Proof Strategies for Hardware Verification

Robert Eastham and Krishnaprasad Thirunarayan
Department of Computer Science and Engineering
Wright State University
Dayton, Ohio 45435

Abstract

Ascertaining correctness of digital hardware designs through simulation does not scale-up for large designs because of the sheer combinatorics of the problem. Formal verification of hardware designs holds promise because its computational complexity is of the order of number of different types of components (and not number of components in the design). This approach requires the specification of the behavior and the design in a formal language, and reason with them using a theorem prover. In this paper we attempt to develop a methodology for writing and using these specifications for some important classes of hardware circuits. We examine digital hardware verification in the HOL-90 environment. (HOL-90 is a proof checker written in Standard ML which assists in mechanically checking a formal proof of hardware correctness.) In particular, we analyze proofs for a variety of circuits, and develop proof strategies for combinational circuits and restricted sequential circuits. Overall, this approach makes the theorem proving task less tedious and provides guidance to the user in carrying out proofs.

I. Introduction

Circuit designs are getting increasingly large and complex. It is difficult to ensure that the devices are free of design errors. Verification of circuits is important in safety critical systems, remote/expensive systems, or mass produced systems. That is, where a failure would cause a loss of life, significant property loss, or costly redesign.

Formal hardware verification refers to designing the system and proving formally that it satisfies its specification for all possible inputs [Mel93] [Bar84] [Bir94]. There are three steps in accomplishing the goal of verifying circuits: specify the intent of the designer formally, define the implementation, and verify that the implementation meets the specification.

In contrast, simulation is the experimental process that aims to mimic the dynamic behavior of hardware as it evolves through time and is the most widely used method for testing designs. However, on medium to large size circuits, the number of test cases to simulate becomes unmanageable. To solve this problem, a small subset of test cases can be selected and the test results can then be extrapolated to show the correctness of a design [Bar84]. As a result, most complex designs are only partially validated since only a small percentage of the set of possible inputs is tested on.

Formal verification, which refers to the process of demonstrating design correctness using mathematical proof techniques has a computational complexity that depends upon the number of different kinds of components and not the total number of components. The behavior of a hardware design is described formally and proofs are used to verify that they meet specifications of intended behavior.

CH35934-96/0000-0451 $1.00©1996 IEEE

II. Formal Verification

For rigorous specification of hardware and for carrying out the formal proof, we need a language for describing the hardware and expressing propositions about it and a deductive calculus for proving propositions expressed in this language. The idea is to provide a structural description of the implementation and a behavioral description of the components and the design, and to prove that they are equivalent.

Various approaches to hardware verification have been based on algebraic and logic techniques. However, in general, all approaches use the same four steps [Mel93]:

Step 1. Write a formal specification S to describe the correct behavior of the device to be verified.

Step 2. Write a formal specification for each kind of primitive hardware component used in the device. These specifications are intended to describe the actual behavior of real hardware components.

Step 3. Define an expression D which describes the behavior of the design to be proved correct. The definition of D has the following general form

$$D = P_1 + \cdots + P_n$$

where $P_1, \ldots, P_n$ specify the behaviors of the constituent parts of the device and "+" is the composition operator which models the effect of wiring components together. The expressions $P_1, \ldots, P_n$ used here are instances of the specifications for primitive devices defined in Step 2.

Step 4. Prove that the device described by the expression D is correct with respect to the specification S. This is done by proving a theorem of the form

$$\vdash D \text{ satisfies } S$$

where 'satisfies' is some relation on specifications of hardware behavior. This correctness theorem asserts that the behavior described by D satisfies the specification of intended behavior S.

III. Background: Verification in Higher-Order Logic

A simple logic formalism may support easy to automate proof procedures and can be used to represent simple devices. However, complex devices may not be modeled easily. A powerful formalism may make very complex circuit relatively easy to model but the proofs can be very difficult. Higher-order logic is used because it is a very powerful formalism that supports reasoning about higher level abstractions.

Higher-order logic can be used to naturally specify the input and output signals of the hardware devices as functions of time and to recursively specify n-bit regular structures. It also allows the hierarchical and modular verification of systems, which is a requirement for complex devices [Gor93][Mel93].

Specification

The specification of a digital circuit describes the behavior of the circuit. It consists of specifying the relationship between the input signals and the output signals. A specification treats the circuit as a black box and indicates what can be seen on the output lines in terms of current and previous values on the input lines. A specification is expressed formally in logic by a Boolean-valued term whose free variables correspond to these external wires.

The circuit behavior of a 1-bit fulladder can be described as:

```
fulladder_spec (a, b, cin, s, c) =
(s = ~(~ (a = b) = cin)) ∧
(c = (a ∧ b) ∨ ((~(a = b) ∧ cin)))
```

The variables *a, b, cin, s,* and *c* have logical type *bool*. They can have the truth values T and F to represent the logic values *true* and *false*. The term fulladder_spec (a b cin s c) describes a relationship between the inputs and the outputs.

A *partial specification* is a specification that does not describe the behavior of a device on every possible input value. This is useful for large or complex designs. A partial specification constrains the values that can occur on the external lines to those that are considered significant. In all other situations it leaves them unconstrained. For example, a partial specification of a 11-detector can be defined as:

```
detect11_spec(inp out) =
 ∀ t. (out (t + 1) = inp t ∧ (inp (t + 1)))
```

This specification is partial since the variable *out* remains undefined at t=0.

Implementation

The implementation of a circuit is a description of the structure (architecture) of the circuit. The specification lists the circuit's simpler components and how they are wired together. An implementation is composed by joining two devices together by connecting them at identically named external wires. Syntactically, this is done by forming a conjunction of the logical terms that specify the devices to be connected together. A model will have free variables that correspond to internal wires that are used only for internal communication between components. These variables are hidden from the external environment by existentially quantifying the free variables that correspond to the internal lines. The result is a term where the hidden variables are bound and no longer represent externally observable variables.

The fulladder circuit specified above can be implemented as:

```
fulladder_imp (a, b, cin, s, c) =
∃ p q r.
 xor2 a b p ∧ xor2 p cin s ∧
 and2 a b q ∧ and2 p cin r ∧
 or2 q r c
```

The internal wires (p, q, r) are existentially quantified and are hidden from the external environment.

Verification from Specification and Implementation

Given a specification and an implementation, a circuit can be verified by proving a proposition that asserts that the implementation in some sense satisfies the specification. The verification involves proving that for any size *n* and for all inputs and outputs.

```
∀n inputs outputs.
    implementation n ≡ specification n
```

Proving equivalence is then proving implication in both directions. That is,

```
∀n inputs outputs.
    implementation n → specification n and
∀n inputs outputs.
    specification n → implementation n
```

If both of these cases are proven correct then the original goal of equivalence is proven correct.

For a partial specification, we prove implication in the first direction only.

Higher-Order Logic

The HOL90 system supports higher order logic which is an extension of first-order predicate calculus. In higher-order logic, variables are typed and can range over functions and predicates. Functions and predicates can be the arguments and results of other functions and predicates. Higher-order logic has special functions denoting terms called λ-expressions. The logic allows one

to construct whatever mathematical tools are needed.

The HOL90 system supports proof by natural deduction. Assertions in the HOL90 logic are not individual formulas, but are *sequents* of the form (Γ, t), where Γ is a set of formulas called the *assumptions* and t a formula called the *conclusion*. A sequent asserts that if all the formulas in Γ are true, then so is t. A *theorem* is a sequent that has a proof. A *formal proof* is a sequence, each of whose elements is either an axiom or follows from earlier members of the sequence by a rule of inference. A theorem is the last element of a proof. A theorem can only be constructed by using operations exported by ADT theorems. Theorems are represented by the abstract data type **thm**. The only way to create a theorem is by generating a proof. The HOL90 system provides tools to help the user organize and generate proofs. These include inference rules, tactics, and tacticals.

Proofs can be generated in a goal-oriented or backward style using tactics. The goal-directed proof style starts with a formula (representing the goal one wants to prove) and then reduces this to sub-goals and its descendants until the problem is decomposed into trivial goals. This is in contrast to forward proof in which one works forward from axioms, using rules of inference, until the desired theorem is deduced.

IV. Hardware Proofs

Here we describe the first steps in defining proof strategies in HOL90. The types of circuits that we analyze are: fundamental combinational circuits, sequential circuits, and iterated circuits. These are described in detail in [Eas95].

Combinational Circuits

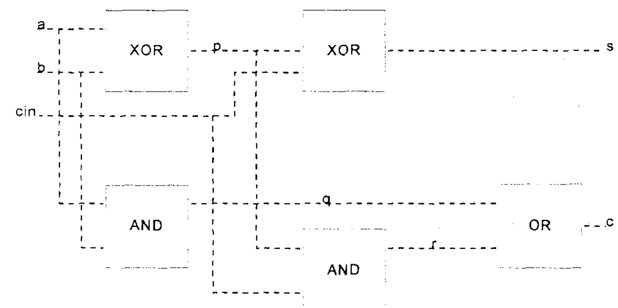
Circuits in this class have no feedback loops and there are no time dependencies. The output depends only on the current input. As an example (one which will be proven later), a fulladder falls into this class. Proofs were done on several circuits and were analyzed to see if there were any patterns in the proof that could be captured and thus automated for any circuit in this class.

Combinational circuit verification turns out to be proofs in first-order logic since the variables do not range over functions. This makes the verification problem convenient for automation.

A generic strategy for solving this class of circuits is:

- Step 1. Define the specification and the implementation.
- Step 2. Set the goal (implication or equivalence).
- Step 3. Remove universal quantifiers.
- Step 4. Rewrite the implementation and the specification with their definitions.
- Step 5. Replace the definitions of the primitives with their specification. This removes all hardware component names and leaves only an expression containing existential quantifiers, variables, and Boolean arithmetic operators.
- Step 6. Unwind and prune internal lines. That is, remove existentially quantified variables.
- Step 7. Prove the goal, which is a Boolean or an arithmetic expression.
- Step 8. Save the theory for possible reuse.

An example of a combinational circuit is a full-adder.



This can be specified and implemented as follows:

```
val fulladder1_spec = new_definition
          ("fulladder1_spec",
          --` ! a b cin s c .
             fulladder1_spec a b cin s c
             = (s = ~(~(a = b) = cin)) /\
         (c = (a /\ b) \/ ((~(a = b) /\ cin)))`--
          );

val fulladder1_imp = new_definition
          ("fulladder1_imp",
          --` ! a b cin s c .
            fulladder1_imp a b cin s c =
             ? p q r .
                  (XOR2 a b p) /\
                  (XOR2 p cin s) /\
                  (AND2 a b q) /\
                  (AND2 p cin r) /\
                 (OR2 q r c) `--
          );
```

Set the goal to be proved. Here we prove that the implementation is equivalent to the specification.

```
- set_goal ([], --` ! a b cin s c .
fulladder1_imp a b cin s c =
  fulladder1_spec a b cin s c `--);

val it =
 Status: 1 proof.
 1. Incomplete:
     Initial goal:
     (--`!a b cin s c.
      fulladder1_imp a b cin s c =
        fulladder1_spec a b cin s c `--)

  : proofs
```

The proof using HOL90 is as follows:

```
- e (REPEAT GEN_TAC
THEN REWRITE_TAC [fulladder1_spec,  fulladder1_imp,
xor2, and2, or2]
THEN CONV_TAC (DEPTH_CONV (CHANGED_CONV
(UNWIND_AUTO_CONV THENC PRUNE_CONV)))
THEN TAUT_TAC);
OK..
val it =
   Initial goal proved.
|- !a b cin s c. fulladder1_imp a b cin s c = fulladder1_spec a
b cin s c : goalstack
val it = "runtime: 0.510000s, gctime: 0.040000s." : string
Total inferences = 1566
val it = () : unit
```

As can be seen from the example circuit verification shown above fundamental combinational circuits have the same steps repeated over and over. Therefore, the specific HOL90 proof strategy for this class of circuits can be given as:

```
new_theory "x";

load_library_in_place taut_lib;

open Taut;

load_library_in_place unwind_lib;

open Unwind;

val x_spec = new_definition
    ...

val x_imp = new_definition
    ...

set_goal ([], --`! x y. x_spec x y = x_imp x y`--);

e (REPEAT GEN_TAC
THEN REWRITE_TAC [x_spec, x_imp, prim_1, prim_2, ...,
prim_n]
THEN CONV_TAC (DEPTH_CONV (CHANGED_CONV
(UNWIND_AUTO_CONV THENC PRUNE_CONV)))
THEN TAUT_TAC);

close_theory();
```

These steps are sufficient to prove every circuit that reduces to a tautology.

Sequential Circuits

Sequential circuits are digital circuits whose outputs depend not only on the current input but also on the sequence of past inputs.

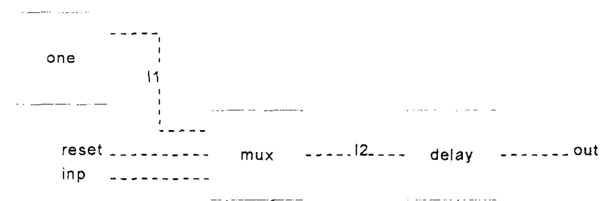
Strategies for verification of sequential circuits are much more complex than those for combinational circuits shown above. This is due to the fact that the inputs and outputs of these circuits are time dependent. Thus the variables

range over functions, and the expressions do not reduce to a simple tautology. The proofs are heavily dependent on the specific circuit.

One proof strategy for solving sequential circuits is given below:

- Step 1. Define the implementation and the specification.
- Step 2. Set the goal. If the specification is complete, then imp = spec. If the specification is incomplete, then imp ==> spec.
- Step 3. Remove universal quantifiers.
- Step 4. The definitions of the specification and the implementation are expanded. Rewrites are done on the specification and the implementation using rewriting tactics. The primitive gates are rewritten with their definitions.
- Step 5. The goal is then split into subgoals using knowledge about mathematical logic. This step may or may not be necessary depending on the circuit. For small circuits, this step is usually not necessary or else is very easily accomplished. For complex circuits, this may be a creative step needing application of induction rules, lemmas, and type specific rules, e.g., theorems about n-bit values, natural numbers, etc.
- Step 6. If this is an implicative proof, there should only be one subgoal to solve. Use rewriting to solve this subgoal. If this is an equivalence proof, then there is more than one subgoal. Use rewrites to solve top subgoal on stack.
- Step 7. If goals remain to be solved, then internal lines are removed by doing rewrites to eliminate any remaining existential variables. The internal wires are eliminated by unwinding and pruning as much as possible.
- Step 8. After this is done, the remaining steps of the proof are not as clear. The user may be left with recursive functions of the internal lines that cannot be unwound. The user must guide the HOL90 system to prove each of the subgoals. An automated theorem prover of first order logic or simplified higher order logic outside of the HOL90 system can be used to prove each subgoal separately. The circuit may be simple enough that the user can rewrite using rules of logic on the remaining subgoals using rewriting tactics.
- Step 9. Repeat step 7 and step 8 until all subgoals are solved.
- Step 10. Save theorem for further use in another proof or in a hierarchical proof.

An example of a sequential circuit is a resetable one-bit register. This can be specified and implemented as:




```
val rreg_spec =
  |- !reset inp out.
     rreg_spec reset inp out =
     (!t. (out 0 = F) /\ (out (t + 1) = reset t /\ T \/ ~(reset t) /\
inp t))
  : thm
val rreg_imp =
  |- !reset inp out.
     rreg_imp reset inp out =
     (?l1 l2. ONE l1 /\ MUX (reset,l1,inp,l2) /\ DELAY
(l2,out)) : thm
val rreg_exp = EXPAND_AUTO_RIGHT_RULE [one_def,
mux_def] rreg_imp;
val it = () : unit
```

The goal is stated as an equivalence.

```
- set_goal ([],--`!reset inp out. rreg_imp reset inp out =
rreg_spec reset inp out`--);

val it =
  Status: 1 proof.
  1. Incomplete:
     Initial goal:
     (--`!reset inp out. rreg_imp reset inp out = rreg_spec
reset inp out`--) :proofs
```

The proof in HOL90 is as follows:

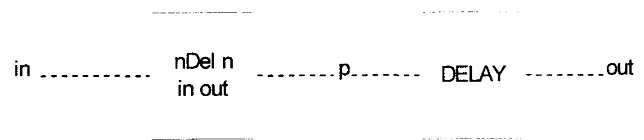
```
-e(REPEAT GEN_TAC
THEN REWRITE_TAC [rreg_exp, rreg_spec, delay_def]
THEN CONV_TAC (TOP_DEPTH_CONV num_CONV)
THEN REWRITE_TAC (theorem "arithmetic"
"ADD_CLAUSES");
THEN ALL_TAC THEN EQ_TAC THEN STRIP_TAC
THEN ASM_REWRITE_TAC[]
THEN EXISTS_TAC (Parse.term_parser '(\t.reset t \/ ~reset
t /\ inp t):num->bool')
THEN BETA_TAC
THEN ASM_REWRITE_TAC);
OK..
val it =
  Initial goal proved
|- !reset inp out. rreg_imp reset inp out = rreg_spec reset inp
out

val it = () : unit
```

The overall HOL90 strategy for sequential circuits can be given as:

```
- new_theory "x";

- add_theory_to_sml "arithmetic";

- val x_spec = new_definition
    ...

- val x_imp = new_definition
    ...

- set_goal ([], --`! x y. x_spec x y = x_imp x y`--);

- e (REPEAT GEN_TAC);

- e (REWRITE_TAC [x_spec, x_imp, prim1, prim2, ...,
prim_n]);

- e (CONV_TAC (TOP_DEPTH_CONV num_CONV));

- e (REWRITE_TAC [arithmetic theorems]);

(* Split the goal into subgoals. EQ_TAC THEN
STRIP_TAC is for equivalence and STRIP_TAC for
implications *)

- e (ALL_TAC THEN EQ_TAC THEN STRIP_TAC);

(* From here on, the proof strategy may diverge *)
- e (ASM_REWRITE_TAC[]);

- e (other rewrites to solve subgoal(s) )

- export_theory();
```

Iterated Circuit

An iterated circuit is a circuit which contains *n* cascaded identical modules. Iterated circuits are specified using recursive functions. The following *n*-element delay is an example of an iterated circuit.



This circuit can be specified and implemented as:

```
DELAY =
! (in:num->bool) (out:num->bool).
deln_spec n in out =
 ! t. (out(t+n) = in t) /\
     ((t<n) => (out t = T))

DELN_IMP =
  (deln_imp 0 in out = ! (t:num). out t = in t)
  /\
  (deln_imp (SUC n) in out
   = ? p. (deln_imp n in p) /\ (delay p out))
```

The goal is stated as follows:

```
set_goal ([], --` ! n in out. deln_imp n in out =
deln_spec n in out '--);
```

The steps required to prove the n-element delay stated concisely are:

```
- e (INDUCT_TAC THEN REPEAT GEN_TAC);
- e (ASM_REWRITE_TAC[DELN_IMP, DELN_SPEC,
DELAY]);
- e (REWRITE_TAC[NOT_LESS_0, ADD_CLAUSES]);
- e (REWRITE_TAC[DELN_IMP]);
- e (REWRITE_TAC[DELN_SPEC, DELAY]);
```

```
- e (CONV_TAC (TOP_DEPTH_CONV num_CONV)
THEN REWRITE_TAC [ADD_CLAUSES]);
- e (EQ_TAC THEN STRIP_TAC);
- e (INDUCT_TAC);
- e (ASM_REWRITE_TAC [LESS_0]);
- e (ASM_REWRITE_TAC [LESS_MONO_EQ]);
- e (EXISTS_TAC "(\t. out(SUC t)): num->bool") THEN
BETA_TAC);
- e (REPEAT STRIP_TAC);
- e (ASM_REWRITE_TAC []);
- e (FIRST_ASSUM (STRIP_ASSUME_TAC o
(REWRITE_RULE [LESS_MONO_EQ]) o
(SPEC "SUC t")));
- e (RES_TAC);
- e (REFL_TAC);
- e (FIRST_ASSUM (STRIP_ASSUME_TAC o (SPEC
"0")));
- e (FIRST_ASSUM MATCH_MP_TAC);
- e (MATCH_ACCEPT_TAC LESS_0);
```

At nearly every step, knowledge about the subgoal to be proven must be taken into account. This makes automation an extremely difficult problem.

V. Conclusions

In this paper, the concepts involved in formal verification of digital circuits were introduced.

Several hardware verification proofs were performed on various classes of digital circuits using HOL90 and the resulting steps were examined to see if there were common steps that could be abstracted.

Working with HOL90 has shown that although it is a very powerful tool for performing mathematical proofs, it is very difficult to use for a beginner. This is in part due to the "unfriendly" user interface and the lack of experience on the part of users for carrying out formal proofs. The next step to be accomplished in a proof is not always obvious, and even if the user knows what to do in the next step, telling HOL90 to do it is not always obvious.

The overall approach was to use tactics for combinational and sequential circuits which capture the repetitive reasoning found in hardware proofs. These proofs can be saved as theorems which can be used to solve larger circuits in a hierarchical manner.

Formal verification of combinational circuits can be fully automated by using the same sequence of steps. This is because there are no time dependencies on the circuits.

Sequential circuit verification can only be partially automated or automated for only a restricted subset of sequential circuits. If the original goal has not been proven, the remaining subgoals need to be "proved" manually. The existentially quantified variables and limitations of built-in primitives are the source of the problem. This is particularly true in circuits which contain feedback loops. In these circuits, there is a mutual recursion that cannot be automatically eliminated, in general.

VI. References

[Bar84] H.G. Barrow. VERIFY: A program for proving correctnes of digital hardware Design. In *Artificial Intelligence*, 24:437-491, 1984.

[Bir94] G. Birtwistle, S.K. Chin and B. Graham. *new_theory:"HOL"*. unpublished, 1994.

[Eas95] R. Eastham, *Proof Strategies for Digital Circuit Design in HOL90.* M.S. Thesis, Wright State University, 1995.

[Gor93] M. Gordon and T. Melham. *Introduction to HOL.* Cambridge University Press 1993.

[Mel93] T. Melham. *Higher Order Logic and Hardware Verification.* Cambridge University Press, 1993.