Combining ACL2 and an Automated Verification Tool to Verify a Multiplier

Erik Reeber
University of Texas at Austin
Department of Computer Sciences
reeber@cs.utexas.edu

Jun Sawada
IBM Austin Research Laboratory
sawada@us.ibm.com

ABSTRACT

We have extended the ACL2 theorem prover to automatically prove properties of VHDL circuits with IBM's Internal SixthSense verification system. We have used this extension to verify a multiplier used in an industrial floating point unit. The property we ultimately verify corresponds to the correctness of the component that produces a pair of bit-vectors whose summation is equal to the product. This property is beyond the scale of the SixthSense system alone. In this paper we show how we verified the multiplier by illustrating key ACL2 lemmas and theorems, and also properties checked by SixthSense.

Categories and Subject Descriptors

B.5.2 [**Register-Transfer-level implementation**]: Design Aids—*Verification*; I.2.3 [**Artificial Intelligence**]: Deduction and Theorem Proving—*Deduction*

Keywords

Hardware verification, theorem proving, model checking

General Terms

verification, design

1. INTRODUCTION

Fully automatic methods, such as model checking and symbolic evaluation, are well suited to handling the verification of relatively small low-level hardware designs. However, automatic methods often fail to scale to large components and systems. On the other hand, theorem provers can verify large systems such as an entire processor model, although it requires a significant amount of human effort to guide the provers. We believe that these two approaches should be combined so that automated tools are used to handle the low-level hardware details, while theorem proving is used in proofs of higher-level mathematical properties that require human insights, or to combine results produced by automated verification tools.

In this paper we show how we use a combination of the ACL2 theorem prover and IBM's internal verification tool called SixthSense to verify a multiplier used in an industrial floating point unit (FPU). The strength of this combination is in our use of the SixthSense system both to hide the low-level details of the proof and to avoid embedding the complex semantics of VHDL within ACL2.

In particular we have verified that the product of two input vectors to the multiplier is equivalent to the summation of two output vectors. Due to the FPU data-flow organization, the final two output vectors are added at another portion of the FPU. This paper focus on the ACL2 proof of the multiplier, and the techniques used for it.

In Section 2, we provide background information on the SixthSense system and our integration technique with ACL2. In Section 3 we describe the multiplier circuit and provide its ACL2 specification. In Section 4 we describe the verification of the Booth encoding algorithm and in Section 5 we describe the verification of the compression algorithm. In Section 6 we provide some analysis of the human and machine efforts required to perform this verification. Finally, in Section 7 we discuss related work, before concluding in Section 8.

2. BACKGROUND

Although this paper focuses on the proof of the multiplier, we will briefly explain the ACL2 extension for the SixthSense verification tool, as our work is based on that particular setting. We assume the reader is familiar with ACL2, which is described in the book by Kaufmann, Manolios, and Moore [5] and demonstrated through the ACL2 tutorial on the web [6]. The details of this ACL2 extension for SixthSense is provided in another paper[15].

SixthSense is an IBM internal verification tool. The target hardware and the checked properties are provided as VHDL. SixthSense can prove that a property always holds after a given number of initial cycles, and present counter-examples as waveforms if it does not. SixthSense employs a transformation-based verification approach[8]. It has a number of automated verification engines that take a verification problem and either prove it, or convert it to a simpler problem and pass it to the next engine. Just a few of the engines used in SixthSense are:

- Redundancy removal engine: It identifies functionally redundant gates and removes them.

ACL2 '06 Seattle, Washington USA

- Re-timing engine: It moves logic gates beyond hardware latches, in an attempt to reduce the number of latches.
- Localization engine : An over-approximate transformation that isolates a cut point of the net-list and replaces it by primary inputs.
- Semi-formal search engine: Combines symbolic and random simulations.

SixthSense applies a dozen of these engines successively in an attempt to verify or contradict a property. One can also provide SixthSense with a configuration file that guides it to use engines in a specified order with specific parameters.

In a sense, SixthSense's approach is similar to the ACL2 theorem prover. ACL2 takes the original problem and applies rewriting, linear arithmetic, generalization, mathematical inductions and other simplifying mechanisms to the given hypothesis until it is reduced to T. Similarly, SixthSense applies a number of verification engines successively to the given verification problem. The order and types of engines applied to the verification problem is critical to the success of the verification. This problem has been addressed by using an expert-system mechanism in selecting the verification engines. It tries to intelligently call a number of verification engines with various parameters, sometimes backtracking to earlier stages and applying different series of engines. This approach has automatically verified a number of industrial-size problems.

Although SixthSense uses random simulation techniques in an attempt to disprove properties and also help formal verification techniques, it declares that a property is valid only when the property is formally verified. The soundness of the SixthSense tool itself is maintained in a similar way to the ACL2 system, that is, by fixing soundness bugs with the utmost urgency.

We built an extension of ACL2 that connects to SixthSense. As a first step, we modified the ACL2 source code to add a new `:external` hint. This modification is a prototype of an external tool integration mechanism, which we hope to be present in some future version of the ACL2 theorem prover. The `:external` hint is similar to ACL2's computed hint, as it uses a user defined function. The major difference is how the user defined functions are used. While the user defined function for the computed hint only decides the timing and the kind of ACL2 hints to be applied, the user defined functions for `:external` hint actively tries to simplify or prove the ACL2 terms.

With this extension, a user can write his own little prover as an ACL2 program-mode function and call it through the `:external` hint. The user defined function inputs a clause and returns a list of simplified clauses that imply the original, or an empty list if the input clause is proved. We can also write an ACL2 function that calls other verification tools through ACL2's `sys-call` function, and uses its result to simplify the original clause.

The `:external` hint itself is written as a small extension to the ACL2 source code, with only 57 lines of additional code. Although we use this new hint mechanism for the connectivity to SixthSense, the `:external` hint implementation itself has nothing specific to it, and can be used for the connection to other decision procedures.

We use the combination of ACL2 and SixthSense for reasoning hardware design in VHDL. Meanwhile, we specify all the properties to be checked in the ACL2 logic. The proof using SixthSense is carried out by passing such a property to an `:external` hint function named `acl2six`. First, `acl2six` translates a given ACL2 property into VHDL, and combines it with the VHDL description of hardware. Then `acl2six` calls SixthSense, which tries to prove that the translated property holds for the hardware. If SixthSense successfully proves the property, then the proven property is added as a hypothesis of the original clause and returned to ACL2, which continues to simplify it. The definition of `acl2six`, which is a pure ACL2 program-mode functions, is written as 3200-line of ACL2 logic. We also used the ACL2VHDL translator[13] for the language translation.

In ACL2 logic, signals in the VHDL model are represented by two function symbols, `sigbit` and `sigvec`, with the following type signatures:

```
(sigbit entity signame cycle phase) => bit
(sigvec entity signame lb hb cycle phase) => bv
```

These functions are constrained to output a single-bit value `bit` or bit-vector value `bv` of a signal in our hardware design, given a machine model `entity`, the signal name `signame`, integer pair `lb` and `hb` providing the lowest and highest index of a bit-vector, and natural numbers `cycle` and `phase` denoting the time. Our view is that these functions are constrained functions, and all the properties that can be proved by SixthSense are treated as implicit axioms of ACL2.

We use the simple adder as an example to illustrate the use of `acl2six` through `:external` hint. In Fig. 1, the adder is surrounded by the dashed box labeled A. This adder takes two 32-bit inputs, `a` and `b`, and produces their 32-bit summation, `sum`, when `clk` is triggered. This adder design is given as a hand-written VHDL, and its internal can be as complex as a typical carry-propagate adder.

The verification script for this adder is shown in Fig. 2. Essentially, theorem `adder-adds` states that the output of the adder is the summation of two inputs. Here, function `bv+` represents the binary addition of bit vectors. The function `adder32` defines the `entity` information as a list of the VHDL entity name, its interface, and clocking information. The input `clk` is driven by a built-in clock source called `c0`. The function `adder32` definition is "disabled" to prevent the theorem prover from expanding it. This entity definition allows us to succinctly write the `sigvec` signal expression, even for a hardware module with many interface ports and many other execution conditions.

The integer `n` specifies the cycle at which the signals are sampled. Since the adder model takes one clock cycle to compute the addition, the input is sampled at the second phase of clock cycle `(1- n)`, while the output is sample at clock cycle `n`.

To prove that the adder adds we use an `:external` hint, which calls our function `acl2six` to simplify the specified ACL2 goal. It translates the property in `adder-adds` and generates the VHDL file corresponding to the dashed box B in Fig. 1. It then calls SixthSense to prove that the signal coming out of the equality comparison always stay true after initial cycles specified by `:ignore-init-cycles`. Finally, the `acl2six` function adds the proved property as a hypothesis to the original term and returns it, which is easily reduced by ACL2 to T.

Our prototype `:external` hint can be dangerous in the sense that it may introduce unsoundness to the system. The

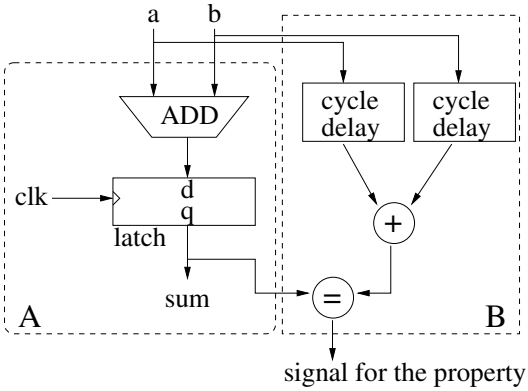

Figure 1: A 32-bit adder model

```
(defun adder32 ()
    '(adder32
      (port (clk :in std_ulogic)
            (a :in std_ulogic_vector (0 31))
            (b :in std_ulogic_vector (0 31))
            (sum :out std_ulogic_vector (0 31)))
      (extra-assigns  (clk "c0"))))

(defthm adder-adds
  (implies
   (and (integerp n) (<= 1 n))
   (equal
    (bv+ (sigvec (adder32) a (0 31) (1- n) 2)
         (sigvec (adder32) b (0 31) (1- n) 2))
    (sigvec (adder32) sum (0 31) n 2)))
  :hints
  (("Goal" :external
            (acl2six ((:cycle-expr n)
                      (:ignore-init-cycles 1))))))
```

Figure 2: The entity definition of the adder and its proof script.

current `:external` hint can call any function, even the function that "proves" nil. In order to mitigate this danger, future external tool mechanisms should only allow the use of `:external` functions that the user has declared to be trusted. Also unsoundness can result from functional instantiation of `sigbit` or `sigvec`. When new theorems are proven by `acl2six`, it is equivalent to introducing new constraints to encapsulated function `sigbit` or `sigvec`. However, the functional instantiation does not check that the instantiating function satisfies these theorems introduced by `acl2six`. One way to remove this unsoundness is to disallow the instantiation of functions that are associated with an `:external` function, such as `sigbit` and `sigvec`.

3. MULTIPLIER

We worked on a multiplier used in a double-precision FPU design. A double-precision floating-point number has a 52 bit mantissa, not including the implicit leading bit [2]. Including the leading bit, we need at least 53x53 multiplier for a double-precision FPU.

A simplified block diagram of the multiplier is given in Figure 3. This multiplier is named `da_fp_mult`. It takes two operands A and C, and, after three and half cycles later, produces two bit-vectors Sum and $Carry$, such that $Sum + Carry = A \times C$.

The multiplier first performs Booth-encoding. It produces 27 vectors representing either $-2 \times C$, $-1 \times C$, 0, $1 \times C$ or $2 \times C$, by looking at the three consecutive bits of A at 27 different places. When these 27 vectors are shifted by 2-bit increments and added together, it equals to the product $A \times C$. These additions are performed by the sequence of carry-save-adder stages 1 through 5. A typical 3-to-2 carry-save-adder takes three inputs I_0, I_1 and I_2 and produces output O_0 and O_1 such that $I_0 + I_1 + I_2 = O_0 + O_1$. Stage1 consists of such 3-to-2 carry-save-adders and reduces 27 vectors to 18 while preserving the total sum. Similarly, the subsequent 3-to-2 and 4-to-2 carry-save-adder stages reduce the number of bit vectors to 12, 6, 4, and to the final 2 vectors.

A complete binary multiplier adds the final two bit vectors using a carry-propagate adder or similar scheme to get the final product. However, in a typical FPU design, the final addition is performed separately from the multiplier to increase its performance, and it is not part of our design, either.

The correctness of this design is encoded by the following ACL2 theorem:

```
(defthm multiplier-correct
  (implies
   (and (integerp n)
        (<= 7 n))
   (equal (bv+ (Sum-output n 1)
               (Carry-output n 1))
          (bv (* (bv-val (A-input (- n 4) 2))
                 (bv-val (C-input (- n 4) 2)))
              108))))
```

Essentially, this theorem states that the product of `A-input` and `C-input` is equal to the addition of `Sum-output` and `Carry-output`. In the VHDL implementation of the multiplier, the correct output starts to stream out after 7 cycles of initialization and filling the pipeline, thus adding the condition `(<= 7 n)`. As we have seen in the previous section, `bv+` returns the bit vector representing the sum of two arguments. Function `bv-val` returns the value represented by a bit-vector, and `(bv v l)` returns a bit vector of length `l` representing value `v`.

`A-input`, `C-input`, `Sum-output`, and `Carry-output` designate the corresponding VHDL input and output signals of the multiplier at a given cycle and phase. It hides some of the hardware implementation details. For example, the actual hardware outputs are logically negated due to design constraints. `A-input` and `C-input` also include the implicit leading bit at the most significant position and extra constant bits at the least significant position which do not exist in real hardware.

Multiplier designs are particularly difficult to verify with an OBDD-based symbolic simulation or a SAT-solver. Our experiment shows that it is hard to verify even that stage 1 preserves the summation of its 27 input and 18 output vectors. This can be easily understood if we consider the OBDD representation of the most significant bit of the sum of 27 vectors. We think this example shows the strength of the combined use of theorem prover and automated verification tools.

The overall proof strategy is to use the theorem prover to reduce the final theorem into properties that can be proven automatically with our integration of ACL2 and SixthSense.

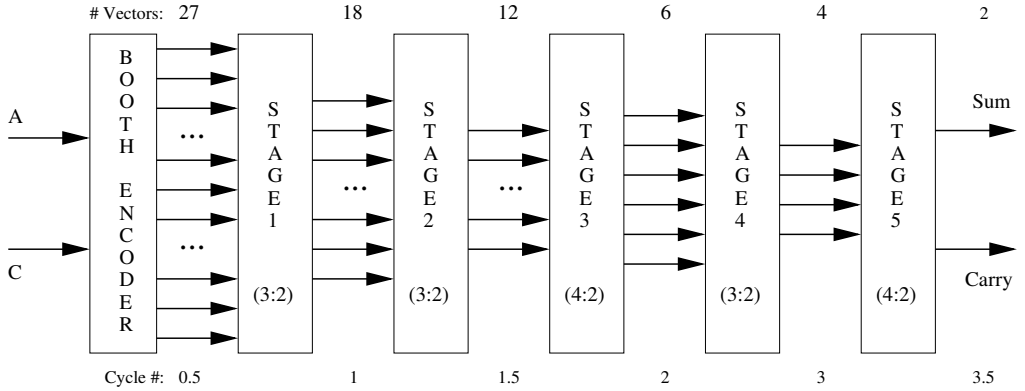

Figure 3: An overview of the `da_fp_mult` design.

First the final theorem is reduced to two major lemmas, the correctness of the Booth encoder and the correctness of the subsequent compressions stages. The Booth encoder lemma states that the addition of all the bit-vectors coming out of the Booth encoder is equal to the product of A and C. The second lemma states that the summation is preserved during the compression stages. Combining these two major lemmas leads to the proof of `multiplier-correct`.

4. VERIFICATION OF THE BOOTH ENCODING

We verify the Booth encoding by creating three ACL2 models of a Booth encoder, which we call the high-level model, the low-level model, and the BV model. We first verify the high-level model, then prove the equivalence of the high-level and the low-level model, then prove the equivalence of the low-level and the BV model, and finally prove the equivalence of the BV model and the actual hardware design.

We start this verification process by defining the high-level Booth encoder in ACL2:

```
(defun acl2-booth-vector (x y)
  (cond
   ((equal x 3)
    ;; x = 3
    (* 2 y))
   ((or (equal x 2) (equal x 1))
    ;; x = 2 or x = 1
    y)
   ((or (equal x 0) (equal x 7))
    ;; x = 0 or x = -1 mod 8
    0)
   ((or (equal x 6) (equal x 5))
    ;; x = -2 mod 8 or x = -3 mod 8
    (- y))
   (t
    ;; x = -4 mod 8
    (* -2 y))))
(defun acl2-booth-mult1 (x y)
  (if (zp x)
      0
    (+ (acl2-booth-vector (mod x 8) y)
       (* 4 (acl2-booth-mult1 (floor x 4) y)))))
(defun acl2-booth-mult (x y)
  (acl2-booth-mult1 (* 2 x) y))
```

The function `acl2-booth-mult` specifies the Booth encoding algorithm. It multiplies two numbers x and y by summing Booth-encoding vectors, each of which consist of either $2 \times y$, y, 0, $-y$, or $-2 \times y$.

These functions can be easily defined with the help of ACL2 arithmetic books. The reader who is are not familiar with Booth-encoding algorithm may play with `acl2-booth-mult`. For example, `(acl2-booth-mult 7 3)` multiplies 7 by 3, by calling `(acl2-booth-mult1 14 3)`. This leads to the calculation of Booth vectors `(acl2-booth-vector 6 3)` $= -3$, and `(acl2-booth-vector 3 3)` $= 6$. Finally these values are added to $-3 + 4 \times 6 = 21$, a correct product of 7 and 3.

The correctness of the Booth encoding algorithm specified by `acl2-booth-mult` is stated as the following theorem:

```
(defthm acl2-booth-mult-multiplies
  (implies (and (natp x)
                (integerp y))
           (equal (acl2-booth-mult x y)
                  (* x y))))
```

With the help of the arithmetic-2 book [6], its proof is short and straightforward. We used the following generalized lemma to help the proof:

```
(defthm acl2-booth-mult1-multiplies
  (implies
   (and (natp x)
        (integerp y))
   (equal (acl2-booth-mult1 x y)
          (* (floor (1+ x) 2) y))))
```

This lemma is then proved by the induction scheme suggested by `acl2-booth-mult1`. Some helper lemmas regarding `floor`, `mod`, and `integerp` are also required.

We next define our low-level Booth encoder model as follows:

```
(defun acl2-lbitn (n x)
  (if (not (and (integerp n) (< 0 n)))
      (not (equal (mod x 2) 0))
    (acl2-lbitn (1- n) (floor x 2))))
```

```
(defun acl2-ll-booth-vector (nm x y)
  (let* ((x (* 2 x))
         (bv0 (acl2-lbitn (* 2 nm) x))
         (bv1 (acl2-lbitn (+ (* 2 nm) -1) x))
         (bv2 (acl2-lbitn (+ (* 2 nm) -2) x)))
    (cond
     ((and (not bv0) bv1 bv2)
      ;; x = 3
      (* y (expt 2 (+ -1 (* 2 nm)))))
     ((and (not bv0) (or bv1 bv2))
      ;; x = 2 or x = 1
      (* y (expt 2 (+ -2 (* 2 nm)))))
     ((or (not bv0) (and bv1 bv2))
      ;; x = 0 or x = -1 mod 8
      0)
     ((or bv1 bv2)
      ;; x = -2 mod 8 or x = -3 mod 8
      (- (* y (expt 2 (+ -2 (* 2 nm))))))
     (t
      ;; x = -4 mod 8
      (- (* y (expt 2 (+ -1 (* 2 nm))))))))))
(defun acl2-ll-booth-mult (nm x-size x y)
  (if (or (not (integerp nm))
          (< (floor (+ 2 x-size) 2) nm))
      0
    (+ (acl2-ll-booth-vector nm x y)
       (acl2-ll-booth-mult (1+ nm) x-size x y))))
```

The main difference between the low-level Booth encoder and the high-level one is that the high-level encoder shifts `x` on each iteration of `acl2-booth-mult1`, whereas the low-level encoder passes `x` unmodified to the Booth vector function, `acl2-ll-booth-vector`. Instead of using the least significant three bits of `x`, `acl2-ll-booth-vector` uses an accessor function `acl2-lbitn` to obtain the appropriate three bits for Booth selection. The following is the key lemma used to prove the equivalence of the low-level and high-level Booth encoders:

```
(defthm acl2-ll-booth-vector-equiv
  (implies
   (and (integerp n)
        (natp x)
        (natp y)
        (<= 1 n))
   (equal (acl2-ll-booth-vector n x y)
          (* (expt 4 (1- n))
             (acl2-booth-vector
              (mod (floor (* 2 x)
                          (expt 4 (1- n)))
                   8)
              y)))))
```

where `(expt 4 (1- n))` produces 4^{n-1}. The `mod` term corresponds to the three bits used to select Booth vector. The entire lemma states that the high-level `acl2-booth-vector` and low-level `acl2-ll-booth-vector` are the same vector after shifting appropriately.

Our final Booth encoding model, which we call the BV model, uses the following definition to define Booth vectors:

```
(defun booth-vector (total-n n x x-size y y-size)
 (let* ((x (bv&& x (pad0 1)))
        (x-size (1+ x-size))
        (b0 (lbitn (* 2 (- total-n n)) x x-size))
        (b1 (lbitn (+ (* 2 (- total-n n)) -1)
                   x x-size))
        (b2 (lbitn (+ (* 2 (- total-n n)) -2)
                   x x-size)))
  (bv-cond
   ((b-and (b-not b0) (b-and b1 b2)) ;case 3
    (bv&& (pad0 (+ 1 (* 2 n)))
          y
          (pad0 (+ -1 (* 2 (- total-n n))))))
   ((b-and (b-not b0) (b-ior b1 b2)) ;case 1 or 2
    (bv&& (pad0 (+ 2 (* 2 n)))
          y
          (pad0 (+ -2 (* 2 (- total-n n))))))
   ((b-ior (b-not b0) (b-and b1 b2)) ;case 0 or 7
    (pad0 (+ (* 2 total-n) y-size)))
   ((b-ior b1 b2)                    ;case 5 or 6
    (bv-neg
     (bv&& (pad0 (+ 2 (* 2 n)))
           y
           (pad0 (+ -2 (* 2 (- total-n n)))))))
   (*b1*                             ;case 4
    (bv-neg
     (bv&& (pad0 (+ 1 (* 2 n)))
           y
           (pad0 (+ -1 (* 2 (- total-n n))))))))))
```

where `lbitn` is defined as:

```
(defun lbitn (n x x-size)
  (bitn (- (1- x-size) n) x)).
```

The `booth-vector` function has a similar structure as the function `acl2-ll-booth-vector`, but is considerably more verbose. The input `total-n` represents the total number of Booth vectors in our encoding, 27 for our system; the input `x-size` represents the size of the bit vector `x`, which is 54 for our system; and the input `y-size` represents the size of `y`, which is 53 for our system.

The main difference between the low level model and the BV model is that the low-level model `acl2-ll-booth-vector` uses Boolean, arithmetic, and bit vector operations, while BV model `booth-vector` are purely constructed of bit-vector operations. These bit vector operators include the function `bv-cond`, which is similar to `cond` but uses a bit rather than a Boolean for each condition; `bv&&`, which concatenates bit vectors; `bv-neg`, which negates a bit vector; `*b1*`, which is a bit constant 1; `b-not`, `b-ior` and `b-and`, which perform bit logical operations; and `pad0`, which produces a bit vector of zeros. It is necessary for these properties to contain only bit-vector operations so that they can be translated to VHDL and input to SixthSense.

Yet another difference between the two models is that the low-level model uses indexing starting from the right end of the bit vector, while the BV model indexes from the left. Furthermore, the BV model is only designed to model Booth encodings where the size of `x` is an even number.

The following is the key lemma relating the low-level Booth vector model, `acl2-ll-booth-vector`, to the BV model, `booth-vector`:

```
(defthm bv-booth-ll-vector-equiv
  (implies
   (and (equal (mod x-size 2) 0)
        (natp n)
        (integerp total-n)
        (<= 1 total-n)
        (< n total-n)
        (bvp x)
        (bvp y)
        (equal (* 2 total-n) (bv-size x))
        (< 0 (bv-size x))
        (< 0 (bv-size y))
        (not (neg-bvp x))
        (equal x-size (bv-size x))
        (equal y-size (bv-size y)))
   (equal
    (bv-val (booth-vector total-n
                          n
                          x x-size
                          y y-size))
    (mod (acl2-ll-booth-vector (- total-n n)
                               (bv-val x)
                               (bv-val y))
         (expt 2 (+ (* 2 total-n) y-size))))))
```

Here, `bv-val` is used to convert a bit vector into a natural number, and `neg-bvp` is used to determine if the most significant bit of a bit-vector is high. This lemma shows that booth vector defined in the low-level model as integer is equivalent to the BV-model's Booth vector defined in terms of bit vectors.

Since the BV model can be compiled into VHDL we can use SixthSense to compare the BV model to the actual VHDL design. To do this we verify 27 equivalence theorems, each relating the signals representing the Booth encoding in the design to the BV model's `booth-vector` function. Most of these theorems follow a repetitive structure that can be generated with a macro. For example, our macro produces the following theorem relating the 24th Booth vectors:

```
(defthm booth-row-24
  (implies
   (and (integerp n)
        (<= 3 n))
   (equal
    (vhdl-booth-vector 24 n 1)
    (bv+all
     (booth-vector-imp-24
      (A-input (1- n) 2)
      (C-input (1- n) 2))
     (bv&& (pad0 47)
           (pad1 2)
           (pad0 59))
     (bv&& (pad0 105)
           (b2bv (vhdl-booth-sign 25 n 1))
           (pad0 2))
     (bv-neg
      (bv&& (pad0 103)
            (b2bv (vhdl-booth-sign 24 n 1))
            (pad0 4))))))
  :hints
  (('`Goal'`
      :external
      (acl2six
       ((:cycle-expr n)
        (:ignore-init-cycles 3)
        (:config_file "equiv_check.config"))))))
```

Here `vhdl-booth-vector` and `vhdl-booth-sign` use `sigbit` and `sigvec` to obtain the value corresponding to the given Booth vector in the design and its sign bit. The macro `(booth-vector-imp-24 x y)` expands into a term equal to `(booth-vector 27 24 x 54 y 53)`.

The theorem is verified by translating it into VHDL and running SixthSense. The `booth-row-24` theorem states that the 24th Booth vector in hardware is equal to the 24th Booth vector of the BV model, plus 2^{59}, plus the 25th Booth vector's sign bit, and minus the 24th Booth vector's sign bit.

The addition of the sign bits and constants are an optimization implemented in the hardware that minimizes the size of the design. When all 27 Booth vectors are added together these sign bits and constants cancel to zero. This cancellation is verified using the ACL2 theorem prover, by referring to theorems such as the following:

```
(defthmd reorder-bv+-3
  (implies
   (and (syntaxp (not (lower-booth-signals x y)))
        (bvp x)
        (bvp y)
        (bvp z)
        (equal (bv-size x) (bv-size y)))
   (equal (bv+ x (bv+ y z))
          (bv+ y (bv+ x z)))))
```

This theorem relies on the commutativity of `bv+` to reorder elements in a summation. Here `lower-booth-signals` returns true if `x` is smaller than `y` by a syntactic measure. By forcing elements with this measure to be close together the sign bits are canceled with their negations and constants are combined, eventually adding to zero. The resulting sum is therefore equal to the BV model, which is in turn equal to the multiplication of the inputs.

Verifying the Booth encoding requires a significant amount of theorem proving effort. The great majority of this effort, however, simply involved producing a verified Booth encoding that can be compiled to VHDL. This VHDL then be-

comes a verification artifact used by SixthSense to compare against the actual implementation. By using SixthSense we are able to verify the actual Booth encoding implementation while reasoning only about the simpler, less efficient BV model. We therefore avoid reasoning about the internals of the implementation. Any change to the design that does not affect the value of the Booth vectors will not affect the proof, since we can reuse the BV model.

5. VERIFYING THE 5 STAGE COMPRESSION ALGORITHM

We next verify the 5 stage compression algorithm, which reduces 27 bit-vectors to 2 bit-vectors while preserving their sums. This theorem can be easily expressed as an ACL2 theorem that can be translated into VHDL. However, the verification of even the stage 1 in Figure 3, which is implemented by nine 3:2 compressor carry-save-adders in parallel, is too much for SixthSense to handle all at once. Instead we verify each individual compressor carry-save-adder separately. For example, the following theorem verifies the correctness of one of the Stage 1 compressors:

```
(defthm stage1-row8-help
  (implies
   (and (integerp n)
        (<= 4 n))
   (equal
    (bv+all (stage1-sum8 n 2)
            (stage1-car8 n 2))
    (bv+all (vhdl-booth-vector 26 n 1)
            (vhdl-booth-vector 25 n 1)
            (vhdl-booth-vector 24 n 1))))
  :hints
  (('‘Goal’’
      :external
      (acl2six
       ((:cycle-expr n)
        (:ignore-init-cycles 4)
        (:config_file "equiv_check.config"))))))
```

Here the macros `stage1-sum8` and `stage1-car8` use `sigvec` to produce the values of the VHDL signals input into the eighth compressor. This theorem is proved directly with SixthSense, with the assistance of the same configuration file used previously.

To compose the nine theorems about stage 1 carry-save-adders, we first convert the above theorem to a rewriting rule of different form, just like:

```
(defthm stage1-row8
  (implies
   (and (integerp n)
        (<= 4 n))
   (equal
    (stage1-sum8 n 2)
    (bv+all (vhdl-booth-vector 26 n 1)
            (vhdl-booth-vector 25 n 1)
            (vhdl-booth-vector 24 n 1)
            (bv-neg (stage1-car8 n 2))))))
```

Here we have simply moved the `stage1-car8` term to the right-hand side of the equality and negated it. However, this forms of rewriting rule helps the proof of the entire stage 1 compressor.

The proof rewrites the summation of `stage1-sum`n terms and `stage1-car`n terms for all $0 \leq n \leq 8$ into the sum of 27 Booth vectors. The rewriting rule `stage1-row`m replaces all the `stage1-sum`n terms to `vhdl-booth-vector` terms and the negation of `stage1-car`n terms. After applying associativity and commutativity rules of `bv+`, `stage1-car`n terms and its negations cancel out, and the summation of 27 `vhdl-booth-vector` terms remain, which verifies the equivalence of the summation.

The same technique is used to compose the stage2, stage 3, stage 4, and stage 5 compressor theorems. We therefore obtain a proof that the summation of the original 27 inputs to stage 1 is equivalent to the summation of the two outputs of stage 5.

The verification of the compression algorithm is left almost entirely to SixthSense. The theorem prover is only used to compose the theorems about the compressor subunits and to compose the resulting stage input and output equivalence theorems. This composition relies merely on lemmas involving `bv+` and `bv-neg`. We have avoided the internals of the VHDL module almost entirely; only signals from the top-level multiplier module are visible within the theorem prover.

6. ANALYSIS

We ultimately verify the ACL2 theorem in Section 3, which states that the product of the inputs to the multiplier is equal to the sum of its outputs after 7 cycles. This theorem is proven by reducing it into a series of properties that can be verified by our integration of ACL2 and SixthSense. Each of these properties is translated into VHDL properties involving the final implementation, which SixthSense formally verifies are valid.

The entire verification effort required about a month, 21 eight hour work days, of human effort from a single experienced ACL2 user. About one third of this time was spent finding properties that could be verified by SixthSense and writing the necessary configuration files. This time likely would decrease significantly on future efforts due to increased experience with the SixthSense system. The remaining two third of the time was spent developing the necessary ACL2 proof. No bugs in the design were discovered, but we were able to greatly increase the assurance that the design is correct. Furthermore, the proof, which requires about 50 minutes to run, should become a valuable tool for finding bugs in any future modifications to the design.

We found that the proof of the multiplier compressor went smoothly, by combining the power of SixthSense and the clever application of ACL2 rewriting rules. However, the proof of the Booth encoder was more time consuming. This was due to the large difference between the high-level Booth multiplier and the Booth multiplier that operates on bit vectors. It is a part of future work to improve this part of the proof.

7. RELATED WORK

There have been many integrations between model-checking and theorem proving tools. Most notably, the PVS theorem prover was built with model checking as a primitive proof engine [9]. The SyMP model prover uses a more general approach to integrating the two techniques [1]. Within the ACL2 theorem prover, numerous model-checking inspired engines such as UCLID [7] have been integrated. What makes our system unique is the fact that we are verifying, in a scalable manner, industrial RTL-level designs written

in an HDL.

Within the ACL2 theorem proving community there are three systems of which we know that verify RTL-level designs in HDL: AMD's system that translates Verilog to ACL2 [11], Hunt and Reeber's system that translates Verilog to DE2 [4], and Borrione's system that translates VHDL to ACL2 [10]. All of these use the embedding of HDL in ACL2 logic, making our approach unique. And of these only AMD's system captures a broad enough range of the RTL to capture many industrial designs.

Outside the ACL2 theorem proving community, the only system of which we know that verifies designs within a broad range of industrial RTL is at Intel. Intel's system, FORTE, uses a custom theorem prover, based on HOL, built on top of a high-performance symbolic trajectory evaluator (STE) [3]. Our work may be similar, but we were able to avoid building a custom theorem prover by creating the general-purpose extension to ACL2. Also our example here shows the relatively heavy use of theorem prover for the proof of arithmetic components.

8. CONCLUSION

We have verified a multiplier used in an industrial FPU, implemented in VHDL. In particular we verified a Booth encoding implementation and a design that compresses the summation of the resulting 27 vectors into a summation of two vectors. By using SixthSense we were able to hide the low-level details of the design from the theorem prover and achieve a greater degree of automation than previously present.

Our use of the `:external` hint mechanism shows the advantages of a general-purpose link between ACL2 and external tools. Building such an external tool link within the standard ACL2 system will enable ACL2 to be extended with SixthSense, as well as other external tools. We do not have to modify the source code of ACL2 for every individual extension. We only need to write an ACL2 program-mode function to interface with each tool.

We believe the combination of ACL2 and SixthSense will scale well to much larger designs. For example, the verification of the entire FPU, relative to the IEEE-754 standard, seems reasonable within our system. The divide and square root verification [14, 12], which require rich arithmetic analysis, may also be a good target for our system.

8.1 Acknowledgments

We would like to acknowledge Sandip Ray, for building the initial prototype of the `acl2six` system; Matt Kaufmann, for helping design the `:external` extension of ACL2; and Jason R. Baumgartner, Hari Mony, and Viresh Paruthi, for their help with the SixthSense tool.

9. REFERENCES

[1] S. Berezin. *Model Checking and Theorem Proving: A Unified Framework*. PhD thesis, Carnegie Mellon University, 2002.

[2] Institute of Electrical and Electronic Engineers. *IEEE Standard for Binary Floating-Point Arithmetic*. ANSI/IEEE Std 754-1985.

[3] R. B. Jones, J. W. O'Leary, C.-J. H. Seger, M. Aagaard, and T. F. Melham. Practical Formal Verification in Microprocessor Design. *IEEE Design & Test of Computers*, 18(4):16–25, 2001.

[4] W. A. H. Jr. and E. Reeber. Formalization of the DE2 Language. In *CHARME*, pages 20–34, 2005.

[5] M. Kaufmann, P. Manolios, and J. S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Press, 2000.

[6] M. Kaufmann and J. S. Moore. *ACL2: A Computational Logic for Applicative Common Lisp*. URL:http://www.cs.utexas.edu/users/moore/acl2/-acl2-doc.html.

[7] P. Manolios and S. K. Srinivasan. Automatic Verification of Safety and Liveness for XScale-Like Processor Models Using WEB Refinements. In *DATE*, pages 168–175, 2004.

[8] H. Mony, J. Baumgartner, V. Paruthi, R. Kanzelman, and A. Kuehlmann. Scalable Automated Verification via Expert-System Guided Transformations. In A. J. Hu and A. K. Martin, editors, *Formal Methods in Computer-Aided Design: Third International Conference FMCAD 2004*, volume 3312 of *Lecture Notes in Computer Science*, pages 217–233. Springer-Verlag, 2000.

[9] S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. K. Srivas. PVS: Combining Specification, Proof Checking, and Model Checking. In *CAV*, pages 411–414, 1996.

[10] V. M. Rodrigues, D. Borrione, and P. Georgelin. Using the ACL2 Theorem Prover to Reason about VHDL Components. *RITA*, 7(1):129–148, 2000.

[11] D. Russinoff. A Mechanically Checked Proof of IEEE Compliance of a Register-Transfer-Level Specification of the AMD K7 Floating Point Multiplication, Division and Square Root Instructions. *LMS Journal of Computation and Mathematics*, 1:148–200, 1998.

[12] D. Russinoff. A Mechanically Checked Proof of IEEE Compliance of a Register-Transfer-Level Specification of the AMD-K7 Floating-Point Multiplication, Division, and Square Root Instructions. *London Mathematical Society Journal of Computation and Mathematics*, 1:148–200, December 1998.

[13] J. Sawada. ACL2VHDL Translator: A Simple Approach to Fill the Semantic Gap. In *Proceedings of the Fifth International Workshop of the ACL2 Theorem Prover and its Applications (ACL2-2004)*, 2004.

[14] J. Sawada and R. Gamboa. Mechanical Verification of a Square Root Algorithm Using Taylor's Theorem. In *Formal Methods in Computer Aided Design (FMCAD '02)*, volume 2517 of *LNCS*, pages 274–291. Springer Verlag, 2002.

[15] J. Sawada and E. Reeber. ACL2SIX : A Hint used to Integrate a Theorem Prover and an Automated Verification Tool. In *To appear in FMCAD 2006*.