Verification of RTL Generated from Scheduled Behavior in a High-Level Synthesis Flow

Pranav Ashar Subhrajit Bhattacharya Anand Raghunathan
Akira Mukaiyama
C&C Research Labs, NEC USA, Princeton, NJ

Abstract

We propose a complete procedure for verifying register-transfer logic against its scheduled behavior in a high-level synthesis environment. Our proposal advances the state of the art because it is the first such verification procedure that is both complete and practical. Hardware verification is known to be a hard problem and the proposed verification technique leverages off the fact that high-level synthesis - performed manually or by means of high-level synthesis software - proceeds from the algorithmic description of the design to structural RTL through a sequence of very well defined steps, each limited in its scope. The major contribution is the partitioning of the equivalence checking task into two simpler subtasks, verifying the validity of register sharing, and verifying correct synthesis of the RTL interconnect and control. While state space traversal is unavoidable for verifying validity of the register sharing, we automatically abstract out irrelevant portions of the design, significantly simplifying the task that must be performed by a back-end model checker. The second task of verifying the RTL is not only shown to reduce to a combinational equivalence check, we present a novel and fast RTL technique for combinational equivalence check instead of using slower gate level techniques. The verification procedure has been applied to several large circuits, and is illustrated on the implementation of a sort algorithm.

1 Introduction

The trend for the future is for designers to start their designs with design descriptions more abstract than structural RTL. The ultimate focus of our work is to provide methodologies and tools for equivalence checking between structural RTL and their more abstract higher level initial descriptions. These techniques must exploit knowledge of circuit properties and the synthesis flow to be viable.

We are aware of various efforts in the verification of designs generated from high-level descriptions[1, 2, 3]. Without going into the details of these methods, it should suffice to say that these methods are either not complete or do not derive any efficiency out of their knowledge of the high-level synthesis domain, limiting their applicability. Similarly, technologies like symbolic model checking, language containment, theorem proving, techniques for modeling arithmetic in the verification context, are extremely powerful but must be applied wisely and in the proper context to be effective.

We wish to eventually develop a strategy for structural RTL validation in which the equivalence between structural RTL and its most abstract initial description is established by proving the equivalence between the initial and final descriptions at each synthesis step [4]. As in the case of formal equivalence checking tools at the purely structural level, we must make assumptions about the scope of each synthesis step and the properties of the synthesized designs. Our basic assumption is that synthesis proceeds in the well defined steps of algorithmic transformations, followed by scheduling, finally followed by structural RTL synthesis. Taking a bottom up approach, we have taken the first step of developing an equivalence checking procedure for the validation of structural RTL against scheduled behavior which we equivalently call functional RTL. This is the problem we address in this paper.

At the level of scheduled behavior, each data path operation is clearly associated with the state in which it will be executed. The transition between states as a function of the present state, primary inputs and the results of data path operations is also clearly defined. What is not defined is the association of variables with registers, and the actual circuit implementation of the control/data flow in each state. The assignment of registers to variables (with registers possibly being shared among variables), the generation of hardware to enable the register assignment and sharing, and the generation of hardware to enable the sharing of functional blocks among data path operations are exactly the synthesis steps performed in the generation of structural RTL from scheduled behavior. In this paper, we propose algorithms which can be used to check the correctness of these synthesis steps applied to scheduled behavior. The algorithms are novel in that they are tuned specifically to the synthesis steps that they are supposed to check. That is what makes them practical without sacrificing completeness.

In essence, our algorithms are based on the observation that the state space explosion in most designs is caused by the data-path registers rather than the number of control states. Given the clear delineation between data-path and control in the high-level synthesis environment, we are able to divide the equivalence checking task into the checking of (1) local properties which are checked on a per control state basis, and (2) non-local properties which require a traversal of the control state space. The non-local properties are checked in the following manner: A number of assertions are generated for each non-local property such that checking all the assertions is equivalent to checking the non-local property. Each assertion is then checked separately on a model of the design relevant to the assertion being checked, with the rest of the design abstracted out. A number of model checking tools can be used for the purpose, e.g. [5]. The abstraction of the design for the assertion being checked is key. Again, this is made possible by the clear delineation between data-path and control and the small number of control states. Details of our algorithms are provided in the following sections.

2 Verifying RTL Implementation of a Schedule

In this section, we develop an algorithm to verify the correctness of a structural register-transfer level (RTL) implementation of a scheduled behavioral specification or schedule. We formally define the verification problem that we are addressing. We prove that the implementation is equivalent to the specification if two key properties can be verified. The two properties, the *valid register sharing* property, and the *intra cycle equivalence* property, thus allow us to partition a complex verification problem into two simpler subproblems. The problem formulation and the two properties are discussed in Section 2.1. Algorithms for checking the two properties are discussed in Sections 2.2 and 2.4.

2.1 Problem Formulation and Partitioning

We first define an RTL specification. An RTL specification can be defined in terms of variables V, operations O, and clocks, the clocks governing the updating of the value of variables. Variables are divided into 4 sets V = (PI, PO, R, T). PI is the set of primary inputs, PO is the set of primary outputs, R is the set of register variables, and T is the set of temporary variables. Operations are categorized into two types O = (C, A), C being the set of control operations, and A being the set of assignment operations. The result of a control operation is boolean, and the results are used to control execution of other operations. Assignment operations assign to and change value of variables. Every operation op_i has a corresponding condition c_i associated with it. The condition c_i is a logical expression and may be composed using the results of other assignment and control operations. The operation op_i is executed only if c_i is true. The definition holds for functional RTL specifications, such as the schedule in Figure 1 which has been taken from[6], as well as for structural RTL specifications, such as the RTL circuit in Figure 6.

The updating of values of variables is controlled by the clock. The clock splits time into discrete integral values starting at time $t = 0$, with t being incremented by 1 at every clock tick. At time $t = 0$, some register variables are initialized. The value of a register variable at any given time $t = T$, $t > 0$, is fixed and independent of operations taking place at that time. If a register variable is assigned a value at $t = T$, the register variable assumes the value at $t = T + 1$. If no assignment is made to a register variable at $t = T$, it retains its value from $t = T - 1$. The value of a temporary variable or primary output is defined at $t = T$ only if an assignment is made to it at that time. Primary inputs are not assigned to and can only be used as inputs for an operation.

Schedules of a behavioral specifications have some additional properties. The schedule consists of a distinguished variable called the **state** variable. The state variable can attain a fixed set of known values. A state of the schedule corresponds to a particular value of the state variable. It is assumed that at $t = 0$, the $state$ variable is initialized to s_0. The initialization is not shown in Figure 1(a). Corresponding to each state, there is a set of **next state** operations which assigns a constant value from the known set of values to the state variable. If the schedule is in a particular state at time $t = T$, the next state operations define the possible values of the state variable at time $t = T + 1$. Depending upon the conditions, only one of the next state operations is executed at any time. For each state of the schedule, there is a set of operations which are conditionally executed only in the state they belong to. Thus, the condition for execution of op_6 is $c_6 = (state == s_1)$ *AND* $(p == 0)$.

The schedule specification properties, Property 1, and the properties of the RTL synthesis procedure, Property 2, stated next are important for our verification algorithms to work successfully.

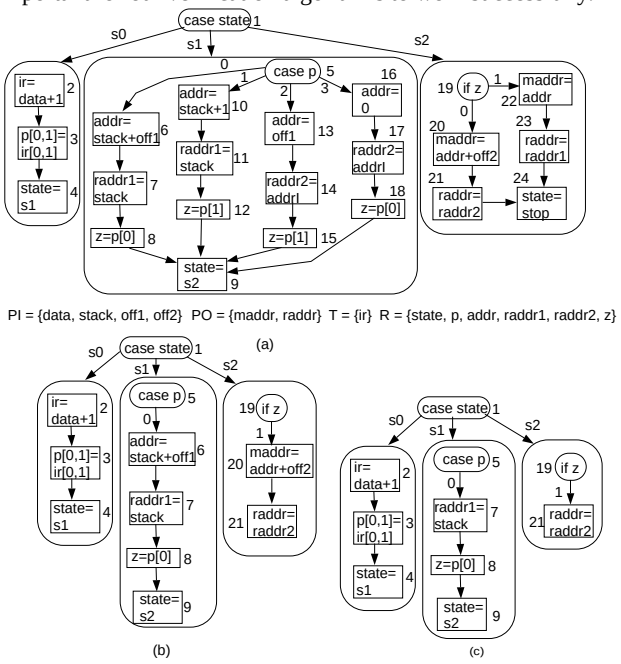

Figure 1: (a) The schedule of a behavioral description. $raddr1_1$ and $raddr2_1$ share register R_1. (b) Subgraph containing conflicting gen-use pair, op_{21} and op_7. (c) Subgraph derived from (b) after removing operations not essential for proving whether the gen-use pair is benign or not.

Property 1 (Specification) *The following properties are for any RTL specification.*

1. *We assume that at any time, only one assignment can be made to a variable.*
2. *If any register variable is used in an operation at $t = T$, $T > 0$, then it must have been initialized or assigned at some time $t < T$. Note that at $t = 0$, none of the register carry a valid value due to initialization, since if a register is assigned a value at $t = T$, it contains the value only at $t = T + 1$. Thus, none of the operations at $t = 0$ can use a register operand, and can use only primary inputs.*

Property 2 (Transformation Invariants) *The following properties are assumed for an RTL implementation synthesized from an RTL specification.*

1. *There is a 1 to 1 mapping between INPUTS and OUTPUTS in the two representations.*
2. *There is a many to one mapping from the variables in the set* R *of the specification and the variables in the set* R *of the implementation. Let R_i be a register variable in the implementation, and M_i be the set of register variables of the specification which have been mapped to R_i. If M_i has more than one element, we say that R_i is being shared. For*

ease of explanation, if a variable x belongs to M_i, we simply rename it x_i.

Problem Statement. *Given a functional RTL specification and a structural RTL specification which satisfies Property 1 and Property 2, verify that at any time $t = T$, if an output variable of the specification is assigned a value, the same value is assigned to the corresponding output variable of the implementation.*

We next present two properties, Property 3 and Property 4, and prove in Lemma 1 that if the two properties are satisfied by the implementation, then the implementation's outputs are assigned the same values as the corresponding outputs of the specification, as is required by the problem statement.

Definition 1 (Path in Schedule) *We say that there is a path in a schedule from state s_i to s_j, if there are next state transitions which can change the state from s_i to s_j.*

Definition 2 (Conflicting Gen-Use pair) *Let two specification variables x_i and y_i, $x_i \neq y_i$, be mapped to the same register R_i in the implementation. Let operation op_m in state s_j generate x_i and operation op_n use y_i in state s_k, the two operations being executed in different clock cycles, and there is a path P from state s_j to s_k such that no assignment to R_i takes place in any of the states between s_j and s_k. Then we say that the operation pair op_m and op_n is a conflicting gen-use pair for register R_i along P.*

Definition 3 (Benign Gen-Use Conflict) *Let there be a conflicting gen-use pair op_m in state s_j and op_n in state s_k along path $P^{''}$. Let there be a path $P^{'}$ from s_0 to s_j. For the path $P^{'}$ followed by $P^{''}$, let $state = s_j$ at $t = T1$ and $state = s_k$ at $t = T2$. Then when the sequence of transitions in $P^{'}$ is executed followed by the sequence of transitions in $P^{''}$, if either $c_m^{'}$ is FALSE at $t = T1$ or $c_n^{'}$ is FALSE at $t = T2$, the gen-use conflict is said to be benign.*

Property 3 (Valid register mapping) *An implementation is said to satisfy the valid register mapping property if for any register R_i in the implementation, either the corresponding set M_i has only one element, or M_i has multiple elements, but all conflicting gen-use pairs for such registers are benign.*

The structural RTL circuit of Figure 6 which is an implementation of the schedule of Figure 1(a) shares register R_1 amongst specification variables $raddr1$ and $raddr2$. Hence, $M_1 = \{raddr1_1, raddr2_1\}$. Since $raddr1_1$ is assigned by operation op_7 in state s_1 and $raddr2_1$ is used by operation op_{21} in state s_2, and there is a state transition from s_1 to s_2, there is a possible conflicting gen-use pair. However, as we will show later, execution of op_7 is never followed by execution of op_{21} in a subsequent cycle due to the existence of a false path. Thus, the conflicting gen-use pair for R_1 is actually benign.

Definition 4 (Operation modulus using register mapping) *An operation in the specification modulus the register mapping is another operation derived from the specification operation by replacing all operands x_i which are register variables with the corresponding registers R_i to which the variables have been mapped in the implementation.*

As an example, consider operation op_7 in Figure 1. Assume that in the implementation of Figure 6 variables *stack*, and *raddr1* are mapped to registers R_{stack}, and R_1. Thus op_7 which is $raddr1 := stack$ modulus the register mapping is $R_1 := R_{stack}$.

Property 4 (Intra-Cycle Equivalence Property) *An implementation is said to satisfy the* intra-cycle equivalence *property if, at any time $t \geq 0$, the set of assignment operations executed in the specification modulus the register mapping is the same as the set of assignment operations performed in the implementation.*

Definition 5 (Variable contained in register) *For an implementation register R_i, and the corresponding set M_i, the variable contained in R_i is an element of M_i. For a sequence of operations from time $t = 0$ until $t = T$, different elements of M_i may have been assigned at different times t, $t < T$. For the given operation sequence, the element of M_i assigned for the largest t, $t < T$, is defined to be contained in R_i at $t = T$.*

Consider again Figure 1. As stated before but not shown in the figure, at $t = 0$, the state variable is initialized to s_0. If operations $op_1 \ldots op_4$ are executed at $t = 1$, and operations $op_5, op_6, \ldots op_9$ at $t = 2$, then R_1 contains $raddr1_1$ at $t = 3$. If operations $op_5, op_{13}, \ldots, op_{15}, op_9$ are executed at $t = 2$, then R_1 contains $raddr2_1$ at $t = 3$.

Lemma 1 *For an implementation of a schedule specification and a register mapping which satisfies Property 3 and Property 4, at $t = T$, $T > 0$, if R_i contains x_i, then R_i contains the correct value of x_i, that is, R_i contains the same value as the value of x_i in the schedule specification at $t = T$.*

Theorem 1 *If an implementation of a specification satisfies Properties 3 and 4, then if at any time $t = T$, an output variable of the specification is assigned a value, the same value is assigned to the corresponding output variable of the implementation.*

Proof: The proof for $t = 0$ follows from Properties 1 and 2. The proof for $t > 1$ follows from Lemma 1. Details are omitted. ∎

In the next two sections, Section 2.2 and Section 2.4, algorithms for verifying Property 3 and Property 4 are discussed. These two algorithms combined will give us a complete algorithm for verifying that the implementation of a specification is correct as required by the problem statement presented earlier in this section.

2.2 Algorithm for Verifying Validity of Register Sharing

In this section, we present an algorithm for verifying the validity of register mapping for schedules of behavioral descriptions. Definitions 2, and 3 and the statement in Property 3 form the basis of the algorithm.

The first step involves identifying paths along which conflicting gen-use pairs occur, as defined in Definition 2. Suppose an operation node op_n is a *gen* node for a variable x_i, and M_i has more than 1 element. There may be many paths along which conflicts arise involving op_n. Instead of enumerating each path separately, we identify a conflict subgraph *CSG* which contains all the conflicting paths involving op_n. The procedure for identifying *CSG* is a constrained depth-first search (DFS), DFS_CSG, as given in Figure 2. Let $uset(op_n, x_i) = \{y_i | y_i \neq x_i, y_i \in M_i, y_i$

input to op_n}), and $gset(op_n, x_i) = \{y_i | y_i \in M_i, y_i$ assigned by $op_n\}$. Let op_k be a node visited during the forward traversal phase of the DFS. If $uset$ is not empty, then a conflicting use operation has been identified and DFS marks the node as a CSG node and returns the value MARKED. If $gset$ for op_k is not empty, an assignment is being made to R_i in op_k. Since this violates the condition for a conflicting gen-use pair, node op_k can not be on a conflicting path. Thus, the DFS routine does not mark the node and returns with the value NOTMARKED. If neither of the above conditions hold, then the DFS marks the node op_k only if one of its successors returns MARKED.

```
Procedure DFS_CSG(op_k, x_i)
  if (uset(op_k, x_i) ≠ ∅) {
    op_k ->mark := 1;
    return MARKED; }
  if (gset(op_k, x_i) ≠ ∅) {
    op_k ->mark := 0;
    return NOTMARKED; }
  for all successors op_j of op_k {
    if (DFS_CSG(op_j, x_i) == MARKED) {
      op_k ->mark := 1;
      return MARKED; } }
```

Figure 2: Algorithm for identifying subgraph with paths containing conflicting gen-use pairs.

To verify that the gen-use conflicts are benign, as per definition 3, the reachability subgraph *RG* containing paths from the start state of the schedule to op_n is identified, and is added to the *CSG* subgraph to form the conflict graph *CG*. Let there be a path $(op_0, \ldots, op_{i1}, \ldots, op_{i2}, \ldots op_{i3}, \ldots, op_{i4})$ in the CG. Let op_{i1}, op_{i2} and op_{i3}, op_{i4} be two conflicting gen-use pairs. It is sufficient to prove that the path segment $(op_0, \ldots, op_{i1}, \ldots, op_{i2})$ is false, since it implies that the complete path must be false. We do not include paths which contain a conflicting gen-use pair for the same register when generating the *RG*, since the *CSG* already consists of paths with a conflicting gen-use pair for a given shared register. The reachability subgraph *RG* is identified with another constrained DFS routine, DFS_RG, details of which are omitted in this paper. The procedure Extract_Marked_Nodes identifies nodes marked by DFS_CSG and DFS_RG, and adds the marked nodes and edges between the marked nodes to create the CG.

Consider the schedule shown in Figure 1(a) and the mapping of it's variables $raddr1_1$ and $raddr2_1$ to register R_1. There are several gen-use conflicts for register R_1. One conflicting pair is given by op_7 which assigns to $raddr1_1$ and op_{21} which reads $raddr2_1$. The CSG consisting of paths with conflicting gen-use operations involving $raddr1_1$ in op_7 includes the following operations $(op_6, op_7, op_8, op_9, op_{19}, op_{20}, op_{21})$ and edges between the operations, while the corresponding RG consists of operations $(op_1, op_2, op_3, op_4, op_5)$ and edges between them. The CG which is a union of the CSG and RG is shown in Figure 1(b).

To prove that a gen-use conflict is benign, we need to prove that the paths in the graph CG are false. Thus, we are primarily interested in relationships between operations which control the execution of the path such as op_5 and op_9 in Figure 1(b). Any operation which does not affect the control conditions in the CG can be removed from the subgraph, except for the operations which cause the gen-use conflict, thus reducing the size of the subgraph. Consider the subgraph of Figure 1(b). Since op_6 does not affect any conditions in the subgraph, so it can be removed from the subgraph. For the subgraph of Figure 1(b), the pruned subgraph is shown in Figure 1(c). The subgraph pruning is implemented by the function Prune_SubGraph. Prune_SubGraph first creates data dependency arcs between CG operations op_i and op_j if the result of op_i is used by op_j. Subsequently, a DFS using the data dependency arcs is used to identify all operations which affect the control conditions in CG.

The pseudo-code for the algorithm verifying the validity of a register mapping is given in Figure 3

```
Procedure CHECK_REG_MAPPING(SchedG, RegMapping)
  opset := all op_n with gen(x_i), |M_i| > 1
  for each op_n in opset {
    DFS_CSG(op_n, x_i);
    DFS_RG(op_n);
    CG := Extract_Marked_Nodes(SchedG);
    PG := Prune_SubGraph(CG, op_n);
    if (Benign_Mapping(PG, op_n) ≠ TRUE)
      Declare mapping to be invalid. }
  Declare mapping to be valid.
```

Figure 3: Algorithm for verifying validity of register mapping

2.3 Verifying the Benignness of a Gen-Use Conflict

The subgraph extracted for each gen-use conflict encapsulates all the paths which must be proved to be unsensitizable (false) for the gen-use conflict to be benign. Doing the analysis by simulation or other means on a path by path basis is obviously not viable given the presence of a large number of paths and loops. We use symbolic model checking techniques in Computation Tree Logic (CTL) [7, 5] for the purpose. Without going into the details of CTL symbolic model checking, suffice to say that it allows us to check properties like *if some specific event happens in a state, another specific event will never happen in the future*. This is exactly the type of property we wish to check on the subgraph since we would like to ensure that *gen* is never followed by *use*. The approach is called *symbolic* since it effectively builds a single BDD-based representation for the entire state transition relation. The resulting analysis is performed implicitly on all paths together rather than on a path by path basis. To make the symbolic model checking viable, it might be necessary to abstract the bit-width of the arithmetic operations in some cases. In the future, it will also be possible avoid this abstraction by using model checking techniques that integrate efficient techniques for modeling arithmetic with symbolic model checking.

In practice, the subgraph for each gen-use conflict is generated in Verilog syntax. Two additional state variables called `gen` and `use` are introduced in the verilog. `gen` and `use` are set to 1 when their corresponding *gen* and *use* events take place. The Verilog code is compiled into the VIS symbolic model checking system [5]. In VIS, we check the property $AG(gen == 1 \rightarrow AG(use == 0))$ (please refer to [5] for CTL syntax). The property states that if in any state the variable `gen` becomes 1, then the variable `use` must be 0 in all subsequent states.

The major contribution of our work is that by dividing up the task of equivalence checking into the task of checking multiple simple assertions, and by abstracting out the irrelevant portions of the design in checking each assertion, we have significantly simplified the task that must be performed by the symbolic model

checker.

2.4 Verifying Intra-Cycle Equivalence Between the Schedule and its RTL Implementation

We perform the intra-cycle equivalence check state-by-state, *i.e.*, for each state S_i in the schedule, we prove that the computations performed in S_i (Sch_i) are equivalent to those performed in the RTL implementation in the same state. In doing so, we exploit the fact that while the number of states in the complete RTL (control and data path) circuit can be very large, the number of schedule states (control steps or control states) in scheduled behavioral descriptions is typically limited [1]. We assume that the register variable R_{state} in the RTL circuit, corresponding to the schedule state variable $State$ is identified, and that the encoding (mapping of symbolic values of $State$ to Boolean values of R_{state}) is known. Without a knowledge of the controller state encoding, the problem becomes significantly more complex and may, in general, require sequential FSM checking techniques.

One approach to establishing the equivalence of RTL and Sch_i could be to obtain gate-level netlists and use BDD or ATPG based equivalence checking techniques (*e.g.* [8]). However, these techniques may not be viable for repeated application (for each schedule state) on large designs, especially when they contain a composition of control logic and arithmetic, or when the structural similarity between the netlists is limited as a result of the application of resource sharing.

We present an alternative approach to checking the equivalence between RTL and Sch_i, that exploits the nature of the various transformations that are applied to the schedule in generating the RTL circuit. The typical optimizations that may be performed during this step consist of register and functional unit (operation) sharing, multiplexer generation, and control logic optimization. An important invariant that is preserved by the above optimizations is the atomic nature of word-level operators (including arithmetic operations, comparison operations, *etc*).

Our equivalence checking technique works at the RT level. First, the set of computations performed in state S_i of the schedule (Sch_i) is converted to an equivalent representation called the *structure graph*, which we formally define later. The structure graph corresponding to Sch_i (the RTL circuit) is called SSG_i (RSG). For example, the structure graphs for state $s2$ of the schedule of Figure 1(a) and the complete RTL implementation are shown in Figures 4 and 6, respectively. We verify that SSG_i is equivalent to RSG when the state variable of RSG is set to the encoded value of state S_i. The process is repeated for every state of the schedule.

Our algorithm for proving equivalence of structure graphs is based on a symbolic simulation of RSG and SSG_i. However, a key difference of our approach from the gate-level combinational equivalence checking approaches is that *we leave the known-good macro-blocks uninterpreted, thus avoiding reasoning about them or building representations for their functionality.* That is achieved by using the notion of *conditional equivalence relationships* between signals in RSG and SSG_i. A similar symbolic simulation approach was presented in [9] for modeling and evaluation of datapaths for implementing a given DFG. The reader might also wish to explore the similarities and differences between our approach and the approaches in [10, 11]. The algorithm starts with equivalence relationships between input variables. It then propagates equivalence relationships forward through the structure graphs until the outputs are reached, and checks for *unconditional equivalence* between the output signals of the RSG and SSG_i.

We apply the following pre-processing refinements in order to enable the comparison of Sch_i against RTL:

- The occurrence of each register variable x_j in Sch_i is replaced with the RTL circuit register variable R_j to which it is mapped. Thus, Sch_i and RTL now use the same set of register variables.
- Register variables are re-named when they appear as the target of an assignment operation, in order to distinguish between the present cycle and next cycle values. Whenever a register variable v appears on the left hand side of an assignment, we re-name it to v_next. This is done for both Sch_i and RTL.
- The encoding of the symbolic variable $State\_next$ is applied to replace $State\_next$ in Sch_i with a vector of Boolean variables. All assignments of symbolic state constant values to $State\_next$ in Sch_i are replaced with the corresponding constant bit-vectors. This is done to enable comparison of the next-state control logic in the RTL circuit against the state transitions specified in the schedule.
- The variable R_{state} in RTL is set to the constant encoded value corresponding to the state under consideration (S_i). This is done since we are only interested in the computations performed by the RTL in state S_i.

We next define *structure graphs* to represent the computations in the schedule and RTL, and outline their generation.

Definition 6 (Structure Graph) *A structure graph is a directed graph $G = (V, A)$, where the set of vertices V represent hardware components that execute the operations of the specification, and the edges represent the structural connectivity of the components. A vertex $v \in V$ has a type attribute, which may assume the following values: IN (to represent primary input variables and current cycle values of register variables), OUT (to represent primary output variables and next cycle values of register variables), OP (to represent arbitrary word-level operators, including arithmetic operators and comparison operators), $LOGIC$ (to represent the control or random logic), and MUX. The edges in the structure graph are annotated with their bit-widths.*

The process of constructing a structure graph from a set of computations is similar to inferring hardware structures from Hardware Description Languages (HDLs). IN and OUT nodes are created to represent primary input and output variables, constant values, and present and next cycle values of register variables. OP nodes are created corresponding to assignment operations that involve word-level computation and conditional operations (*e.g.* comparison operations, case operation, *etc.*). The use of a Boolean operator on single bits or bit-vectors results in the creation of $LOGIC$ nodes in the structure graph. MUX nodes are constructed when different assignment statements assign to the same variable, under different conditions. The outputs of the OP or $LOGIC$ nodes

[1] The state transition graph (STG) for the schedule is typically explicitly specified by the designer or generated by the scheduling algorithm.

that correspond to these conditions are used as select inputs to the MUX node to decide which assignment is executed in a given clock cycle. For example, consider the computations performed in state $S2$ of the schedule shown in Figure 1. The corresponding structure graph is shown in Figure 4(a).

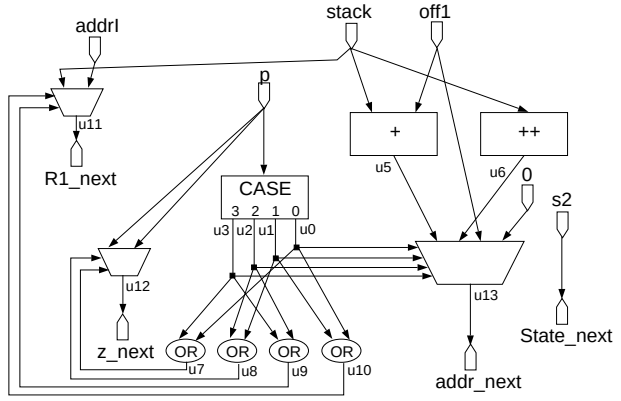

Figure 4: Structure graph for state $S2$ in the scheduled behavioral description of Figure 1

Our algorithm for equivalence checking of structure graphs exploits the following assumptions:

- The atomic nature of the OP nodes is preserved when generating the RTL circuit from the behavioral description.
- If the RTL circuit instantiates macro-block components from an RTL library, it is assumed that the library components have been verified during library development, hence they implement their specified functionality (*e.g.* a library component $ripple\_carry\_adder$ does perform the addition operation correctly).
- Arithmetic transformations (*e.g.* distributivity, replacing multiply by shifts and adds, *etc.*) are not performed. Note that while typical high-level synthesis tools do perform these transformations, they are performed prior to or concurrent with the scheduling step which generates the schedule.

Definition 7 (Conditional equivalence) *A signal v in RSG is said to be conditionally equivalent to signals $u_1, u_2, \ldots, u_n$ in SSG_i, if there exist corresponding conditions $c_1, c_2, \ldots, c_n$ (a condition represents a non-empty set of value assignments to input variables in SSG_i or RSG) such that under condition c_k, the value at signal v in RSG is guaranteed to be the same as the value at signal u_k in SSG_i. We use the notation ($v \cong \{(u_1, c_1), \ldots, (u_n, c_n)\}$) to represent conditional equivalence relationships.*

We use BDDs to represent the conditions involved in conditional equivalence relationships. In general, the conditions themselves may be expressed in terms of the input variables, and may involve the results of various arithmetic and conditional operations. However, we express conditions in terms of the outputs of OP and MUX nodes, in addition to IN nodes, which we collectively refer to as *basis variables*. In effect, BDDs are constructed only for the control logic (including the next state logic that feeds the PO, R_{state}_next, and the logic that determines which paths through the MUX nodes are sensitized or how multi-function FUs are configured).

Procedure COMPARE_STRUCTURE_GRAPHS(RSG, SSG_i)
  $Arr1$:= DFS_SORT(SSG_i)
  $Arr2$:= DFS_SORT(RSG)
  Identify basis variables in SSG_i;
  Symbolic simulate SSG_i to express non-basis vars in terms of basis vars;
  Construct equivalence lists for IN nodes in RSG;
  For_each_element($Arr2$, v) {
    switch(TYPE(v)) {
    case Mux:
      For each data input v_{fanin} of v {
        For each entry (u,c) in equivalence list of v_{fanin}
          ADD_EQUIVALENCE(v, u, $c \bigcup select\_cond$);
    case OP:
      For each pair of entries in equivalence lists of inputs {
        $cond$ = conjunction of conditions;
        if $cond \neq 0$ {
          identify corresponding OP vertex u_{op} in SSG_i;
          ADD_EQUIVALENCE(v, u_{op}, $cond$); } }
    case $LOGIC$:
      convert input lists into BDD nodes and propagate;
    case PO:
      If equivalence exists with corresponding PO in SSG_i
      and condition is 1
        continue;
      else
        return($Error$); } }
  return($Equivalent$);

Figure 5: Symbolic simulation algorithm for equivalence checking of structure graphs

The pseudo-code for the algorithm to compare SSG_i and RSG is shown in Figure 5. The algorithm starts with equivalence relationships for the IN nodes of RSG (these relationships are available since a 1-to-1 mapping exists between the IN nodes in RSG and SSG_i). The algorithm generates and propagates conditional equivalence relationships forward through the intermediate signals in RSG until the PO nodes are reached, and checks for *unconditional equivalence* between the output signals of RSG and SSG_i.

First, ordered sets $Arr1$ ($Arr2$) are populated to contain all the nodes in SSG_i (RSG) such that each node appears only after all the nodes in its transitive fanin. This is done by performing a backward depth first search traversal from the OUT nodes towards the IN nodes. Next the basis variables in SSG_i are identified as the outputs of PI, OP, and MUX nodes. A traversal through $Arr1$ is then performed, and for each node whose output does not correspond to a basis variable (*i.e.* each $LOGIC$ node), we get the BDD for the output of the node in terms of the BDDs at its inputs. We associate each RSG node with an equivalence list to represent the conditional equivalence relationships between its output and signals in SSG_i. An entry in the equivalence list is a pair (u,c) where u is an identifier for a SSG_i signal, and c is a BDD representing the conditions for equivalence. The correspondence between the inputs of SSG_i and RSG are used to create the equivalence lists for the IN nodes in RSG. Next, $Arr2$ is traversed, and each node is processed to propagate the equivalence lists from its inputs to its output. The techniques for propagating equivalence lists through OP, $LOGIC$, and MUX nodes are

explained below. When a PO node of RSG is reached, the algorithm checks to see if an equivalence has been established with the corresponding OUT node in SSG_i, and if the corresponding condition is a *tautology*. If not, the algorithm reports the RSG and SSG_i as not being equivalent. Only if unconditional equivalences are obtained for all the OUT nodes of RSG does the algorithm declare RSG and SSG_i to be equivalent.

Propagating equivalence relationships through OP nodes. Consider a two-input OP node v in RSG, whose inputs have equivalence lists $\{(x_1, c_1), \ldots, (x_m, c_m)\}$ and $\{(y_1, d_1), \ldots, (y_n, d_n)\}$. For each pair of entries (say, (x_j, c_j) and (y_k, d_k)) in the input equivalence lists, we check whether the conjunction of the BDDs representing c_j and d_k results in a constant 0. If not, we identify all corresponding OP nodes in SSG_i with inputs x_j and y_k, that perform the same operation (*i.e.* $+$, $-$, $<$, *etc*). For each OP node u_l identified, we add the entry (u_l,$c_j \bigcap d_k$) to the equivalence list of v.

Propagating equivalence relationships through $LOGIC$ nodes. Since control logic may be introduced or removed by the process of transforming the schedule to the RTL circuit, it may not be possible to find equivalence relationships for the outputs of $LOGIC$ nodes in RSG. Hence, rather than trying to compute equivalence relationships for $LOGIC$ nodes, we compute BDDs which represent their outputs as functions of basis variables in SSG_i. Each fanin of a $LOGIC$ node can have either an associated equivalence list (if it is a IN, MUX, or OP node), or an associated BDD (if it is another $LOGIC$ node). We first convert equivalence lists for the $LOGIC$ node inputs to BDDs as follows. An equivalence list $\{(x_1, c_1), \ldots, (x_n, c_n)\}$ is converted to the expression $\bigcup_{i=1}^{n} x_i \bigcap c_i$, for which we compute a BDD using the BDDs for the conditions c_i and BDDs representing the functions $f(x_i) = x_i$. Once we have BDDs for all the inputs of the $LOGIC$ node, we can compute the BDD for its output by composing them appropriately.

Propagating equivalence relationships through MUX nodes. Consider a MUX node v with n data inputs $v_1 \ldots v_n$. If v is a *decoded* MUX node (there is a dedicated select input corresponding to each data input), we identify the sensitization conditions $Sel_1, \ldots Sel_n$ for its data inputs as the BDDs for the nodes feeding the corresponding select inputs. If v is an *encoded* MUX node (the select conditions for each data input are specified as a combination of values at the select inputs), we compose the BDDs for the nodes feeding the select inputs appropriately to obtain the sensitization conditions. The entries in the equivalence lists at the data inputs of the MUX node are then propagated to its output by taking the conjunction of the equivalence conditions with the sensitization condition for the appropriate data input. For example, consider an entry (u_1,c_1) in the equivalence list of v_1. A corresponding entry (u_1, $c_1 \bigcap Sel_1$) is added to the equivalence list for v. Note that multiple data inputs of a MUX may have equivalence relationships with the same SSG_i signal, which may result in multiple entries with the same signal in the output equivalence list. The procedure for adding an entry to the equivalence list of a RSG signal avoids this by merging entries that refer to the same SSG_i signal, as explained below.

We next explain procedure ADD_EQUIVALENCE of Figure 5, which is used to add an entry to the equivalence list of a RSG node. When adding an entry (u, c) into an equivalence list $\{(u_1, c_1), \ldots, (u_n, c_n)\}$ for RSG node v, the procedure performs the following tasks:

- If u feeds the input of a MUX node in SSG_i whose output is u', the equivalence relationship between v and u is converted to an equivalence relationship between v and u', with equivalence condition $c \bigcap Sel_u$, where SEL_u is the condition for u to be sensitized to u', and is computed as described in the previous paragraph. If u' itself fans out to other MUX nodes, this step is repeated. This step ensures that when the algorithm reaches an OP node in RSG, all relationships between its inputs and inputs of corresponding OP nodes in SSG_i have been identified so that it can be processed just once.
- It first checks to see if signal u is the same as any of the signals $u_1 \ldots u_n$. If $u = u_i, i \in [1, n]$, the entry (u_i, c_i) is updated to $(u_i, c_i \bigcup c)$. This step helps reduce the size of equivalence lists, and thus improves the computational efficiency of the algorithm.

3 Experiments

There are three components to our verification system: (1) state-by-state checking, (2) gen-use conflict extraction, with the Verilog code of the corresponding subgraph as output, and (3) assertion checking using the VIS [5] symbolic model checker back end. We have implemented prototypes for (1) and (2) so that the state-by-state checking, and the gen-use conflict and subgraph extraction is totally automatic. There might be situations where no gen-use subgraphs are generated even in the presence of register sharing. In that case, VIS would not need to be called. We give our experiences with two illustrative example designs.

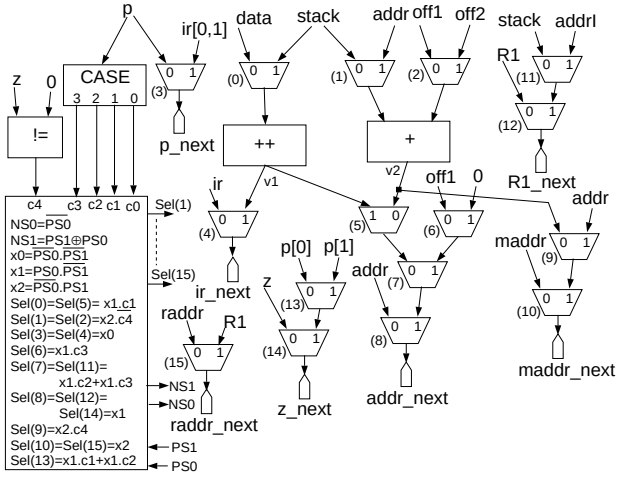

Figure 6: Structure graph for RTL implementation of the schedule shown in Figure 1(a)

The first example that we consider is the schedule of Figure 1. Its RTL implementation is shown as a structure graph in Figure 6. For the sake of clarity, all $LOGIC$ nodes have been converted to the Boolean equations inside the box. Also, the multiplexer marked (i) has its select input connected to signal $Sel(i)$ and its output is named $m(i)$. We focus on variables $raddr1$ and $raddr2$ which are mapped to the same register, $R1$, in the RTL implementation. Our algorithm for identifying gen-use conflicts (Figure 2)

identified four sub-graphs with gen-use conflicts, resulting from the assignment of $raddr1(raddr2)$ to in state $s1$ and the use of $raddr2(radd1)$ in state $s2$. In this example, all four cases are benign gen-use conflicts due to the correlation between the value of variable p used in the $CASE$ operation in $s1$ and the variable z used in the if construct in state $s2$. One such sub-graph, after the pruning of irrelevant operations, is shown in Figure 1(c). We generated Verilog code for the sub-graph, and the CTL assertion for verifying its benignness. The VIS symbolic model checking system [5] was able to easily prove the assertion to be true in 2.2 seconds on a Sun Ultra 10 workstation with 246MB memory.

We also performed the state-by-state equivalence check between the schedule (Figure 1) and the RTL implementation (Figure 6). We illustrate the process for state $s1$, whose structure graph, SSG_1, is shown in Figure 4. For each of the variables not shown in Figure 4 ($maddr$, $raddr$, ir, p), SSG_1 contains a PI node that directly feeds a corresponding PO node, requiring the variable to retain the previous cycle's value. The pre-processing steps assign the values 0 and 1 at the present state lines $PS1$ and $PS0$ in the RTL circuit, and replace the $State\_next$ and $s2$ in the schedule with the bit-vector $\langle NS1, NS0\rangle$ and the constant $\langle 0, 1\rangle$, respectively. The symbolic simulation of SSG_1 results in the construction of BDDs for the outputs of the four $LOGIC$ nodes ($u7 \ldots u10$) of Figure 4 in terms of the basis variables $u0 \ldots u3$. The equivalence lists of the IN nodes of the RSG are created based on the input corrrespondences with SSG_1. The conditions for the input equivalence relationships is set to the function 1. First, the algorithm considers the $CASE$ node in the RSG. The corresponding $CASE$ node in SSG_1 has an equivalent input signal, $c3 \ldots c0$ in RSG are set to be equivalent to $u3 \ldots u0$ in SSG_1. Next, the $LOGIC$ nodes corresponding to the Boolean equations in the box of Figure 6 are processed, leading to the following expressions: *NS1=1, NS0=0, Sel(0)= Sel(5)=u1, Sel(6)=u3, Sel(7) = Sel(11)=u2, Sel(13)=u1+u2, Sel(1) = Sel(2) = Sel(3) = Sel(4) = Sel(9) = Sel(10) = Sel(15)=0, Sel(8) = Sel(12) = Sel(14)=1.*

The remaining nodes in the RSG are evaluated leading to the following sequence of conclusions:

$m(0)$	$\cong$	$(stack, u1), (data, \overline{u1})$	$m(1)$	$\cong$	$(stack, 1)$
$m(2)$	$\cong$	$(off1, 1)$	$m(3)$	$\cong$	$(p, 1)$
$v1$	$\cong$	$(addr\_next, u1)$	$v2$	$\cong$	$(addr\_next, u0)$
$m(5)$	$\cong$	$(addr\_next, uo + u1)$	$m(4)$	$\cong$	$(ir, 1)$
$m(6)$	$\cong$	$(addr\_next, u2 + u3)$	$m(7)$	$\cong$	$(addr\_next, 1)$
$m(8)$	$\cong$	$-do-$	$m(9)$	$\cong$	$(addr\_next, u0)$
$m(10)$	$\cong$	$(maddr, 1)$	$m(11)$	$\cong$	$(R1\_next, 1)$
$m(12)$	$\cong$	$-do-$	$m(13)$	$\cong$	$(z\_next, 1)$
$m(14)$	$\cong$	$-do-$	$m(15)$	$\cong$	$(raddr, 1)$

It can be easily seen that propagating the equivalence lists through the PO nodes in RSG leads to the desired result.

The second example we would like to discuss is an implementation of a binary-tree sort algorithm. We cannot give the code here for lack of space. Suffice to say that the algorithm consists of two parts: the first part generates the sorted binary tree, while the second part walks the tree and outputs the data values in the correct order. The two parts follow each other in time, making it possible to share registers between variables whose life times are restricted to one of the two parts. There is little arithmetic in this algorithm except incrementing the input data index when data is read in, the < operator when two data are compared, and incrementing and decrementing the stack pointer during the tree walk. The state-by-state comparison is, therefore, quite straightforward and the schedule and implementation passed this test. The gen-use test, on the other hand, told us that a gen-use conflict did exist and that at least one path from gen to use was actually sensitizable. When we looked at the schedule carefully, we found that an assignment to one of the variables sharing the register and the use of that variable had been incorrectly placed in the same control state in the tree-walk part of the algorithm. This was, therefore, a case of a bad specification that got highlighted because it led to incorrect register sharing. After we corrected the schdule, the gen-use check also passed trivially since no gen-use subgraph was generated.

4 Conclusions

We have proposed a complete procedure for verifying register-transfer logic against its scheduled behavior in a high-level synthesis environment as the first step in our overall strategy to develop tools for validating a structural RTL implementation against its highest level initial description. We believe it is the first such verification procedure that is both complete and practical. We use of knowledge of the scope of the synthesis steps to partition the equivalence checking task into that of proving multiple subproperties, some of which can be checked locally in each control state, while the others must be checked by checking simple assertions on the entire state space using a symbolic model checker. By checking only simple assertions at a time, and by abstracting out the irrelevant portions of the design in checking each assertion, we have significantly simplified the task that must be performed by the symbolic model checker. The entire process of identifying the assertions to check, performing the abstractions, and carrying out the checks is automatic. We believe that this practice of taking advantage of the limitations in the scope of each synthesis step can be used for validating against levels of abstraction higher than scheduled behavior.

References

[1] J. Gong, C. T. Chen, and K. Kucukcakar, "Multi-dimensional rule checking for high-level design verification," in *Proc. Int. High-level Design Validation & Test Wkshp.*, Nov. 1997.

[2] R. A. Bergamaschi and S. Raje, "Observable time windows: Verifying high-level synthesis results," *IEEE Design & Test of Computers*, vol. 8, pp. 40–50, Apr. 1997.

[3] S. Minato, "Generation of BDDs from hardware algorithm descriptions," in *Proc. ICCAD*, pp. 644–649, Nov. 1996.

[4] R. Camposano and W. Wolf, *High-Level VLSI Synthesis*. Norwell, Massachusetts: Kluwer Academic Publishers, 1991.

[5] R. K. Brayton *et al.*, "VIS: A system for verification and synthesis," in *Proc. CAV*, July 1996.

[6] R. A. Bergamaschi, "The Effects of False Paths in High-Level Synthesis," in *Proc. ICCAD*, Nov. 1991.

[7] J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan, and D. L. Dill, "Symbolic model checking for sequential circuit verification," *IEEE Transactions on Computer-Aided Design*, vol. 13, Apr. 1994.

[8] W. Kunz, "HANNIBAL: an efficient tool for logic verification based on recursive learning," in *Proc. ICCAD*, pp. 538–543, Nov. 1993.

[9] C. Monahan and F. Brewer, "Symbolic modeling and evaluation of data paths," in *Proc. DAC*, pp. 389–394, June 1995.

[10] R. Jones, D. Dill, and J. Burch, "Efficient Validity Checking for Processor Verification," in *Proc. ICCAD*, Nov. 1994.

[11] A. Goel, K. Sajid, H. Zhou, A. Aziz, and V. Singhal, "BDD Based Procedures for a Theory of Equality with Uninterpreted Functions," in *Proc. CAV*, July 1998.