Formal Verification Techniques for Behaviorally Synthesized Loop Pipelines

No Author Given

No Institute Given

Abstract. Behavioral synthesis is the process of compiling an Electronic System Level (ESL) specification of a hardware design to an RTL description. Loop pipelining is a critical transformation employed by behavioral synthesis, and ubiquitous in commercial and academic behavioral synthesis tools. In this paper, we provide an approach to certify loop pipelining transformation. Critical to the approach is the development of a formal, mechanically verifiable loop pipelining algorithm. We show how to construct such an algorithm and mechanically certify it using the ACL2 theorem prover. We integrated our algorithm within a behavioral synthesis certification flow, and used it to verify ESL designs synthesized by a commercial behavioral synthesis tool.

1 Introduction

In recent years, the demand for hardware with smaller form factor and higher transistor density has been steadily increasing. Consequently, it has become difficult to create high quality hardware by hand-crafting RTL designs. A behavioral synthesis tool takes a behavioral design description (in C, C++, or SystemC) and automatically generates an optimized RTL design. It has recently received significant attention, as the steady increase in hardware complexity has made it increasingly difficult to design high-quality designs through hand-crafted RTL under aggressive time-to-market schedules [23]. Nevertheless, and in spite of availability of several commercial behavioral synthesis tools [4, 17, 8], 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.

Loop pipelining is one of the most complex transformations in behavioral synthesis. It is available in most commercial synthesis tools [24, 17] and is crucial to producing hardware with high throughput and low latency by allowing temporal overlap of successive loop iterations. Certifying the correspondence between a sequential design and its pipelined counterpart is challenging due to the huge semantic gap between the two designs. 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 [9, 13]. 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 earlier [10]. The most critical and complex component of this approach 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. The viability of this approach was shown by comparing pipeline generated from this algorithm with RTL implementation using SEC. However, this algorithm was not certified as well as incomplete. In fact, during our certification process we found a number of errors in the algorithm, which makes the entire certification flow for behavioral synthesis suspect. Our work on developing a certified loop pipelining algorithm using our framework of certified primitives is important to facilitating formal verification of behaviorally synthesized pipeline designs.

The key contributions of this paper are:

1. Developing a certified executable loop pipelining algorithm in ACL2 using our framework of certified pipelining primitives
2. Demonstrating the viability of the certified loop pipelining algorithm in a behavioral synthesis certification flow for industrial hardware designs

The remainder of this paper is organized as follows. Section 2 provides background on the need for a certified loop pipelining algorithm to certify behaviorally synthesized designs. Section 3 discusses our formalization of the intermediate representations used in behavioral synthesis. We also discuss the correctness statement for loop pipelining algorithms. We discuss our framework and a certified loop pipelining algorithm we have developed using the framework in Section 4. Section 5 provides a proof sketch for our algorithm and Section 6 provides evaluation of robustness and scalability of our algorithm on industrial-strength designs. Section 7 discusses the work that has been done in the area of microprocessor and software pipeline verification before and how our work is different. We then conclude and discuss future work in Section 8.

2 Background and Context

Behavioral synthesis [15] is an automated compilation process where a behavioral synthesis tool [9, 6, 5] 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 following three phases (refer Figure 1). After that, the design can be expressed in RTL which goes through further optimizations

1. **Compiler Transformations:** These include typical compiler operations, *e.g.,* dead-code elimination, constant propagation etc.

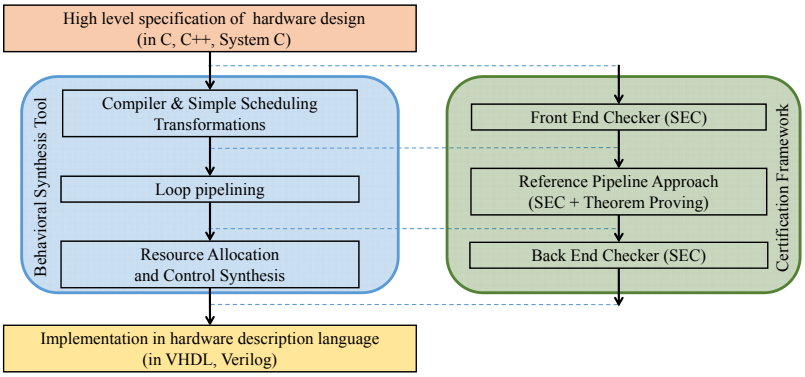

Fig. 1. Certification Model for Behaviorally Synthesized Pipelines

2. **Scheduling Transformations:** Scheduling entails computing the clock cycle for executing each operation accounting for hardware constraints and control/data dependencies. This phase includes loop pipelining.
3. **Resource Allocation and Control Synthesis:** This phase involves mapping a hardware resource to each operation, allocating registers to variables, and generating a controlling finite-state machine to implement the schedule.

. There is a large gap in abstraction between the ESL and RTL descriptions so that there is little correspondence in internal variables between the two. Consequently, direct SEC is not feasible as it reduces to cost-prohibitive computation of input-output equivalence. Theorem proving is impractical as it requires extensive manual effort which needs to be replicated for each different synthesized design. Besides reasons of complexity, it is also not possible to directly certify the implementation of the *synthesis tool* via theorem proving as the implementations are typically closed-source and closely guarded by EDA vendors and thus out of reach of external automated reasoning communities.

Some solutions have been proposed to divide the overall certification problem into manageable portions. An SEC technique has been tested on industrial strength designs which can compare the RTL with the intermediate representation (IR) generated by the tools after the high-level (compiler and scheduling) transformations have been applied [20, 11]. In particular, operation-to-resource mappings generated by the synthesis tool provide the requisite correspondence between internal variables of the IR and RTL. There is another SEC technique that 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 [25]. This technique 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. Thus a specialized approach is warranted for handling its certification. Hao *et al.* proposed a pipeline generation algorithm using feedback (like pipeline interval) from the synthesis tool [10]. They show that to verify the correspondence between sequential CCDFG and pipelined RTL, it is sufficient to

1. Check that the algorithm can generate a pipeline reference model for the parameters reported by synthesis.
2. Use SEC to compare the pipeline reference model with the synthesized RTL.
3. Prove the correctness of the algorithm.

The pipelining algorithm is simpler than that used by the synthesis tool because the synthesis tool uses advanced heuristics to determine the pipeline parameters (such as how many iterations to pipeline, when to introduce stalls etc.), while this pipelining algorithm only uses those parameters to generate a reference model. The algorithm is shown to be scalable but it is not certified.

Our first approach was to certify this 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 [16].

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 work is on identifying certifiable primitives 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. Our framework is independent of the inner workings of a specific tool, and can be applied to certify designs synthesized by different tools Also, the approach produces a certified reference flow, which makes explicit generic invariants that must be preserved by different transformations.

3 Formalization

3.1 Intermediate Representation

A behavioral synthesis tool performs a number of compiler and scheduling transformations (including pipelining). Certification of behavioral synthesis transfor-

mations thus requires a formalization of the design representation manipulated by these transformations. The formalization we use is *Clocked Control Data Flow Graph* (CCDFG) [20]. Structurally, a CCDFG is a control and data flow graph augmented with a schedule. The control flow is broken into basic blocks. The instructions are grouped into microsteps which can be executed concurrently. A scheduling step represents a group of microsteps which can be executed in a single clock cycle. 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 a scheduling step. The underlying language we use is the LLVM language [14]. LLVM is a popular compiler infrastructure for many behavioral synthesis tools [24, 6] 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 as shown in Figure 2, in our syntax, an *assignment statement* can be expressed as a list of a variable and an expression. 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 [16]. 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 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))))))

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

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

Fig. 2. ACL2 syntax

3.2 Correctness of Loop Pipelining

For the purposes of this paper, a *pipelinable loop* is a loop with the following restrictions [10]: (1) no nested loop; (2) only one *Entry* and one *Exit* block; and (3) no branching between the scheduling steps. Our well-formed pipelinable loop has 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* [1] 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.

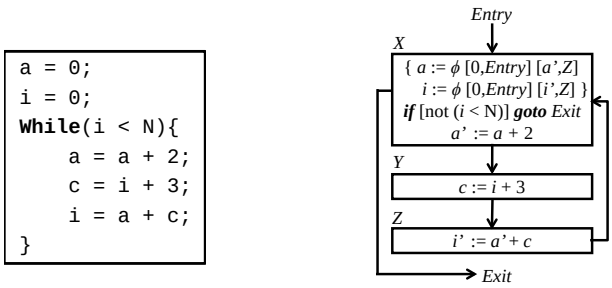

Fig. 3. (a) Loop in C (b) Loop CCDFG before pipelining.

Figure 3(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(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'.

The main lemma involved in the correspondence proof between the sequential and pipelined CCDFG can be 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]

```
(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] 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`.

[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.

[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.

[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.

4 Our Approach

Pipeline synthesis is based on the key observation that 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.

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 in detail.

4.1 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 – 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 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 known.

Shadow register primitive – We define a shadow register microstep as 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.

Branch primitive – Branch instructions are required for 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 4(b)), 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

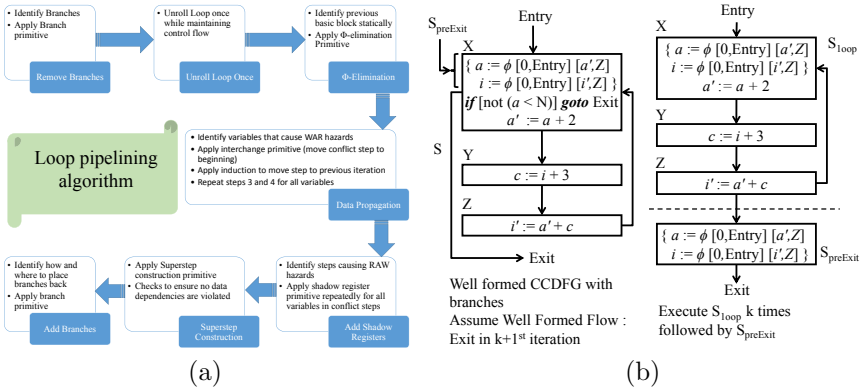

Fig. 4. (a) Algorithm Using Primitives (b) Branch Primitive

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 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 collection of microsteps) in a CCDFG where both m and n do not have read hazards and any microsteps containing branch statements. Then, the interchange primitive allows us to interchange their order without affecting execution.

Superstep construction primitive – This 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.

4.2 Our Loop Pipelining Algorithm

Given a sequential loop S in CCDFG C and pipeline interval I, we can create a pipelined loop P (refer Figure 4 (a)). Note that every step of the algorithm is build from ground up using our framework of provable primitives such that the algorithm can be certified by theorem proving. Now, we describe the steps to convert a sequential loop CCDFG S (c.f. Figure 4) to a pipelined loop CCDFG:

Remove Branches: We apply the branch primitive on S to remove branches.

Unroll Loop Once: The first iteration behaves differently than the rest due to ϕ-construct. So, we unroll the loop S_{loop} once and call it S_{pre}.

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.

Algorithm 1 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
```

Data propagation: Algorithm 1 describes how to compute candidates for data propagation across pipeline iterations. It is a critical step as 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* relocates the microstep which is causing the data hazard to the previous iteraton. Note that this step requires multiple applications of interchange primitive and very complex induction. 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 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(a) we found that the loop carried dependency i' in X would create a conflict when we would move X before Z while pipelining. So, we relocate the microstep $i := i'$ as shown in Figure 5(a). This step needs to be repeated for every variable found using *GetLoopCarriedDependencies*.

Generate shadow registers: Algorithm 2 inserts shadow registers to prevent variables from being overwritten before being read. 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

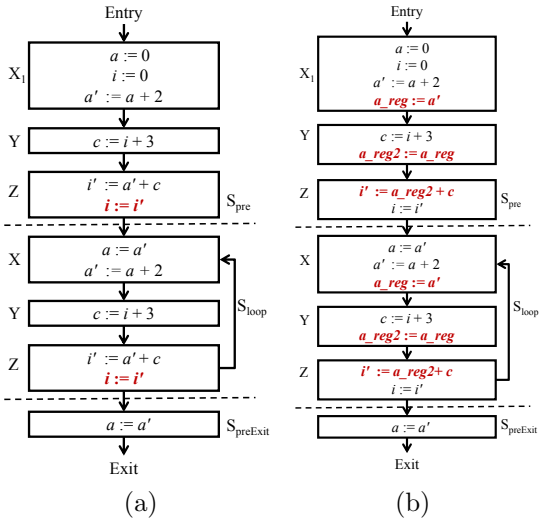

Fig. 5. (a) After Data Propagation (b) After adding Shadow Register

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. In Figure 5(b), 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 $GetAllVariables$.

Algorithm 2 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
```

Superstep construction: Now that we have removed the data hazards, we can successfully pipeline the loop using the pipeline interval I. We combine the

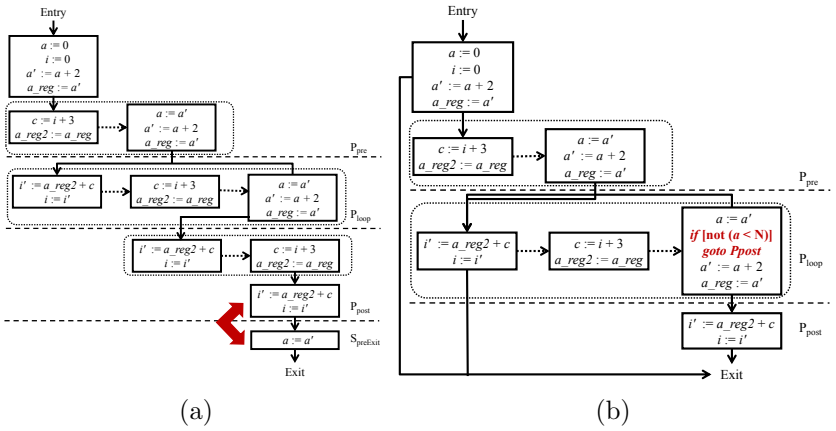

Fig. 6. (a) After superstep construction (b) Final Pipelined CCDFG

scheduling steps of the successive iterations, forming scheduling "supersteps" that act as scheduling steps for the pipelined implementation. 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}, P_{loop} which is the full pipeline stage and epilogue P_{post} as shown in Figure 6.

Add Branches: To add the branches back, we use the a combination of interchange primitive and reverse of Branch primitive. Note in Figure 6, 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 6 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 6(b).

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

5 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.

5.1 Correctness of Primitives

We prove that applying a particular primitive is correct, *i.e.*, maintains a certain invariant. This proof does not consider how it is applied in the context of a pipeline synthesis algorithm. In *ϕ-elimination primitive*, 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. In *Shadow register primitive*, we prove that adding a shadow register microstep, $a\_reg = a'$ 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. In *Interchange primitive*, we prove that we can interchange any two adjacent microsteps (excluding branch microsteps) which do not have read-write conflict by reasoning about execution semantics of all types of microsteps present in the language. In *Branch primitive*, we prove that executing S such that it exits in the $(k+1)$ st iteration is same as executing S_{loop} k times followed by $S_{preExit}$. 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 to track the backedge along the unconditional branch and ensure that it points back to the beginning of loop S. *Superstep construction 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 a key invariant on correspondence between back-edges of sequential and pipelined loops [19].

5.2 Correctness of Our Algorithm

The algorithm is essentially built from ground-up using primitives as shown in Section 4. 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. 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. 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. After this step and for all the subsequent steps, we need to show that there are no relevant branches in CCDFG. In the ϕ-to-assign stage, we replace one microstep of C with more than one microsteps in C'. 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. *Datapropogation* 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. *Shadowregisterstep* also 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 introduced and we can only claim that the relevant variables have same value.This step requies proof of invariant and multiple applications of interchange primitive as explained earlier. 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 Viability of our Approach

We have successfully tested the pipeline reference model generated by our certified algorithm with that generated by the previous algorithm [12] across several

industrial strength designs in different domains(c.f Figure 1). The designs are non-trivial to pipeline with varied pipeline intervals and depths.

Table 1. Behaviorally synthesized pipelined designs tested using our algorithm

Design	RTL Lines	Application Domain	Loop Interval	Loop Depth	No. of ops	Pipeine 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

7 Related Work and Novelty of Our Approach

Besides behaviorally synthesized pipelines, there are mainly two other kinds of pipelines, hardware pipelines and software pipelines. Microprocessor pipelines [18, 7, 3, 21] 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*. Our invariant is very different from a typical invariant used in the verification of pipelined machines [21] . 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.

Our understanding of hazards and reasoning behind pipelining algorithm is very closely related to verification of software pipelines. In particular, Tristan and Leroy [22] 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 [2] 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 provides 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. Section 3); achieving this for arbitrary software pipeline may require further more subtle invariants.

8 Conclusion and Future Work

We have developed a framework of succinct certified primitives essential to build pipelining algorithms. We utilize our framework of certified primitives as backbone to build our certified loop pipelining algorithm. 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 industrial strength designs across different application domains.

Definitions	296
Lemmas	1012

Table 2. ACL2 Effort

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.

Bibliography

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

[2] 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.

[3] 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.

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

[5] 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.

[6] 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.

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

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

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

[10] 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.

[11] 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.

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

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

2008.

[14] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO 2004)*, pages 75–84, March 2004.

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

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

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

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

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

[24] Xilinx. *AutoESL Reference Manual*, 2011.

[25] 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.