Verification of VLSI circuits using LP *

Stephen Garland, John Guttag †
Laboratory for Computer Science
Massachusetts Institute of Technology

Jørgen Staunstrup ‡
Computer Science Department
Aarhus University

Abstract

We present an approach to reasoning about the functional behavior of circuits. The approach begins with a technique, Synchronized Transitions, for specifying circuits and culminates with a technique for constructing machine checked proofs that invariants are preserved. We also report on some successful experiments using the approach to verify properties of simple, but nontrivial, VLSI circuits.

1 Introduction

For many years engineers have used simulation to convince themselves that the circuits they design behave as intended. As circuits get more complex, it becomes tempting to augment simulation with formal proofs. Typically,

*This research was supported in part by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contract N00014-83-K-0125, by the National Science Foundation under grant 8706652-CCR, and by the NYNEX Corporation.

†Address: 545 Technology Square, Cambridge, MA 02139, USA
‡Address: Ny Munkegade, DK-8000 Aarhus C, Denmark

these proofs involve a large number of simple steps. Doing them by hand is cumbersome, boring, and prone to mistakes. Unless these proofs are machine-generated or machine-checked, there is little reason to believe them.

This paper describes a successful experiment to use a theorem prover, LP, to verify properties of VLSI circuits. We started with several circuits that had previously been verified by hand. We then tried to construct machine-checked proofs with the same structure as the original proofs. In the process of using LP to verify the circuits, we uncovered several minor errors in, and simplifications to, the original circuits and manual proofs.

Any formalized verification of a circuit must be based on an abstract description of the circuit. The choice of descriptive mechanism depends upon the intended use. Differential equations, for example, are useful in verifying physical properties such as power consumption, timing, or heat dissipation. The approach presented here is aimed at verifying functional properties of a design, and is based on describing the circuit as a parallel program, using Synchronized Transitions.

While the circuits described in this paper are not particularly complex, the experiments reported on yielded several interesting insights. These include the following:

- Even for simple circuits, one cannot rely on proofs that have not been machine-checked.

- Combined with Synchronized Transitions, the technique of invariant assertions [7] developed to verify safety properties of concurrent programs is useful for machine-aided reasoning about circuits.

- The verification process is quite sensitive to the exact way in which the problem is formulated. For example, proofs seem to work better when induction is done over the structure of the circuit rather than over time.

- Circuit verification seems more amenable to machine checking than traditional program verification.

- The style of mechanical theorem proving supported by LP seems well-suited to reasoning about circuits.

Sections 2–4 present an overview of Synchronized Transitions, LP, and proofs based on invariants. Section 5 contains two extended examples of

using LP to verify properties of circuits. Section 6 summarizes our results and draws some conclusions based upon them.

2 Synchronized Transitions

A VLSI chip consists of registers and combinational functions. To structure the design, the registers are grouped into state components changed by the combinational functions. In the Synchronized Transitions notation, state components are denoted by variables and combinational functions by transitions. In an electrical circuit all parts run continuously; this is reflected by letting all transitions run in parallel. Synchronization of these transitions is controlled by predicates on state components.

The computational model underlying the Synchronized Transitions notation is to view a circuit as a collection of asynchronously computing automata. Each step of a computation is performed by a *transition*, which may involve any number of automata. Viewed from any one of them, a transition takes the automaton from one state to another. From a more global perspective, the participating automata appear to make the transition simultaneously; therefore, it is called a Synchronized Transition. While one group of automata participates in one synchronized transition, other automata (or groups) may independently make other transitions. No clock (i.e., global time reference) is assumed.

Transitions specify state changes using an imperative notation similar to that found in many high-level programming languages. For example,

$$\mathit{TRANSITION}\ \mathit{doneleft}\ \langle \neg \mathit{reql} \rightarrow \mathit{grl} := \mathit{false} \rangle$$

says that *doneleft* is performed only when $\neg \mathit{reql}$ holds, and that it leads to a state in which $\mathit{grl} = \mathit{false}$.

To describe any nontrivial circuit, a large number of transitions is needed. Each transition corresponds to a piece of circuitry. Transitions are atomic, as indicated by the notation $\langle \ldots \rangle$; thus each transition appears to be executed indivisibly. All transitions are independent; thus, there is no global thread of control determining the order of execution. This is an abstraction of a circuit, which is capable of simultaneously executing all its parts.

The general form for defining a transition is

$$\mathit{TRANSITION}\ \mathit{name}(n_1, n_2, \ldots)\ \langle C(n_1, n_2, \ldots) \rightarrow A(n_1, n_2, \ldots) \rangle$$

This definition specifies three types of information.

- n_1, n_2, ... are the state variables used by the transition.
- $C(n_1, n_2, \ldots)$ is the *precondition* of the transition; it is a boolean expression on n_1, n_2, ... A transition can only be performed when its precondition is satisfied.
- $A(n_1, n_2, \ldots)$ is the *action* of the transition; it is an assignment that specifies the *state transformation* made by the transition. It may only change n_1, n_2, ...

A transition need not be performed immediately after its precondition becomes satisfied, and there is no upper bound on when it takes place. This is an abstraction for delays within a circuit. For example, $\langle true \rightarrow y := a \vee b \rangle$ describes an **or** gate. The precondition *true* specifies that the transition is always allowed to set the output to the **or** of the inputs; however, an arbitrary delay may elapse between a change in the inputs and a change in the output. In fact, other transitions could change the values of the inputs while a transition is enabled. Thus, for example, the precondition of a transition may become false without the transition having been performed.

2.1 An example

An *arbiter* is a frequently used circuit for providing indivisible access to some shared resource, e.g., a bus or a peripheral. The arbiter described here is implemented as a tree in which all nodes (including the root and the leaves) are identical. The arbitration algorithm is based on passing a unique token around the tree. An external process, connected to a leaf, may use the resource only when that leaf has the token.

Each node has three pairs of connections, one for its parent and one for each of its children. A connection pair consists of two signals, *req* and *gr*, standing for "request" and "grant." Such a pair is used according to the following four-phase protocol.

1. A node raises the request (*req* signal to the parent) to indicate that it wants to get the token.

2. When the grant is raised (*gr* signal from the parent) the node has the token, and it may pass it down the tree.

3. The token is handed back by lowering the request.

4. Lowering the grant implies the end of one cycle; i.e., now a new request can be made.

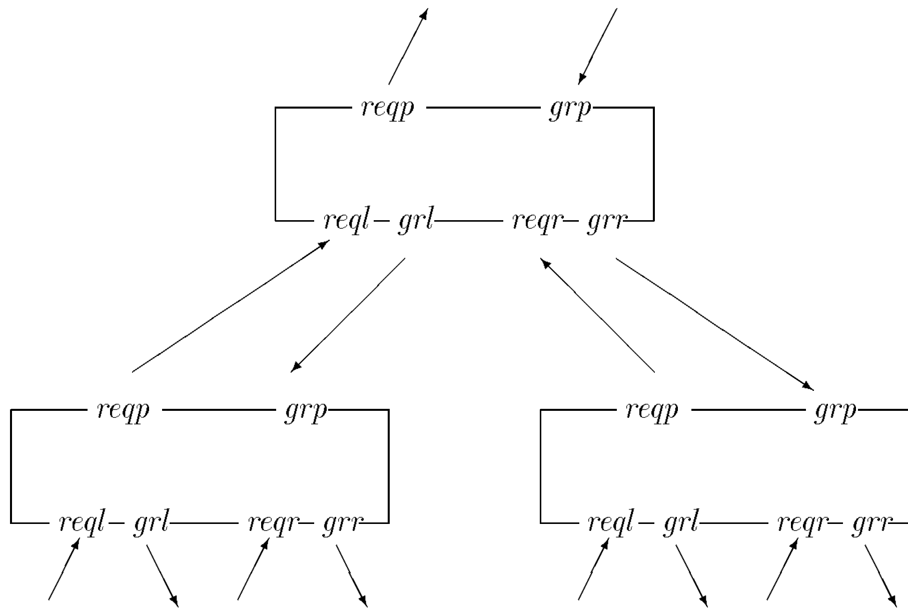


Figure 1: Two levels of arbiter tree

Figure 1 shows a few nodes and their interconnections. An external process requests the resource by setting *reqi* (where *i* is *l* or *r*, depending on the leaf to which the process has been assigned). When the corresponding *gri* is set (by the arbiter), the process may go ahead and use the resource. The signals *reqp* and *grp* are connected to *reqi* and *gri* at the next (higher) level of the tree. When both children of a particular node request the resource, it is given first to the left child; when that child releases the resource, it is given to the right child. The *reqp* and *grp* of the root node are connected. This means that the root is able to grant a request immediately.

The following invariant states that an arbiter node grants access to at most one child, and that access is only granted to a child when the parent has requested and been granted access.

$$\forall\, x \;\in\; nodes: \; I(x) = I^1(x) \wedge I^2(x) \wedge I^3(x)$$

$$\begin{array}{l} I^1(x):\; \neg(x.grl \wedge x.grr) \\ I^2(x):\; x.grl \vee x.grr \Rightarrow x.grp \\ I^3(x):\; x.grl \vee x.grr \Rightarrow x.reqp \end{array}$$

Note that $x.grp \Rightarrow x.reqp$ is not an invariant. The transitions needed for one node of the arbiter tree can be described as follows.

- *TRANSITION requestparent* $\langle \neg grp \wedge (reql \vee reqr) \rightarrow reqp := true \rangle$
- *TRANSITION grantleft* $\langle grp \wedge reqp \wedge reql \wedge \neg grr \rightarrow grl := true \rangle$
- *TRANSITION grantright*

  $\langle grp \wedge reqp \wedge reqr \wedge \neg grl \wedge \neg reql \rightarrow grr := true \rangle$
- *TRANSITION doneleft* $\langle \neg reql \rightarrow grl := false \rangle$
- *TRANSITION doneright* $\langle \neg reqr \rightarrow grr := false \rangle$
- *TRANSITION done* $\langle grp \wedge \neg grl \wedge \neg grr \rightarrow reqp := false \rangle$
- *TRANSITION init* $\langle reset \rightarrow grp, reql, reqr := false, false, false \rangle$

The Synchronized Transitions notation has been used on a variety of VLSI designs, both synchronous [9] and asynchronous [3]. There are strong similarities between Synchronized Transitions and UNITY, as developed by Chandy and Misra [1]. Both describe a computation as a collection of atomic conditional assignments without any explicit flow of control. UNITY proposes this as a general programming paradigm. The goal of Synchronized Transitions is more restricted, namely, development of special purpose VLSI circuits.

3 Overview of LP

LP is a theorem prover designed for use in the analysis of formal specifications written in Larch [4] and in reasoning about algorithms involving concurrency. In these applications, LP is most often used to debug a specification or a set of invariants. Hence, it is more important to report when and why a

proof breaks down than to try all avenues for pushing a proof through to a successful conclusion. For this reason, LP does not employ heuristics to derive subgoals automatically from conjectures to be proved. Instead, it relies largely on forward rather than backward inference, with the user rather than the program being responsible for inventing useful lemmas.

The basis for proofs in LP is a logical system containing the following types of user-supplied information.

- *Rewrite rules* of the form $t1 \rightarrow t2$, which LP uses to reduce terms to normal form and which LP keeps normalized with respect to one another.
- *Equations* of the form $t1 == t2$, which LP converts into rewrite rules by various *ordering* procedures that attempt to guarantee the termination of the resulting set of rules.
- *Assertions* about operators, e.g., that $+$ is associative and commutative. Logically, these assertions are merely abbreviations for equations. Operationally, LP uses them in equational term-rewriting to avoid non-terminating rules such as $x + y \rightarrow y + x$. These assertions are combined using the algorithm described in [10].
- *Deduction rules*, which LP uses to infer new equations from existing equations and rules. For example, the proofs in this paper apply the simple rule

  $$\textbf{when}\ \ x \,\&\, y == true\ \ \textbf{yield}\ \ x == true\ \ \ y == true$$

  to deduce the truth of two boolean expressions from the truth of their conjunction. More generally, deduction rules such as

  $$\textbf{when}\ \ \textbf{forall}\ z:\ \ z \in x == z \in y\ \ \textbf{yield}\ \ x == y$$

  are equivalent to universal-existential axioms such as set extensionality:

  $$\forall x \forall y [\, \forall z [(z \in x) = (z \in y)] \Rightarrow (x = y)\ ]\ .$$

- *Induction schemas*, which LP uses to generate subgoals to be proved for the basis and induction steps in proofs by induction [2].

LP provides both forward and backward rules of inference. Some forward rules are applied automatically each time new information is added to the

system; others are invoked explicitly by the user. Backward rules are invoked explicitly by the user. Among the inference rules in LP are the following.

- *Reduction to normal form:* a forward rule, invoked by users to prove theorems and applied automatically by LP to reduce rewrite rules, equations, and deduction rules in the system whenever a new rewrite rule, assertion, or deduction rule is added.
- *Induction:* a backward rule, for which LP generates subgoals (i.e., lemmas to be proved, usually with the aid of additional axioms) from the induction schemas.
- *Proofs by cases and contradiction:* backward rules, for which LP generates subgoals.
- *Instantiation:* a forward rule invoked by users, applicable to rewrite rules, equations, and deduction rules.
- *Deduction:* a forward rule, applied automatically.
- *Critical pairs:* a forward rule, invoked by users. Critical pairs between two rewrite rules are computed to derive equational consequences that do not follow by rewriting. Consider, for example, the rewrite rules

  1. $true \Rightarrow x \rightarrow x$
  2. $P(x) \Rightarrow Q(x) \rightarrow true$
  3. $P(0) \rightarrow true$

  The equation $Q(0) == true$ cannot be derived by rewriting, but it can be derived by computing the critical pair $true \Rightarrow Q(0) == true$ (i.e., by equating two different reductions of $P(0) \Rightarrow Q(0)$ obtained with rules 2 and 3) and normalizing it using rule 1. The Knuth-Bendix completion procedure [6, 8] is the closure of the critical pair computation.

LP also provides a wide variety of user amenities. There is extensive on-line help as well as facilities for naming objects and sets of objects, executing scripts, logging and replaying input, generating transcripts of sessions, taking checkpoints, getting statistics, and for displaying, adding, and deleting information. Section 5 contains several examples of the use of LP.

4 Invariance proofs

The goal of an invariance proof is to show that a predicate, I, on the variables denoting state components is always true. Since we assume that all transitions are atomic, this is equivalent to showing that I holds after every transition. Thus an invariance proof involves an implicit induction over the computation. If we partition the transitions into two classes, the basis and nonbasis transitions, then we must show two facts for all transitions A.

- If A is a basis transition, $\{true\}A\{I\}$. This means that no matter what values the variables have at the start of A, if A terminates, it will do so in a state satisfying I.
- If A is a nonbasis transition, $\{I\}A\{I\}$. This means that if A terminates when started in a state satisfying I, it will do so in a state satisfying I.

One has considerable latitude in setting up proofs within this framework. When using LP, we begin with a library of common types, e.g., booleans, natural numbers, and trees. Associated with each type is a set of operators, together with axioms and rules of inference, that can be used for reasoning about predicates built using those operators. This gives us the basic theory in which proofs are done.

We axiomatize each action A by equations that describe the relation between the state S^O before the transition (the old state) and the state S after the transition has been performed. For example, if A is the unconditional multiple assignment $\langle true \rightarrow a, b := b, a\rangle$, we axiomatize A by two equations

$A.1: \; a == b^O$

$A.2: \; b == a^O$

to describe the change of state and by additional equations such as

$A.3: \; c == c^O$

to specify that the values of other state variables are unchanged. Next, we define two instances of each invariant, one over the variables of S^O and one over the variables of S. For example, if the invariant I is $a \vee b$ we make the following definitions:

$I_{pre} == a^O \vee b^O$

$I_{post} == a \vee b$

To show that the nonbasis transition A preserves I, we show that I_{post} follows from the basic theory enhanced by the axiomatization of A and by $I_{pre} ==$

true. For the above example, this is easy. By $A.1$ and $A.2$, $I_{post} == b^O \vee a^O$. By I_{pre} and the commutativity of $\vee$, I_{post} reduces to *true*.

5 Extended examples

In this section, we present several examples that illustrate how LP can be used to verify properties of circuits. We show how to translate ordinary mathematical notation into that used by LP, how to use that notation to formulate properties of circuits, and how to use the proof tactics available in LP. Our aim is to present enough information so that readers can themselves attempt proofs using LP. Furthermore, we hope that readers find the formulation of a proof in LP almost as easy to understand as in ordinary mathematical notation, while at the same time finding it much more credible.

5.1 Proving the invariant for the arbiter tree

We first illustrate the use of LP by proving the invariant for the arbiter tree (Section 2.1), following the outline given in Section 4. To formulate the basic theory of the arbiter tree within the equational setting of LP, we introduce functions `l` and `r` such that `l(x)` and `r(x)` are the left and right children of the node `x`. To formulate the invariant, we introduce functions `gr` and `req`, and we rewrite the state variable $x.grp$ at node x as `gr(x)`, $x.reqp$ as `req(x)`, $x.grl$ as `gr(l(x))`, etc. With these conventions, the invariant, `Ipost`, which must be shown to hold at each node `x` in the tree after each transition, is defined by the following equation:

```
Ipost(x) ==
  not(gr(l(x)) & gr(r(x)))             &        % I1(x)
  ((gr(l(x)) | gr(r(x))) => gr(x))     &        % I2(x)
  ((gr(l(x)) | gr(r(x))) => req(x))             % I3(x)
```

Here `not`, `&`, and `|` are boolean operators for negation, conjunction and disjunction, and `%` is a comment delimiter. To formulate the invariant, `Ipre`, which is assumed to hold at each node `x` before the transition, we introduce functions `gr0` and `req0` to denote the values of the state variables before the transition occurs:

```
Ipre(x) ==
  not(gr0(l(x)) & gr0(r(x)))              &    % I1(x)
  ((gr0(l(x)) | gr0(r(x))) => gr0(x))     &    % I2(x)
  ((gr0(l(x)) | gr0(r(x))) => req0(x))         % I3(x)
```

Finally, we describe transitions such as `requestparent` by conjoining equations describing their preconditions and their actions, being careful to describe what each action leaves unchanged as well as what it changes. This "extra" work required for mechanical verification is usually overlooked, and is often a source of errors, in hand proofs.

```
% TRANSITION: (exists n)(all y) requestparent(n, y)
requestparent(n, y) ==
 not(gr0(n)) & (req0(l(n)) | req0(r(n)))  &  % precondition
 req(n)                                   &  % action
 (gr(y) = gr0(y))                         &  % unchanged
 ( (n = y) | (req(y) = req0(y)) )
```

In the definition of `requestparent`, `y` and `n` play very different roles. By an LP convention, `y` is a variable and `n` is a constant. Therefore, the equation `requestparent(n, y) == true` asserts that the `requestparent` transition has occurred at some node `n` and that the values of `gr0` and `req0` have not changed at any node `y` different from `n`. The technique of using constants to stand for existentially quantified variables is known as Skolemization and is frequently employed when using LP.

To show that the `requestparent` transition preserves the invariant, we first issue the LP commands:

```
thaw arbiter
set name invariant; add Ipre(x)
set name transition; add requestparent(n, y)
```

The `thaw` initializes LP with a previously frozen theory that contains axioms for booleans, numbers, and trees together with the above definitions of `Ipre`, `Ipost`, and `requestparent`. The next two lines add named axioms to the theory. LP treats these axioms as abbreviations of the equations:

```
Ipre(x) == true
requestparent(n, y) == true
```

The first asserts that the invariant holds at every node `x` in the tree. As noted above, the second asserts that a `requestparent` transition has occurred at node `n`.

The proof that `requestparent` preserves the invariant requires a modest amount of user guidance. Given the axioms just described, we prove `Ipost(x)` by issuing the following commands:

```
prove Ipost(x)
set name caseHyp; resume by cases x = n
critical-pairs caseHyp with transition  % for not(x=n)
```

The second line splits the proof into two cases, according to whether or not `x` is the node at which the transition occurs. This standard tactic is useful when axioms for transitions contain Skolem constants. For the first case, LP tries to prove `Ipost(c1)` using the axiom `c1 = n`, where `c1` is a fresh (i.e., not already in use) constant. For the second case, LP tries to prove `Ipost(c1)` using the axiom `not(c1 = n)`. This axiom makes it clear why LP must introduce the fresh constant. It allows LP to assume that `c1` is a particular node distinct from `n`, not that all nodes are distinct from `n`.

The proof for case `x = n` goes through immediately. The third line completes the proof by causing LP to discover that `req(c1) == req0(c1)` is a consequence of the axiom `requestparent(n, y)` and the case hypothesis `not(c1 = n)`. Computing critical pairs with case hypotheses is a frequently used tactic. The proofs that the other transitions preserve the invariant are similar in spirit (though sometimes a bit longer).

The invariants and transitions given here evolved during the formal verification using LP. Previously, the consistency of earlier versions of the invariant and transitions had been checked manually. To our surprise we discovered some problems with these when we tried to check them using LP. For example, the transition *done* was specified first as

$$\mathit{TRANSITION}\ \mathit{done}\ \langle grp \wedge \neg reql \wedge \neg reqr \rightarrow reqp := \mathit{false} \rangle$$

This seemed plausible, since lowering the request of both children means they are both done. However, lowering the request must not be propagated to the parent node until the grants of the children have been removed, and this version of *done* does not preserve the invariant. This became apparent when we attempted the machine-checked proof. A contributing factor in making this mistake was that the invariant and transitions used in the manual proofs

were somewhat more complicated than necessary. A study of the LP proof led us to remove the superfluous conjunct

$$reqp \Rightarrow (reql \vee grl) \ \vee \ (reqr \vee grr)$$

from the invariant, and to weaken the preconditions of several transitions.

5.2 Mutual exclusion in the arbiter tree

We also used LP to show that the entire arbiter tree ensures mutual exclusion, i.e., that there will never be two leaf nodes that have their grant signals true simultaneously. The proof proceeds by induction over the structure of the arbiter tree, i.e., by induction over the level of nodes in the tree. For this proof, we start with a theory that contains axioms for booleans, natural numbers, and trees together with the invariant for the arbiter tree (proved in the last section). Included in this theory are the following equations that describe a tree with root node `R` and infinitely many levels.

```
not(R = l(x))
not(R = r(x))
level(R) == 0
level(l(x)) == level(x) + 1
level(r(x)) == level(x) + 1
```

We also assert that all nodes in the tree (equivalently, all subtrees of the tree) are generated by the operators `R` (the root node), `l`, and `r`.

```
generators  R : -> node
            l : node -> node
            r : node -> node
```

This assertion provides LP with an inductive rule of inference. Now we issue the commands

```
prove ( (level(x) = level(y)) & gr(x) & gr(y) )  => (x = y)
set name indHyp; resume by induction x node
```

to begin an inductive proof of the mutual exclusion property. The proof requires some user assistance. LP first attempts to prove the basis case

```
((level(R) = level(y)) & gr(R) & gr(y)) => (R = y)
```

for the induction. This proof requires a subsidiary induction on `y`, which LP completes successfully after the user types

```
resume by induction y node
```

LP then generates the induction hypothesis

```
((level(c1) = level(y)) & gr(c1) & gr(y)) => (c1 = y)
```

(which asserts that mutual exclusion occurs at the level of the node `c1`) and tries to prove

```
((level(l(c1)) = level(y)) & gr(l(c1)) & gr(y)) => (l(c1) = y)
((level(r(c1)) = level(y)) & gr(r(c1)) & gr(y)) => (r(c1) = y)
```

(which assert that mutual exclusion occurs at the next level). Again it is necessary to instruct LP to induct over `y`. The basis case for this induction goes through by reduction to normal form. The two induction steps for `y` then combine with the two for `x` to yield four lemmas that must proved:

```
((level(l(c1)) = level(l(c2))) & gr(l(c1)) & gr(l(c2)))
       => (l(c1) = l(c2))
((level(r(c1)) = level(l(c2))) & gr(r(c1)) & gr(l(c2)))
       => (r(c1) = l(c2))
((level(l(c1)) = level(r(c2))) & gr(l(c1)) & gr(r(c2)))
       => (l(c1) = r(c2))
((level(r(c1)) = level(r(c2))) & gr(r(c1)) & gr(r(c2)))
       => (r(c1) = r(c2))
```

To prove the first lemma, we first partition the proof into cases with the commands

```
set name caseHyp
resume by case c1 = c2
resume by case gr(l(c1)) & gr(l(c2))
```

which serve to reduce the proof to one nontrivial case. Finally, we complete the proof by entering the command

```
critical-pairs caseHyp with invariant indHyp
```

to compute critical pairs between rewrite rules that describe the nontrivial case and rewrite rules that describe the invariant in the state before the transition and the induction hypothesis. LP orders these critical pairs automatically into rewrite rules, and these rewrite rules help reduce the lemma to an identity. The proof of the remaining three lemmas is identical in form.

5.3 A ring oscillator

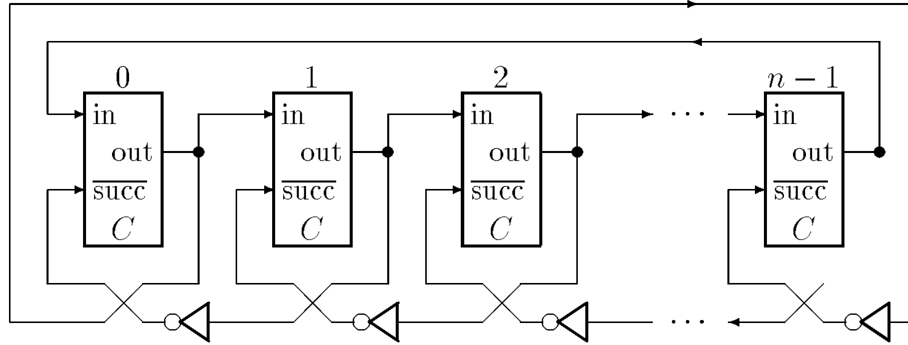
A ring oscillator can be understood as a wave traveling around in a loop. The crest of the wave is represented by a true bit in the elements of the loop and the trough is represented by a false bit. For proper operation, the rising edge of the wave must never overtake the falling edge, and the falling edge must never overtake the rising edge; i.e., the elements whose outputs are true must form a nonempty set of adjacent elements, as must the elements whose outputs are false. We formalize this invariant as

$$\begin{aligned}\exists i \exists j \forall x\, (\quad & st(i) \wedge \neg st(s(i)) \wedge \neg st(j) \wedge st(s(j)) \wedge \\ & [x = i \vee x = j \vee st(x) = st(s(x))] \ )\end{aligned}$$

where $st(x)$ is the input state of element x and $s(x)$ is the element following x in the ring (see Figure 2).

We now describe a ring oscillator using Synchronized Transitions and prove that it preserves the invariant. The circuit itself is a simple example of a self-timed circuit [3]. If the circuit were simply a ring of buffers, the oscillation would eventually die out. The problem is that propagation delays are different for rising and falling values and for different elements in the circuit. Thus, in a ring of buffers, either the trough will eventually overtake the crest, or the crest will overtake the trough. The ring of C elements shown in Figure 2 avoids this problem. The lower (succ) input to a C element ensures that the output of that element can become true only when the output of the successor element is false. This guarantees that the crest cannot overtake the trough. Likewise, the trough cannot overtake the crest.

The basic theory for the proof is that of a ring of size $n+1$, where $n > 1$. We formalize this theory in LP by using a constant `n` and a function `s` that maps each element in the ring to its successor; element `0` is the successor of element `n`.



$CELL\ ringosc(n : 3..)$ {the loop has n elements}
$VAR\ st :\ ARRAY[0..n-1]\ OF\ BOOLEAN$
$TRANSITION\ \ C(in, out, succ : BOOLEAN)$
{C element with one input inverted}
$\langle in \neq succ \rightarrow out := in \rangle$
$BEGIN$
$\|_{i=0}^{n-1}\ C(st[i],\ st[s(i)],\ st[s(s(i))])$
$END\ ringosc.$

Figure 2: Ring Oscillator

```
s(x) = 0 == x = n
s(x) = s(y) == x = y
not(x = s(x))
not(x = s(s(x)))
```

To reason about rings of a particular size an equation such as `n == 3` could be added. To reason about rings of arbitrary size, `n` is left unconstrained, which is what we do in our proofs.

We formalize the effect of a C transition at an element `a` as in Section 5.1: `st0(s(x))` is the output state of element `x` before the transition occurs, and `st(s(x))` is its state after the transition.

```
% TRANSITION c: (exists a) (all y) c(y)
c(a, y) ==
```

```
(not(st0(a) = st0(s(s(a)))))     &       % precondition
(st(s(a)) = st0(a))              &       % action
((y = s(a)) | (st(y) = st0(y)))          % unchanged
```

Since the invariant contains existential quantifiers, it requires some care to formalize. For the invariant, `Ipre`, we assume that there exist two elements, `i0` and `j0`, that begin the crest and trough of the wave. These are Skolem constants of the kind used in Section 5.1.

```
% invariant (assumed before transition)
% (exists i0) (exists j0) (all x) Ipre(i0, j0, x)
Ipre(i0, j0, x) ==
  st0(i0) & not(st0(s(i0))) & not(st0(j0)) & st0(s(j0)) &
  ( (x = i0) | (x = j0) ) | (st0(x) = st0(s(x)) )
```

To establish the invariant in the after the transition, we must show that there are two elements `i` and `j` that begin the new crest and trough.

```
%  invariant (must hold after transition)
%  (exists i) (exists j) (all x) Ipost(i, j, x)
Ipost(i, j, x) ==
  st(i) & not(st(s(i))) & not(st(j)) & st(s(j)) &
  ( (x = i) | (x = j) ) | (st(x) = st(s(x)) )
```

Figures 3 and 4 display the LP commands used to prove that the invariant is preserved. The proof begins in Figure 3 by assuming the basic theory, the invariant `Ipre`, and the description of the transition. It continues by proving some useful lemmas. The commands used to accomplish each proof are indented under the `prove` command that initiates the proof. When the arguments to these commands do not fit on a single line, they are terminated by a line containing a single period.

The third lemma provides information for the principal cases in the subsequent proof of the main theorem. It reveals both that a transition can occur even when it produces no change in the circuit and that, when a transition does produce a change, it must occur at the head of the crest or trough. Its proof illustrates two typical proof tactics in LP. First, when proving a conditional (i.e., a boolean term involving `if`) or an implication (i.e., a boolean

```
thaw ring
set name invariant;   add Ipre(i0, j0, x)
set name transition;  add c(a, y)

set name lemmas
prove (a = x) | (st(s(x)) = st0(s(x)))          % lemma 1
  critical-pairs ring with transition
prove not(i0 = j0) by contradiction             % lemma 2
prove                                           % lemma 3
  if(st0(a) = st0(s(a)), st(x) = st0(x),
        not(j0 = s(a)) & not(i0 = s(a)) )
  .
  set name caseHyp; resume by cases st0(a) = st0(s(a))
    resume by cases x = s(a)     % for case st0(a) = st0(s(a))
      critical-pairs caseHyp with transition
    resume by cases          % for case not(st0(a) = st0(s(a))
      i0 = s(a)
      j0 = s(a)
      not(i0 = s(a)) & not(j0 = s(a))
      .
```

Figure 3: Proof of invariant for ring oscillator, part 1

term involving `=>`), we often consider separately the case in which the hypothesis is true and the case in which it is false. Second, if we can identify a term that has special properties (e.g., `s(a)`, because the state changes only at `s(a)`), we often consider separately the cases in which that term is equal to other terms in the equation (e.g., to `x` when the transition does not change the state, or to `i0` or `j0` when it does).

The proof that the invariant is preserved (Figure 4) is divided into cases. Since we are trying to prove that `Ipost(i, j, x)` is true for *some* values of `i` and `j`, we help the proof along in each case by supplying appropriate values for `i` and `j`. When the transition does not change the state of any node (Case 1.1), the location of the wave does not change, and `i` and `j` are identical to `i0` and `j0`. When the trough advances (Case 2.1), `i` is the successor of `i0`,

```
set name theorem; prove Ipost(i, j, x)
  set name caseHyp
  resume by cases  st0(a) = st0(s(a))
  % Start Case 1.1: st0(a) = st0(s(a))
  add  i == i0  j == j0             % Choose values for i, j
  % Start Case 1.2: not(st0(a) = st0(s(a))
  prove (i = i0) | (a = j0)
    critical-pairs caseHyp with invariant
  resume by cases  a = i0  a = j0
    % Start Case 2.1: a = i0
    add  i == s(i0)  j == j0        % Choose values for i, j
    critical-pairs lemmas with transition invariant
    resume by cases  x = j0  x = s(a)  not((x=j0) | (x=s(a)))
      % Cases 3.1-2: x = j0, x = s(a) proved without help
      % Start Case 3.3: not((x=j0) | (x=s(a)))
      critical-pairs caseHyp with invariant transition
      resume by cases a = c1
      critical-pairs caseHyp with transition
    % Cases 2.2: a = j0
    add  i == i0  j == s(j0)        % Choose values for i, j
    critical-pairs lemmas with transition invariant
    resume by cases  x = i0  x = s(a)  not((x=i0) | (x=s(a)))
      % Cases 4.1-2: x = i0, x = s(a) proved without help
      % Start Case 4.3: not((x=i0) | (x=s(a)))
      critical-pairs caseHyp with invariant transition
      resume by cases a = c1
      critical-pairs caseHyp with transition
```

Figure 4: Proof of invariant for ring oscillator, part 2

and `j0` is unchanged. Likewise, when the crest advances (Case 2.2), `j` is the successor of `j0`, and `i0` is unchanged. In each of these cases, we compute critical pairs to derive consequences from equations that have been simplified by the definitions supplied for `i` and `j`.

We are not entirely happy with the explicit instantiation of Skolem constants in this proof. In some sense it compromises the integrity of the proof, since LP has no way of knowing that `i` and `j` are Skolem constants and thus play a different role from `n`. Future versions of LP will provide better facilities for direct proofs of existential statements.

We have constructed an alternate version of this proof in which explicit instantiation is not used. Instead the proof proceeds by contradiction. That proof, however, was harder to construct and harder to follow.

6 Summary and conclusions

We have presented a notation, Synchronized Transitions, for describing clockless VLSI circuits. This notation is designed to capture the parallelism inherent in these circuits. We have also presented a proof technique for reasoning about the functional behavior of circuits described using Synchronized Transitions. This proof technique is based on the preservation of invariants, and it exploits the atomicity of the transitions.

After giving a brief overview of a theorem proving system, LP, we showed how invariance proofs about circuits described with Synchronized Transitions can be formulated in a way that permits machine checking. This approach was illustrated by some simple, but nontrivial, examples. The examples were chosen to illustrate both the general structure of such proofs and the details involved in pushing them through LP. We believe that the similarity of the proof tactics used in these examples is not coincidental; i.e., these tactics seem applicable across a wide range of proofs of this kind. Hence, these tactics should provide models useful to others wishing to attempt mechanical verification of VLSI circuits.

On the whole we are quite encouraged by the experiences reported in this paper. It took us some time to push the first few proofs through LP. As we gained experience, however, it became easier. Our three main problems were the following:

- learning how to translate proof tactics used in the manual proofs into

tactics that could be checked by LP, e.g., instantiation of existentially quantified variables and setting up the appropriate inductions;

- remembering to include details that were "obvious" in the manual proof, e.g., remembering to say that something is not changed; and

- dealing precisely with relatively informal proof tactics such as "without loss of generality assume."

Some of the problems encountered in describing proofs might be alleviated if the logic of LP and the heuristics used by LP were more powerful. On the other hand, making LP more powerful in these ways might make it more difficult to find proofs. The point of a proof is often not to certify the correctness of a design but rather to shed light on why the design is not correct or on how it might be improved. It is important, therefore, that proofs fail in informative ways. As provers become more sophisticated, the causes of failure become more obscure. Furthermore, as circuits to be verified become more complex, the computational complexity of proofs becomes an important issue. When debugging the proofs presented here, it took less than four minutes of CPU time on a VAX 8600 to rerun an entire proof and discover why it failed. It is particularly important that the prover not take a long time to fail when it cannot prove the conjecture. As provers become more sophisticated, the amount of time they spend in unsuccessful proof attempts can be a serious problem.

Our experience has been that the additional effort of machine proofs is outweighed by their benefits. The principal advantage of machine proofs is that they are more reliable. They are harder than manual proofs primarily where manual proofs are vague, and therefore suspect. An additional benefit of constructing a machine proof is that it becomes easier to see exactly what is necessary to ensure the desired properties. This can, and did, lead to simplifications in one circuit and in the invariants used to reason about both. A final benefit of machine proofs is that they are easily repeatable. This allowed us to do regression testing when we changed the circuits or the invariants used to reason about them.

Acknowledgement

We are grateful to Jim Horning and Jim Saxe who made the initial experiments using LP to verify circuits.

References

[1] K. M. Chandy and J. Misra, *A Foundation of Parallel Program Design*, Prentice Hall, 1987.

[2] S. J. Garland and J. V. Guttag, "Inductive Methods for Reasoning about Abstract Data Types," *Proceedings of the 15th ACM Conference on Principles of Programming Languages*, 1988.

[3] M. R. Greenstreet, T. E. Williams and J. Staunstrup, "Self-Timed Iteration", *Proceedings from VLSI-87*, Vancouver, North Holland 1987.

[4] J. V. Guttag and J. J. Horning, "Report on the Larch Shared Language" and "A Larch Shared Language Handbook," *Science of Computer Programming* 6:2 (Mar. 1986), 103–157.

[5] C. A. R. Hoare, "An Axiomatic Basis for Computer Programming," *Communications of the ACM* 12 (Oct. 1969), 576–580.

[6] D. E. Knuth and P. B. Bendix, "Simple Word Problems in Universal Algebras," in *Computational Problems in Abstract Algebra*, J. Leech (ed.), Pergamon Press, Oxford, 1969, 263–297.

[7] L. Lamport and F. B. Schneider, "The 'Hoare Logic' of CSP, and All That," *ACM Transactions on Programming Languages* 6:2 (April 1984).

[8] G. L. Peterson and M. E. Stickel, "Complete Sets of Reductions for Some Equational Theories," *JACM* 28:2 (Apr. 1981), 233–264.

[9] J. Staunstrup, F. Barrett, M. Greenstreet, and P. Møller-Nielsen, "The Design of a Problem-mesh," *Proceedings from Comp-Euro 87, VLSI and Computers*, IEEE May 1987.

[10] K. A. Yelick, "Unification in Combinations of Collapse-Free Regular Theories," *Journal of Symbolic Computation* 3 (1987), 153–181.