 344–349. ACM, 2012.

[25] K. Hao, F. Xie, S. Ray, and J. Yang. Optimizing equivalence checking for behavioral synthesis. In *Design, Automation and Test in Europe (DATE 2010)*, pages 1500–1505. IEEE, 2010.

[26] Kecheng Hao. *Equivalence Checking for High-Assurance Behavioral Synthesis*. PhD thesis, Portland State Univeristy, 2013.

[27] David S. Hardin. *Design and Verification of Microprocessor Systems for High-Assurance Applications*. Springer Publishing Company, Incorporated, 1st edition, 2010.

[28] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3 edition, 2003.

[29] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, August 1978.

[30] Ravi Mohan Hosabettu. *Systematic Verification of Pipelined Microprocessors*. PhD thesis, The University of Utah, 2000.

[31] Alan Hu. High-level vs. rtl combinational equivalence: An introduction. In *ICCD*, pages 274–279. IEEE, 2006.

[32] James K. Huggins and David Van Campenhout. Specification and verification of pipelining in the arm2 risc microprocessor. *ACM Trans. Des. Autom. Electron. Syst.*, 3(4):563–580, October 1998.

[33] I. Moussa and Z. Sugar and R. Suescun and A. A. Jerraya and M. Diaz-Nava and M. Pavesi and S. Crudo and L. Gazzi. Comparing RTL and Behavioral Design Methodologies in the Case of a 2M Transistors ATM Shaper, 1999.

[34] Christian Jacobi. Formal verification of complex out-of-order pipelines by combining model-checking and theorem-proving. 2404:309, 2002.

[35] Richard Johnson and Keshav Pingali. Dependence-based program analysis. In *In Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 78–89, 1993.

[36] Roel Jordans and Henk Corporaal. High-level software-pipelining in llvm. In *Proceedings of the 18th International Workshop on Software and Compilers for Embedded Systems*, SCOPES '15, pages 97–100, New York, NY, USA, 2015. ACM.

[37] Daher Kaiss, Silvian Goldenberg, and Zurab Khasidashvili. Seqver : A sequential equivalence verifier for hardware designs. In *24th International Conference on Computer Design (ICCD 2006), 1-4 October 2006, San Jose, CA, USA*, pages 267–273, 2006.

[38] M. Kaufmann, P. Manolios, and J S. Moore. *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Publishers, Boston, MA, June 2000.

[39] M. Kaufmann, P. Manolios, and J S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, Boston, MA, June 2000.

[40] Sudipta Kundu, Sorin Lerner, and Rajesh Gupta. *Validating High-Level Synthesis*, pages 459–472. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[41] Sudipta Kundu, Zachary Tatlock, and Sorin Lerner. Proving optimizations correct using parameterized program equivalence. *SIGPLAN Not.*, 44(6):327–337, June 2009.

[42] Raya Leviathan and Amir Pnueli. Validating software pipelining optimizations. In *Proceedings of the 2002 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, CASES '02, pages 280–287, New York, NY, USA, 2002. ACM.

[43] Jeremy Levitt and Kunle Olukotun. Verifying correct pipeline implementation for microprocessors. In *Proceedings of the 1997 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '97, pages 162–169, Washington, DC, USA, 1997. IEEE Computer Society.

[44] Youn-Long Lin. Recent developments in high-level synthesis. *ACM Trans. Des. Autom. Electron. Syst.*, 2(1):2–21, January 1997.

[45] Hanbing Liu and J. Strother Moore. Executable jvm model for analytical reasoning: a study. *Sci. Comput. Program.*, 57(3):253–274, September 2005.

[46] P. Manolios. Correctness of Pipelined Machines. In W. A. Hunt, Jr. and S. D. Johnson, editors, *Proceedings of the 3rd International Conference on Formal Methods in Computer-Aided Design (FMCAD 2000)*, volume 1954 of *LNCS*, pages 161–178, Austin, TX, 2000. Springer-Verlag.

[47] Takeshi Matsumoto, Hiroshi Saito, and Masahiro Fujita. Equivalence checking of c programs by locally performing symbolic simulation on dependence graphs. In *Proceedings of the 7th International Symposium on Quality Electronic Design*, ISQED '06, pages 370–375, Washington, DC, USA, 2006. IEEE Computer Society.

[48] Michael Meredith. *High-Level SystemC Synthesis with Forte's Cynthesizer*, pages 75–97. Springer Netherlands, Dordrecht, 2008.

[49] S. Owre, J. M. Rushby, , and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, jun 1992. Springer-Verlag.

[50] D. Puri, S. Ray, K. Hao, and F. Xie. Mechanical certification of loop pipelining transformations: A preview. In G. Klein and R. Gamboa, editors, 4*th International Conference on Interactive Theorem Proving (ITP 2014)*, volume 7998 of *LNCS*. Springer, 2014.

[51] S. Ray and J. Bhadra. Abstracting and Verifying Flash Memories. In K. Campbell, editor, *Proceedings of the* 9*th Non-Volatile Memory Technology Symposium (NVMTS 2008)*, pages 100–104, Pacific Grove, CA, November 2008. IEEE.

[52] S. Ray, K. Hao, F. Xie, and J. Yang. Formal Verification for High-Assurance Behavioral Synthesis. In Z. Liu and A. P. Ravn, editors, *Proceedings of the 7th International Symposium on Automated Technology for Verification and Analysis (ATVA 2009)*, volume 5799 of *LNCS*, pages 337–351, Macao SAR, China, October 2009. Springer.

[53] S. Ray and W. A. Hunt, Jr. Mechanized Certification of Secure Hardware Designs. In M. S. Abadir, L. Wang, and J. Bhadra, editors, *Proceedings of the 8th International Workshop on Microprocessor Test and Verification, Common Challenges and Solutions (MTV 2007)*, pages 25–32, Austin, TX, December 2007. IEEE Computer Society.

[54] S. Ray and W. A. Hunt, Jr. Connecting Pre-Silicon and Post-silicon Verification. In A. Biere and C. Pixley, editors, *Proceedings of the* 9*th International*

Conference on Formal Methods in Computer-Aided Design (FMCAD 2009), pages 160–163, Austin, TX, November 2009. IEEE Computer Society.

[55] Sandip Ray. *Scalable Techniques for Formal Verification*. 2010.

[56] Hamid Savoj, David Berthelot, Alan Mishchenko, and Robert Brayton. Combinational techniques for sequential equivalence checking. In *Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design*, FMCAD '10, pages 145–150, Austin, TX, 2010. FMCAD Inc.

[57] J. Sawada and W. A. Hunt, Jr. Verification of FM9801: An Out-of-Order Microprocessor Model with Speculative Execution, Exceptions, and Program-Modifying Capability. *Formal Methods in Systems Design (FMSD)*, 20(2):187–222, 2002.

[58] Klaus Schneider. *A Verified Hardware Synthesis of Esterel Programs*, pages 205–214. Springer US, Boston, MA, 2001.

[59] Jens U. Skakkebæk, Robert B. Jones, and David L. Dill. *Formal verification of out-of-order execution using incremental flushing*, pages 98–109. Springer Berlin Heidelberg, Berlin, Heidelberg, 1998.

[60] Sudarshan K. Srinivasan. Automatic refinement checking of pipelines with out-of-order execution. *IEEE Transactions on Computers*, 59(undefined):1138–1144, 2010.

[61] J. Tristan and X. Leroy. A Simple, Verified Validator for Software Pipelining. In M. V. Hemenegildo and J. Palsberg, editors, *Proceedings of the* 37*th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2010)*, pages 83–92, January 2010.

[62] M. N. Velev and R. E. Bryant. TLSim and EVC: A term-level symbolic simulator and an efficient decision procedure for the logic of equality with uninterpreted functions and memories. *International Journal on Embedded Systems*, pages 134–149, 2005.

[63] Miroslav N. Velev. *Formal Verification of VLIW Microprocessors with Speculative Execution*, pages 296–311. Springer Berlin Heidelberg, Berlin, Heidelberg, 2000.

[64] Miroslav N. Velev and Randal E. Bryant. Formal verification of superscale microprocessors with multicycle functional units, exception, and branch prediction. In *Proceedings of the 37th Annual Design Automation Conference*, DAC '00, pages 112–117, New York, NY, USA, 2000. ACM.

[65] David J. Wheeler and Roger M. Needham. Tea, a tiny encryption algorithm. In Bart Preneel, editor, *FSE*, volume 1008 of *Lecture Notes in Computer Science*, pages 363–366. Springer, 1994.

[66] Jin Yang and Carl-Johan H. Seger. Introduction to generalized symbolic trajectory evaluation. In *ICCD*, pages 360–367. IEEE Computer Society, 2001.

[67] Z. Yang, K. Hao, K. Cong, F. Xie, and S. Ray. Equivalence Checking for Compiler Transformations in Behavioral Synthesis. In 31*st International Conference on Computer Design (ICCD 2013)*, pages 491–494, 2013.

[68] Zhenkun Yang, Kecheng Hao, Kai Cong, Li Lei, Sandip Ray, and Fei Xie. Scalable certification framework for behavioral synthesis front-end. In *The 51st Annual Design Automation Conference 2014, DAC '14, San Francisco, CA, USA, June 1-5, 2014*, pages 149:1–149:6, 2014.

[69] Zhenkun Yang, Kecheng Hao, Kai Cong, Li Lei, Sandip Ray, and Fei Xie. Validating scheduling transformation for behavioral synthesis. In *2016 Design, Automation & Test in Europe Conference & Exhibition, DATE 2016, Dresden, Germany, March 14-18, 2016*, pages 1652–1657, 2016.

[70] Zhenkun Yang, Sandip Ray, Kecheng Hao, and Fei Xie. Handling design and implementation optimizations in equivalence checking for behavioral synthesis. In *Proceedings of the 50th Annual Design Automation Conference*, DAC '13, pages 117:1–117:6, New York, NY, USA, 2013. ACM.