Embedding Hardware Verification within a Commercial Design Framework

Thomas Kropf, Ramayya Kumar and Klaus Schneider

Universität Karlsruhe, Institut für Rechnerentwurf und Fehlertoleranz
(Prof. D. Schmid)
P.O. Box 6980, W-7500 Karlsruhe 1, Germany

Abstract. A methodology for verifying complex circuits is presented, based on a strong coupling of design verification with the hierarchical design process. This goal has been achieved by integrating MEPHISTO, a tool for semi-automated hardware verification, into a commercial design framework. MEPHISTO decomposes the verification goal by a set of hardware-specific proof tactics and provides strategies for synthesizing pre-verified regular components. In case of erroneous implementations, MEPHISTO aids the designer in debugging the circuit by generating a counter model, i.e. input stimuli where specification and implementation behave differently.

1 Introduction

To guarantee reliable circuits especially in safety critical applications, and to avoid time consuming and costly redesigns, tools for checking design errors in circuits are mandatory. Usually, this is accomplished by specifying the desired functions and properties of the chip and proving formally that a given implementation behaves according to the specification. Although methods and techniques for formal hardware verification have been developed during the past few years, no tools usable outside academia have evolved yet. Hence simulation with appropriate stimuli is still the widely accepted means for validating circuit correctness. However, since exhaustive simulation is not feasible for large designs, design error freeness cannot be ensured.

Automated high-level synthesis is another means to derive presumably correct circuit implementations from a given specification [1]. However since the synthesis programs are themselves complex and do not have an underlying formal apparatus, the correctness of the circuits generated cannot be guaranteed. Furthermore, they are restricted to certain classes of circuits (e.g. signal processors) and are still inferior to manually designed circuits concerning timing efficiency and area requirements. Hence manual designs continue to exist and have to be checked formally for correctness.

In order to be accepted as an every day design tool, hardware verification approaches have to fulfill the following criteria:

- a high degree of automation
- applicability to real-world designs

- easily readable specification languages
- allow hierarchical verification
- integration within commercial design systems

Some of these criteria are fulfilled by one of the two main approaches to hardware verification, which is based on propositional logics and/or model checking techniques for finite state machines (FSMs). Since satisfiability checking in propositional logic as well as checking the equivalence of two FMSs is decidable, these approaches lead to fully automated tools [2]. Although tremendous progresses in the manageable problem size have been achieved [3] and approaches combined with test generation are possible [4], they are only capable of handling medium sized circuits at gate level. Moreover, the underlying formalisms are not capable of expressing design hierarchies and complex data types like natural numbers may not be directly used. Hence, these approaches are mostly restricted to the verification of controllers and are not yet usable in verifying complex designs.

Therefore, approaches based on first order logic like (Boyer-Moore [5]) or higher-order logic gain more and more importance. Since predicate logics are undecidable, full automation is not possible. Most approaches based on this logic are therefore based on interactive theorem proving systems like HOL [6], where most proof steps have to be triggered manually. However, it has been shown recently, that more automation is possible by integrating first-order automated theorem proving techniques within such an environment [7]. Moreover, in the context of hardware verification, the related proofs are in many cases structured in a similar manner, so that much automation is possible here also [7].

Higher-order logic is well suited for hierarchically describing circuits at different levels of abstraction [8]. Since, in contrast to FSM based techniques there are no size restrictions, this formalism is usable to describe and verify designs of realistic sizes, e.g. significant parts of the TAMARACK microprocessor have been verified [9].

In higher-order logic, hardware specifications may be formalized in a natural manner, i.e. comparable to known hardware description languages [8, 10]. Additionally, it is also possible to transform specifications given in a usual HDL like ELLA [11] into a representation in higher-order logic [12]. For these reasons, the work presented in this paper also relies on the use of higher-order logic for specifying and verifying hardware. However, the related interactive proof system HOL is only used as an implementation platform for different automation approaches and augmented by: FAUST, a first-order based theorem proving tool to automate tedious logical proof steps, and MEPHISTO, a hardware verification workbench which contains heuristics to automate hardware proofs [7].

In this paper a verification environment is presented which is usable by normal circuit designers to perform hierarchical hardware verification, hand in hand with the usual design process. For that purpose FAUST and MEPHISTO, coupled with HOL, have been integrated into the CADENCE design framework [13]. This leads to a close coupling of design and verification. In contrast to a post-design verification style, design errors are found immediately and time consuming redesigns are avoided. However, in general not all proof steps can be fully automated. Therefore a close linkage between design and verification is necessary, since our experience has shown that in most cases necessary manual proof steps closely reflect the designers creative ideas used to design the circuit modules.

The outline of this paper is as follows. First related work in the areas of designing correct circuits is presented. Afterwards, the principles of hierarchical hardware verification are introduced. Then the integration of the verification tool MEPHISTO into the commercial design framework CADENCE [13] is presented along with an example for error detection in circuits. The use of generic modules (regular modules with arbitrary bitsizes), stored in a library of preproven hardware components is elaborated. Using an example circuit, the whole process of designing a correct circuit is illustrated. The paper ends with results and conclusions.

2 Related Work

Although many techniques for hardware verification have been published, only few approaches integrate design systems and verification tools, to achieve a complete system for hierarchically designing correct circuits. However, they do not provide any support if the verification fails.

Fourman et al. have developed a methodology for designing correct circuits, using the LAMBDA theorem proving environment [14]. The approach is mainly based on formally guided synthesis, i.e. a given formal specification is successively decomposed until the circuit is given in terms of basic components. The necessary formal proofs are performed in interaction with a schematic entry tool. However, the refinements have to be carried out manually and the tool is not connected to a real design system.

Busch et al. have refined this approach for interactively designing circuits by applying formal transformation, aimed at the synthesis of non-hierarchical modules. The approach is well suited for regular designs as filters or signal processors, but does not exploit design hierarchies. Since the underlying system is LAMBDA, only minor steps are automated.

3 Basics of hierarchical Hardware Verification

To be able to perform formal reasoning about circuit behaviour, it must be formalized appropriately. Most approaches to hardware verification use a declarative or relational description of hardware components, first proposed by Hanna and Daeche [15].

3.1 Formalization of Hardware Behaviour

Lines and wires are described using functions over time, e.g. $y(t)$ indicates the time dependent value of a wire y. Time is usually formalized using natural numbers $t \in \mathbb{N}$. Depending on the abstraction level, time instants related to a certain value of N may be e.g. clock ticks of a synchronous system or time points of a fixed time schedule (e.g. 10 ns each). The related circuit signals (inputs and outputs) may be boolean values, bit vectors, natural numbers, etc. For example, the output y of a boolean gate is modelled by a function $y : \mathbf{N} \mapsto \{F, T\}$, where F and T denote the boolean values *false* and *true*, respectively.

The behaviour of modules and whole systems is described using higher-order predicates with functions as arguments. Predicates may be seen as characteristic functions indicating the set of admissible valuations at the inputs and outputs of the components to be specified.

A simple XOR-gate with two inputs a and b and an output y is specified by a predicate XOR_SPEC as follows:

$$\boxed{\forall a\, b\, y.\ XOR\_SPEC(a, b, y) \ := \ \forall t.\ y(t) \leftrightarrow \neg(a(t) \leftrightarrow b(t))}$$

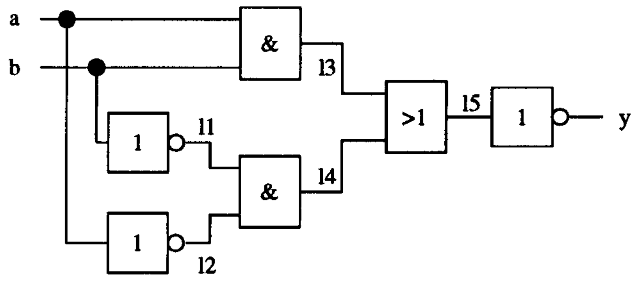

Fig. 1. Implementation of an XOR-gate

Figure 1 shows the realization of an XOR–gate by using AND, OR and NOT-gates. A predicate XOR_IMP describes its structure. It is assumed here, that the behavioural specifications of the modules (INV, $AN2$, $OR2$) are stored in a library.

$$\boxed{\begin{array}{l} \forall\, a\, b.\, XOR\_IMP(a, b, y) \ := \\ \quad (\exists\, \ell_1\, \ell_2\, \ell_3\, \ell_4\, \ell_5. \\ \qquad INV(b, \ell_1) \ \wedge INV(a, \ell_2) \ \wedge AN2(a, b, \ell_3) \ \wedge \\ \qquad AN2(\ell_1, \ell_2, \ell_4) \ \wedge OR2(\ell_3, \ell_4, \ell_5) \ \wedge INV(\ell_5,\ y) \end{array}}$$

Assuming a specification S and an implementation I in logic, circuit correctness is established by proving the validity of $I \rightarrow S$ in case of partial specifications and $I \leftrightarrow S$ for complete specifications [16]. For the XOR–gate the goal to be proven is given below:

$$\boxed{\forall a\, b\, y.\ XOR\_IMP(a, b, y) \ \leftrightarrow \ XOR\_SPEC(a, b, y)}$$

The hardware proofs are generally carried out using:

- domain dependent axioms (e.g. the behavioural definition of modules and structural implementations),

- domain dependent theorems (e.g. properties of components or complex data types) and
- domain independent axioms and theorems of higher-order logic like modus ponens or structural induction [17].

Advances in automating many of the underlying proof steps have been already achieved [7]. This has been implemented in the verification tool MEPHISTO which has been embedded within the HOL theorem prover. A typical HOL-session[1] for verification of the *XOR*-example is shown below, where '-' and '=' indicate the user's input:

```
                                                                   1
- new_theory "xor";
- new_parent "es2";
- new_definition("XOR_SPEC",
= --`XOR_SPEC((a:num->bool),b,y) = !t:num. y t = ~(a t = b t)`--);
- new_definition("XOR_IMP",
= --`XOR_IMP(a,b,y) =
= ?l1 l2 l3 l4 l5.
=                  INV(b,l1)  /\ INV(a,l2) /\ AND(a,b,l3) /\
=                  AND(l1,l2,l4) /\ OR2(l3,l4,l5) /\ INV(l5,y)`--);
-val goal = ([] ,--`!a b y. XOR_IMP(a,b,y) = XOR2(a,b,y)`--);
-val th = take_time TAC_PROOF (goal, NON_GENERIC_TAC);
 Time : 1.700000 secs
- save_thm ("XOR_CORRECT");
```

It can be seen from session 1 given above that the goal has been proved automatically by using a single tactic called **NON_GENERIC_TAC**[2]. The implementation *XOR_IMP* uses components from the formalized CADENCE library called 'es2'. This proven goal can be stored as a theorem called **XOR_CORRECT** and can be used later in hierarchical proofs.

3.2 Hierarchical Hardware Verification

Usually, realistic design implementations are too complex to be represented in a *flat* manner as described above. Hence, during the design phase, they are divided hierarchically into interconnected functional blocks. Each functional block may

[1] In MEPHISTO we have developed a set of tactics which are always applicable for automatically solving the goals at the Register Transfer level. In the HOL notation, the symbols $!, ?, /\backslash, \backslash/, \tilde{}, \backslash$ represent the logical symbols $\forall, \exists, \wedge, \vee, \neg$ and λ, respectively. The function **new_theory** starts a new theory for each verification step, and the function **new_definition** defines constants in this theory. The use of pre-proven definitions and theorems is done by the function **new_parent**. For better readability, unimportant HOL responses are suppressed in all sessions.

[2] Details of proving such goals are given in [7].

be represented again by smaller functional blocks, until each is represented by a structure of basic components, i.e. primitive cells (library components), which are not divided further.

This design inherent hierarchical structure may be also used to divide up the verification process, in order to reduce the problem size, and to identify erroneously designed components. To achieve this goal, each module has to be provided with a formal specification. This specification is then used to verify the correctness with regard to the specifications of the submodules and the interconnection information (figure 2) as described in the last subsection.

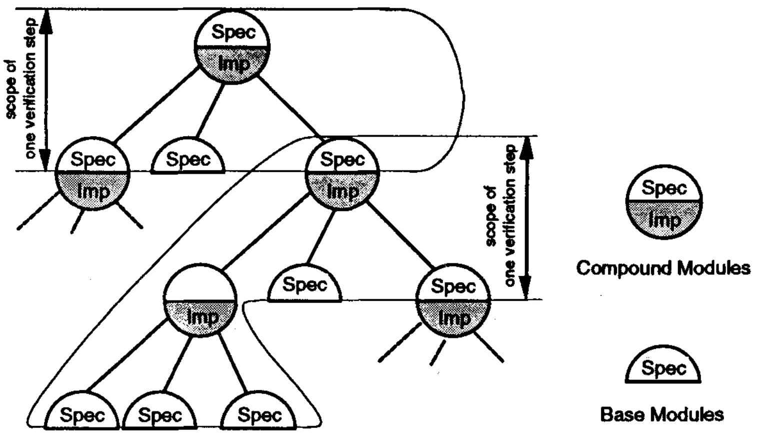

Fig. 2. Complete Hierarchical Decomposition

Proceeding this way, considerable savings in terms of formula complexity are possible, since module specifications are usually more compact than the formula describing the implementation (see e.g. formulas corresponding to XOR_SPEC and XOR_IMP).

As depicted in figure 2, it is not necessary that each compound module carries a formal specification. Especially if a module serves only for grouping submodules and there exists no concise semantical interpretation of the compound module, it can be cumbersome to write down a specification. In such cases, the interconnection structure has to be flattened out until each submodule corresponds to a specification.

A complete chip implementation is proven correct, if the top level chip specification as well as all intermediate module specifications have been proven valid. Only the specification of base modules are handled as axioms and need not be verified.

4 Coupling Design and Verification

To be executed efficiently, the hierarchical design process, as presented in the last section must be supported by appropriate tools. In our case, the design part is covered by the CADENCE system [13] and verification is supported by the MEPHISTO verification workbench, which leads to an automation of most parts of the proof process. Both tools are coupled by a hierarchy manager, which is responsible for converting the implementation data into formal descriptions in higher-order logic, resolving the design hierarchy by determing the scope of the actual verification step, managing the already proven verification goals and performing all the bookkeeping required for guaranteeing completely verified circuits. The structure of the complete system is given in figure 3.

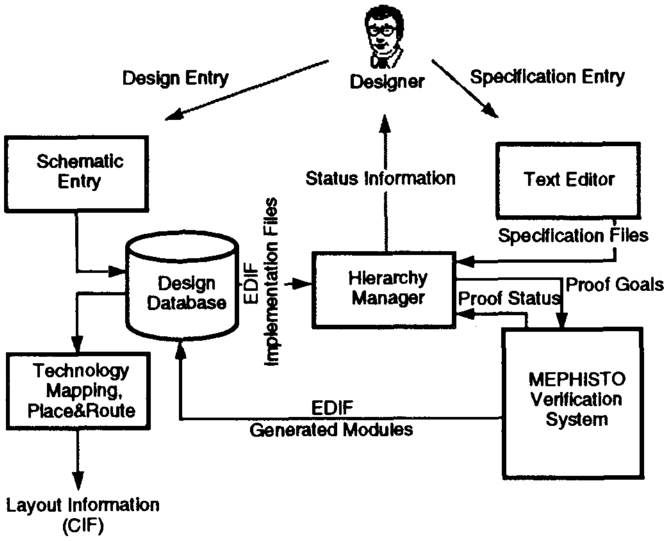

Fig. 3. System for Designing Correct Circuits

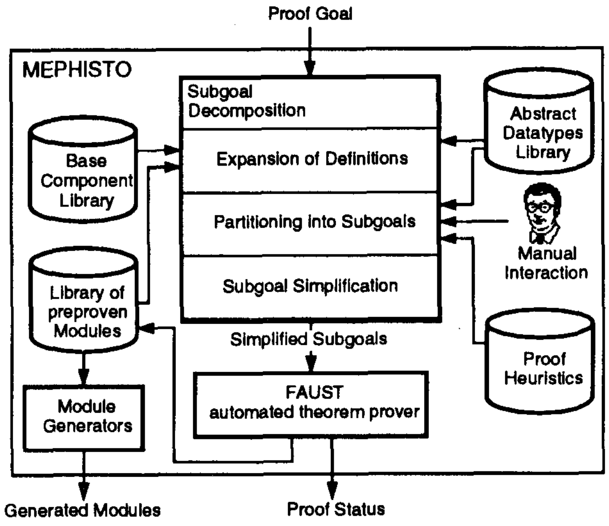

Fig. 4. The MEPHISTO Proof Tool

The hierarchy manager pairs the generated formal implementation with the appropriate specification to form a proof goal, which is then fed to the MEPHISTO verification system. MEPHISTO contains a set of hardware specific heuristics to decompose the verification goal into smaller subgoals, which are easier to solve [7]. The subgoals are then solved by an automated theorem prover FAUST[3], also contained in MEPHISTO. If the goal to be proven is not solvable, i.e. the implementation is erroneous, then a counter model[4] for the goal is generated for aiding the designer in identifying the erroneous component. MEPHISTO has been fully embedded into the HOL theorem proving environment [6], thus ensuring the formal correctness of all the steps.

After the verification process MEPHISTO reports the result of the verification attempt to the hierarchy manager. A more detailed drawing of MEPHISTO is given in figure 4.

The correct formal descriptions of the base components are given in a base component library. In our standard-cell based design system, this library has been manually built from the data sheets of the standard cell library.

For frequently used modules, which are not basic standard cells, a second library is maintained, in which already proven modules are kept, to shorten the verification process. This library is also used as the base of the module generators, described in the next section.

Using the *XOR*-gate as shown earlier, we shall now illustrate the design of a faulty *HALF_ADDER* (figure 5) and the subsequent error detection as reported by MEPHISTO, via a HOL session.

```
                                                                2
- new_theory "half_adder";
- new_parent "es2";
- new_parent "xor";
- new_definition("HALF_ADDER1_IMP",
=    --`HALF_ADDER1_IMP a b sum c =
=       EQUIV(a,b,sum) /\ (* This should be an XOR gate *)
=       AN2(a,b,c)`--);
- new_definition("HALF_ADDER_SPEC",
=    --`HALF_ADDER_SPEC a b sum c =
=       !t:num. (sum t = ~ (a t = b t) )/\ (c t = a t /\ b t) `--);
- val goal1 = ([], --`!a b sum c. HALF_ADDER1_IMP a b sum c =
  HALF_ADDER_SPEC a b sum c`--);
- val th = TAC_PROOF (goal1,NON_GENERIC_TAC);
-->PROOF FAILED!! COUNTER MODEL FOUND!!
```

It can be seen from session box 2 that the tactic **NON_GENERIC_TAC** is not able to solve a goal and outputs the message '**COUNTER MODEL FOUND**'. When this

[3] FAUST is based on an efficient variant of the tableau calculus [18, 7, 19].
[4] Details of the generation of counter models from tableau proofs can be found in [20].

```
                                                                        3
- pp_SHOW_ERROR goal1 std_out;
The following counter model has been found:
        a = (\t. F)     b = (\t. F)
The left hand side of the goal evaluates to:
!t. sum t /\ ~(c t)
The right hand side of the goal evaluates to:
!t. ~(sum t) /\ ~(c t)

- new_definition
=   ("HALF_ADDER_IMP",
=    --`HALF_ADDER_IMP a b sum c =
=       XOR_SPEC(a,b,sum) /\AN2(a,b,c)`--);
- val goal = ([], --`!a b sum c. HALF_ADDER_IMP a b sum c =
 HALF_ADDER_SPEC a b sum c`--);
- val th = take_time TAC_PROOF (goal,NON_GENERIC_TAC);
 Time : 0.250000 secs
val th = [] |- !a b sum c.
             HALF_ADDER_IMP a b sum c = HALF_ADDER_SPEC a b sum c :thm
- save_thm ("HALF_ADDER_CORRECT", th);
```

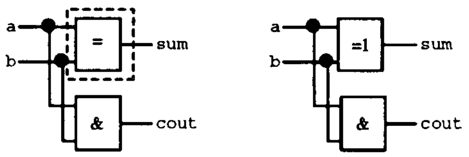

Fig. 5. The implementations of the *faulty* and *correct* *HALF_ADDER*

happens, the designer can call the function pp_SHOW_ERROR to identify the error (session 3). This function assigns a value to the inputs where the implementation differs from the specification. In the *HALF_ADDER* example, when the inputs a and b are set to F, then the value of the *sum* output in the implementation differs from that of the specification.

5 Generic Modules and Module Generators

One of the main advantages in using higher-order logic is the possibility of defining generic components, i.e. recursively defined regular n-bit structures like adders or shift-registers. Once proven correct and added to the system, they may be easily tailored to specific needs and automatically instantiated to a given bit-width by module generators. Since such components are often used in hardware designs, major savings in the proof effort result. To add such a component to the library, the following tasks have to be done:

1. specify formally the n-bit component using predefined operators [21].
2. specify formally and implement the 1-bit component
3. define the recursive construction algorithm (generalized implementation) for the n-bit component, using the implementation of the 1-bit component and predefined operators
4. derive the correctness proof for the n-bit component using MEPHISTO
5. add the component specification and implementation algorithms to the library

To use such a component, the parameter n has to be specialized to the required value, which is performed by a simple logical specialization (e.g. n to 16 bit) of the component specification. To achieve an implementation of the specialized component, the construction algorithm is directly used by the module generator to automatically derive a netlist, which is then used by the commercial design system to define the module in terms of standard cell basic components.
Using the *HALF_ADDER* that we have designed, we can design a 1-bit adder and an n-bit ripple carry adder as shown in figure 6. The specification, the implementations and the proof of correctness is shown in the HOL session 4.

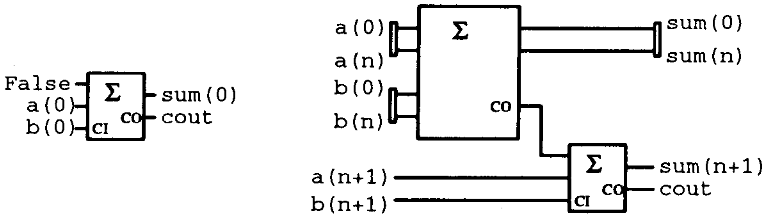

Fig. 6. The implementation of the 1-bit and n-bit ripple carry adder *ADDER_N*

In HOL session 4, the specification of *ADDER_N* has been given using the predefined n-bit operators **PLUS** and **PCARRY**. Such specifications can be interactively derived from specifications using operators on natural numbers. The implementation of the 1-bit adder uses the theory of **half_adder** as a parent. The HOL function called **new_prim_rec_definition** is used to define n-bit adder in a recursive manner. The goal can then be proved automatically by a single tactic called **GENERIC_TAC**, which applies induction over the bitsize n and performs the simplification of the goals in a manner similar to **NON_GENERIC_TAC**. Once the n-bit component has been proved, it can be instantiated by using the function called **INST_DEF**. This automatically defines a new constant called **ADD_2_IMP** in the current theory and proves the correctness theorem **ADD_2_CORRECT** which corresponds to the equivalence between a 2-bit instantiation of the n-bit adder (**ADD_N_IMP 2 a b sum cOut**) and the 2-bit adder (**ADD_2_IMP a b sum cOut**).

6 Design of Verified Circuits

In the following, the methodology for designing a correct circuit is illustrated using an example. Our approach closely reflects the normal design process. The designer has to additionally add a formal specification at each level of the design hierarchy, which is then used for the formal verification process. To design correct circuits in a top-down manner, the following steps have to be performed:

1. create a formal specification of the intended circuit behaviours
2. create an implementation at one hierarchy level
3. extract a formal implementation description
4. create formal specifications for the modules used in the implementation (in case that they are not base modules)
5. verify the correctness of the module versus its specification
6. repeat step 2.-6. for all modules until a module consists only of the base modules of the design system

Step 2 is performed using a commercial design system. The details of step 3 and step 5 are described in detail in [7].

4

```
- new_theory "adder";
- new_parents["es2","natural","half_adder"];
- new_definition("ADDER_N_SPEC",
=    --`ADDER_N_SPEC n a b sum cOut = !t:num.
=        (PLUS n (\x.a x t) (\x.b x t) (\x.sum x t) F) /\
=        (cOut t = PCARRY n (\x.a x t) (\x.b x t) F)`--);
- new_definition("ADDER_1_IMP",
=    --`ADDER_1_IMP cIn a b sum c =
=     ? l1 l2 l3.
=          HALF_ADDER_SPEC a b l1 l2 /\
=          HALF_ADDER_SPEC l1 cIn sum l3 /\ OR2 (l2, l3, c)`--);
- new_prim_rec_definition("ADDER_N_IMP",
= --`(ADDER_N_IMP 0 a b sum cOut =
=     ADDER_1_IMP (\t:num.F) (a 0) (b 0) (sum 0) cOut) /\
=    (ADDER_N_IMP (SUC n) a b sum cOut =
=        ?l1. ADDER_N_IMP n a b sum l1 /\
=             ADDER_1_IMP l1 (a(SUC n)) (b(SUC n)) (sum(SUC n)) cOut)`--);
- val goal = ([],--`!n a b sum cOut. ADDER_N_IMP n a b sum cOut =
    ADDER_N_SPEC n a b sum cOut`--);
- val th = take_time  TAC_PROOF (goal,GENERIC_TAC);
 Time : 6.490000 secs
val th = []|- !n a b sum cOut.
        ADDER_N_IMP n a b sum cOut = ADDER_N_SPEC n a b sum cOut :thm
- save_thm ("ADDER_N_CORRECT",th);
- take_time (INST_DEF (definition "-" "ADDER_N_IMP")) (--`2`--);
 Time : 1.040000 secs
```

We illustrate the design of verified circuits using the 'delta' circuit, whose informal specification is as follows: *out* $\leftrightarrow c < a + b$. This informal specification can be transformed into a formal specification *DELTA_N_SPEC*, by using the predefined operators over bitvectors as used in Section 5. Figure 7 gives a possible implementation (*DELTA_N_IMP*) of the circuit using an n-bit ***adder***, an n-bit ***less than*** circuit and some gates. The theory corresponding to the 'less-than' circuit can be generated in a manner similar to the 'adder' circuit.

```
                                                                     5
Theory:  adder
Definitions:
   ADDER_2_IMP  []|- !a b sum cOut.
       ADDER_2_IMP a b sum cOut =
       (?l1. (?l1'. ADDER_1_IMP (\t. F) (a 0) (b 0) (sum 0) l1' /\
                    ADDER_1_IMP l1' (a 1) (b 1) (sum 1) l1) /\
                    ADDER_1_IMP l1 (a 2) (b 2) (sum 2) cOut)
Theorems:
   ADDER_2_CORRECT
[]|- ADDER_2_IMP a b sum cOut = ADDER_N_IMP 2 a b sum cOut
   ADDER_N_CORRECT  []|- !n a b sum cOut.
       ADDER_N_IMP n a b sum cOut = ADDER_N_SPEC n a b sum cOut
```

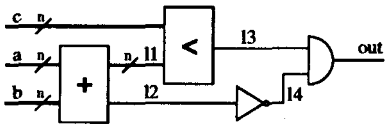

Fig. 7. The implementation of the *DELTA_N* circuit

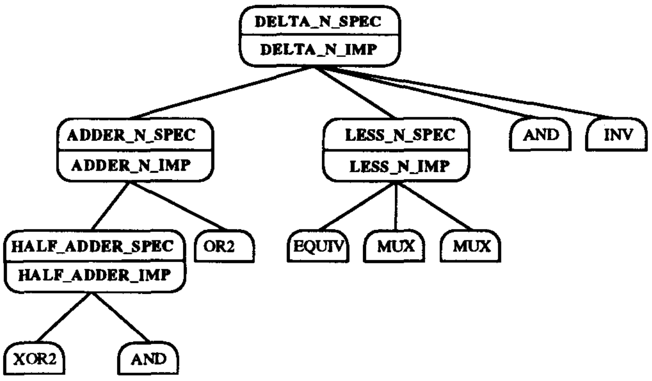

Fig. 8. The hierarchy tree for the *DELTA_N* circuit

The design hierarchy of the implementation of this circuit is given in figure 8. It can be seen in the HOL session 6, that in the verification of the equivalence between *DELTA_N_IMP* and *DELTA_N_SPEC*, only the specifications of the 'adder' and the 'less-than' circuits are used. Since the 'delta' circuit is also a parametrizable circuit, the tactic called **GENERIC_TAC** is sufficient for proving the correctness of the goal. It should however be noted that induction is not required in proving this goal.

```
                                                                    6
- new_theory "delta";
- new_parents["es2","natural","adder","less"];
- new_definition
=   ("DELTA_N_SPEC",
=    --'DELTA_N_SPEC n a b c out =
=       ?l1.!t:num. (PLUS n (\x.a x t) (\x.b x t) (\x.l1 x t) F) /\
=                   (out t = LS n (\x.c x t) (\x.l1 x t) /\
=      ~PCARRY n (\x.a x t) (\x.b x t) F)'--);
- new_definition
=   ("DELTA_N_IMP",
=    --'DELTA_N_IMP n a b c out =
=       ?l1 l2 l3 l4.
=         ADDER_N_SPEC n a b l1 l2 /\ LESS_N_SPEC  n c l1 l3 /\
=         INV (l2, l4) /\ AN2 (l3, l4, out)'--);
- val goal = ([],--'! n a b c out. DELTA_N_IMP n a b c out =
=    DELTA_N_SPEC n a b c out'--);
- val th = take_time  TAC_PROOF (goal,GENERIC_TAC);
 Time : 2.350000
- save_thm("DELTA_N_CORRECT",th);
```

The instantiation of a 4-bit delta circuit is shown in HOL session 7. By using the function **INST_DEF**, the hierarchy tree of the the delta circuit is searched to find the existence of the 4-bit instantiations of all the parametrizable components used in the design, and the required instantiations and their correctness theorems are first generated before the 4-bit delta circuit is created. Assuming that a 4-bit 'less-than' circuit exists in the parent theory 'less', a call to the function **INST_DEF**, generates HOL session 7.

Figure 9 corresponds to the implementation which has been automatically generated by the function **INST_DEF**. This function also generates the correctness theorems **ADDER_4_CORRECT** and **DELTA_4_CORRECT**. The time required for proving the correctness and finding the instantiations (2.76 secs.) illustrate the power of using higher-order techniques for exploiting the hierarchy and using concept of generic components.

```
                                                                      7
- take_time (INST_DEF (definition "-" "DELTA_N_IMP")) (--'4'--);
[opening /tmp/ADDER.sml]
val it =
  []|- !a b sum cOut.
         ADDER_4_IMP a b sum cOut =
         (?l1. (?l1'''. (?l1''.(?l1'.
                    ADDER_1_IMP (\t. F) (a 0) (b 0) (sum 0) l1' /\
                    ADDER_1_IMP l1' (a 1) (b 1) (sum 1) l1'') /\
                    ADDER_1_IMP l1'' (a 2) (b 2) (sum 2) l1''') /\
                    ADDER_1_IMP l1''' (a 3) (b 3) (sum 3) l1) /\
                    ADDER_1_IMP l1 (a 4) (b 4) (sum 4) cOut) :thm
[closing /tmp/ADDER.sml]
[opening /tmp/DELTA.sml]
val it =
  []|- !a b c out.
         DELTA_4_IMP a b c out =
         (?l1 l2 l3 l4.
           ADDER_4_IMP a b l1 l2 /\ LESS_4_IMP c l1 l3 /\
           INV (l2,l4) /\ AN2 (l3,l4,out)) :thm
[closing /tmp/DELTA.sml]
 Time : 2.760000
val it = () :unit
```

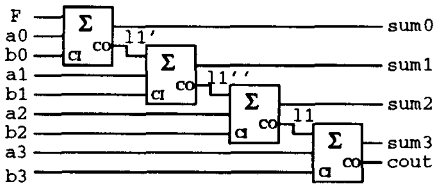

Fig. 9. The automatically generated implementation of a 4-bit adder

7 Experimental Results

We have used MEPHISTO to prove different small and medium sized circuits at RT-level, e.g. parity, serial adder, flipflops, shift registers, twisted ring counter, sequence detectors[5] in a matter of seconds [22]. Circuits at RT-level can be verified automatically [7]. We have also developed heuristics for simple generic components such as n-bit comparators or n-bit adders which allow the verification of these components without manual interaction. However, currently this does not hold for arbitrary generic components e.g. systolic arrays, since this requires the automation of induction and number theory proofs which is not possible in

[5] A detailed description of some of these circuits can be found in the appendix of [7].

general [23]. If complex data types such as stacks, lists or trees are used for the specification, interactive proof steps are required to provide lemmata necessary for the verification process. For these steps the HOL system is used directly in the usual interactive manner. This semi-automated approach allows to verify even complex hierarchically designed circuits. Currently we are validating our system by the verification of a real-sized chip designed in CADENCE.

8 Summary

In this paper, we have presented a methodology for designing real circuits correctly using a commercial design framework for semi-custom designs. It has been shown, how formal specifications are used to verify circuit implementations at different hierarchy levels. To allow a close interaction between the design and the verification tasks at each hierarchy level, the MEPHISTO verification system has been integrated as a design tool into a commercial system. MEPHISTO comprises the automatic generation of formal implementation descriptions in higher-order logic from designed circuit structures, bookkeeping facilities to supervise the hierarchical verification process, a set of transformation rules and heuristics to structure hardware proofs, counter model generation for faulty implementations, automatic module generator for generic components, as well as an automated theorem prover FAUST to free the designer from tedious formal proof tasks.

At the moment, specifications have to be written directly in higher-order logic. Various abstract data types are available to obtain concise and readable specifications [21]. As VHDL is the de-facto standard for specifying hardware, our current work aims at embedding this hardware description language in our verification system [24].

References

1. M.C. McFarland, A.C. Parker, and R. Camposano. Tutorial on high-level synthesis. In *25th Design Automation Conference*, pages 330–336, 1988.
2. O. Coudert, C. Berthet, and J.C. Madre. Verification of synchronous sequential machines based on symbolic execution. In *Workshop on Automatic Verification Methods for Finite State Systems*, pages 365–373, June 1989.
3. J.R. Burch, E.M. Clarke, and D. E. Long. Representing circuits more efficiently in symbolic model checking. In *28th Design Automation Conference*, pages 403–407, 1991.
4. Th. Kropf and H.-J. Wunderlich. A common approach to test generation and hardware-verification based on temporal logic. In *Proceedings of the International Test Conference*, pages 57–66, October 1991.
5. W. A. Hunt. *FM8501: A Verified Microprocessor.* PhD thesis, University of Texas, Austin, 1985.
6. M.J.C. Gordon. A proof generating system for higher-order logic. In G. Birtwistle and P.A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis.* Kluwer, 1988.

7. R. Kumar, K. Schneider, and Th. Kropf. Structuring and automating hardware proofs in a higher-order theorem-proving environment. *Journal of Formal System Design*, 2(2):165–230, 1993.
8. M.J.C. Gordon. Why higher-order logic is a good formalism for specifying and verifying hardware. In G. Milne and P.A. Subrahmanyam, editors, *Formal aspects of VLSI Design*. North-Holland, 1986.
9. J. Joyce. Formal verification and implementation of a microprocessor. In G. Birtwistle and P.A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*. Kluwer, 1988.
10. J. Joyce. More reasons why higher-order logic is a good formalism for specifying and verifying hardware. In *International Workshop on Formal Methods in VLSI Design*, Miami,1991.
11. J.D. Morison, N.E. Peeling, and T.L. Thorp. ELLA: A hardware description language. In *International Conference on Circuits and Computers*, pages 604–607, 1988.
12. R. Boulton, A. Gordon, M. Gordon, J. Herbert, and J. van Tassel. Experiences with Embedding hardware description languages in HOL. In V. Stavridou, T.F. Melham, and R. Boute, editors, *Conference on Theorem Provers in Circuit Design*, IFIP Transactions A-10, pages 129–156. North-Holland, 1992.
13. CADENCE Design Systems Inc. *CADENCE User Manuals*, July 1989.
14. E. Mayger and M. P. Fourman. Integration of formal methods with system design. In *International Conference on Very Large Scale Integration*, pages 59–69, 1991. Edinburgh.
15. F.K. Hanna and N. Daeche. Specification and verification using higher-order logic. In C.J. Koomen and T. Moto-Oka, editors, *Conference on Computer Hardware Description Languages and their Applications*, pages 418–433. North-Holland, 1985.
16. P. Camurati and P. Prinetto. Formal verification of hardware correctness: Introduction and survey of current research. *IEEE-Computer*, pages 8–19, 1988.
17. H. Eveking. *Verifikation digitaler Systeme*. Teubner Verlag, 1991.
18. J.H. Gallier. *Logic for Computer Science—Foundations of Automated Theorem Proving*. Number 5 in Computer Science and Technology Series. Harper & Row, 1986.
19. K. Schneider, R. Kumar, and Th. Kropf. The FAUST prover. In D. Kapur, editor, *11th Conference on Automated Deduction*, number 607 in Lecture Notes in Computer Science, pages 766–770. Springer Verlag, Albany, New York,1992.
20. K. Schneider, R. Kumar, and Th. Kropf. Accelerating tableaux proofs using compact representations. *Journal of Formal System Design*, 1993.
21. K. Schneider, R. Kumar, and Th. Kropf. Modelling generic hardware structures by abstract datatypes. In M. Gordon L. Claesen, editor, *International Workshop on Higher Order Logic Theorem Proving and its Applications*, pages 419–429. Elsevier Science Publishers, 1992.
22. K. Schneider, R. Kumar, and Th. Kropf. Hardware verification with first-order BDD's. In *Conference on Computer Hardware Description Languages*, 1993.
23. K. Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme. *Monatshefte für Mathematik und Physik*, 1, 1931.
24. R. Reetz and Th. Kropf. Ein formales Flußgraphenmodell zur Integration von Hardwarebeschreibungssprachen in die Hardware-Verifikation (in german). In Th. Kropf, R. Kumar, and D. Schmid, editors, *Proc. GI/ITG Workshop on Formal Methods for Correct System Design*, Technical Report 10/93. University of Karlsruhe, 1993.