Hierarchical Verification of Two-Dimensional High-Speed Multiplication in PVS: A Case Study *

Harald Rueß

Universität Ulm, Fakultät für Informatik
D-89069 Ulm, Germany
ruess@informatik.uni-ulm.de

Abstract

It is shown how to use the PVS specification language and proof checker to present a hierarchical formalization of a two-dimensional, high-speed integer multiplier on the gate level. We first give an informal description of iterative array multiplier circuits together with a natural refinement into vertical and horizontal stages, and then show how the various features of PVS can be used to obtain a readable, high-level specification. The verification exploits the tight integration between rewriting, arithmetic decision procedures, and equality that is present in PVS. Altogether, this case study demonstrates that the resources of an expressive specification language and of a general-purpose theorem prover permit highly automated verification in this domain, and can contribute to clarity, generality, and reuse.

1 Introduction

Verifying functional correctness results about arithmetic circuits poses some serious challenges to current hardware verification techniques. Almost all automated approaches to hardware-verification, such as model-checking, rely on *binary decision diagrams* (BDDs) or some variant of these as the underlying representation form. BDD representations of arithmetic circuits such as multiplication or division, however, have been shown to grow exponentially with the word size [Bry94]. Consequently, it is not feasible to directly apply BDD-based methods to prove the overall correctness of many interesting arithmetic

*To appear in *Formal Methods in Computer Aided Design* (FMCAD '96), M. Srivas (Ed.), Palo Alto, November 1996.

circuits with wide data paths. Moreover, such methods suffer from a lack of expressiveness for describing high-level specifications. Theorem-proving based approaches, on the other hand, exploit the regular structure of arithmetic circuits and use inductive reasoning to establish correctness results for arbitrary word sizes. A major complaint against the use of mechanical theorem provers for hardware verification, however, has been that they often require significant human guidance.

The overall aim in this paper is to explore the use of techniques based on high-level specification and highly automated formal proof construction in PVS to systematically verify the functional correctness of arithmetic circuits with fixed, but arbitrary sized operands. The paper is based upon a case study: a specification for a two-dimensional, high-speed integer multiplier, its realization on the gate-level, and its formal verification. The basic strategy for constructing this correctness proof relies on a natural decomposition of the two-dimensional multiplier circuit into vertical and horizontal stages.

This paper is organized as follows: Section 2 discusses the main features of the PVS specification language and the proof-checker. Section 3 informally introduces (unsigned) *iterative array multiplication* and informally describes the decomposition of the verification problem into a hierarchy of verification tasks. The steps in the hierarchy are independent from each other, but the description of these verification steps proceeds strictly bottom-up. Gate-level implementations of low-level building blocks such as *carry-propagate* and *carry-save* addition rows and their automatic verification are developed in Section 4, and the formalization of the vertical and horizontal stages of the *iterative array multiplier* are described in Section 5. It abstracts this multiplier in terms of a series of carry-save addition steps followed by a final carry-propagate step. Next, it is formally verified in Section 6 that the overall functional correctness of the iterative array multiplier follows from the functional correctness of the lower-level building blocks. These higher-level proof steps are independent of details of the gate-level implementations of the lower-level building blocks. The PVS language support for formalizing *generic hardware components* is discussed in Section 7. Finally, Sections 8 and 9 end with a comparison with related work and some conclusions.

Altogether, this case study demonstrates that the resources of an expressive specification language and a general-purpose theorem prover permit highly-automated verification in the domain of arithmetic circuits, and can contribute to clarity, generality, and reuse.

2 An Overview of PVS

The PVS system combines an expressive specification language with a productive, interactive proof checker that has a reasonable amount of theorem proving capabilities, and has been used for reasoning in domains as diverse as microprocessor verification, protocol verification, and algorithm and architectures concerning fault-tolerance [ORSvH95].

The PVS specification language builds on classical typed higher-order logic with the usual base types `bool`, `nat`, `rational`, `real`, ... and the function type constructor `[A -> B]`. The type system of PVS is augmented with *dependent types* and *abstract data types*. A distinctive feature of the PVS specification language are *predicate subtypes* {`x:A | P(x)`}. These subtypes consist of exactly those elements `a` of type `A` satisfying predicate `P(a)`. Predicate subtypes are used to explicitly constrain the domain and ranges of operations in a specification and to define partial functions. Predicate subtypes are used, for example, to define the integer ranges `below[n]` ($[0, \ldots, n)$) or `subrange[-m, n]` ($[-m, \ldots, n]$). In general, type-checking with predicate subtypes is undecidable, and the type-checker generates *type correctness conditions* (TCCs) corresponding to predicate subtypes. A large number of TCCs are discharged by specialized proof strategies, and a PVS expression is not considered to be fully type-checked unless all generated TCCs have been proven correct.

PVS specifications are packaged as *theories* that can be parametric in types and constants. A built-in *prelude* and loadable *libraries* provide standard specifications and proved facts for a large number of theories; for example, fixed-sized `bit-vectors`.

An `N`-bit bit-vector is represented as an array `bvec[N]`, i.e. a function from the type `below[N]` of natural numbers less than `N` to `bit`. The parameter `N` is constrained to be a positive natural number `posnat` since bit-vectors of length `0` are not permitted. Bits are represented as elements of type `upto[1]` which consists of the numbers `0` and `1`.

The function `bv2nat` (see 1) from the bit-vector library defines the unsigned interpretation of bit-vectors of length `N` as natural numbers.

1

```
bv_nat[N: posnat]: THEORY
 BEGIN
  IMPORTING bitvectors@bv[N]

  bv2nat_rec(n: upto[N], bv:bvec[N]): RECURSIVE nat =
   IF n = 0 THEN 0 ELSE exp2(n-1) * bv(n-1) + bv2nat_rec(n-1,bv) ENDIF
  MEASURE n

  bv2nat(bv:bvec[N]): below[exp2(N)] = bv2nat_rec(N, bv)
  CONVERSION bv2nat
 END bv_nat
```

It is defined in the theory `bv_nat` that is parameterized with respect to the length `N` of bit-vectors. `bv2nat_rec` is an auxiliary function used only to define `bv2nat`, and interprets the lower portion of a bit vector as a natural number. Recursive function definitions in PVS must have an associated `MEASURE` function to ensure termination. The type-checker automatically generates type correctness proof obligations to show that the argument to every recursive call is decreasing according to the given measure. Moreover, the `CONVERSION` directive causes the type-checker to implicitly inject the conversion `bv2nat` whenever it finds a bit-vector instead of the required number. In this way it is possible to *overload* bit-vectors `xv` with their unsigned interpretations `bv2nat(xv)`. This kind of overloading is commonly used in computer arithmetic textbooks to express concise and readable circuit descriptions.

Many more basic facts about bit-vectors are predefined in the bit-vector library of PVS. Concatenation of bit-vectors `xv` and `yv`, for example, is denoted by `xv o yv`, and extraction of `(i+j+1)` many bits `i` through `j` from bit-vector `xv` of type `bvec[N]` is denoted by `xv^(i,j)`; `xv(i)` is shorthand for `xv^(i,i)`. Obviously, extraction is restricted, using predicate subtyping, to the cases `(i < N)` and `(i >= j)`.

Proofs in PVS are presented in a sequent calculus. The atomic commands of the PVS prover component include induction, quantifier instantiation, automatic conditional rewriting, simplification using arithmetic and equality decision procedures and type information, and propositional simplification using binary decision diagrams. The `skolem*` command, for example, repeatedly introduces Skolem constants for universal-strength quantifiers, and `assert` combines rewriting with decision procedures.

Finally, PVS has an LCF-like strategy language for combining inference steps into more complicated proof strategies. The strategy `then`, for example, sequentially applies a sequence of sub-strategies, and the defined rule `grind` combines rewriting with propositional simplification using BDDs and decision procedures. This strategy, in combination with case analysis, has been very effective for proving many typical hardware theorems.

3 Iterative Array Multipliers

Many multiplier circuits are based on the grade school principle of multiplying any two given `N` bit numbers: computing the partial products, and adding the partial products to obtain the required result. The two basic operations of multiplication, the generation of partial products and their summation, may be merged. This merging speeds up the multiplication, since the overhead that is due to the separate controls is avoided. Such multipliers, which consist of identical cells, each capable of forming a new partial product and adding it to the previously accumulated partial product, are called *iterative array multipliers* [Kor93].

To illustrate the operation of an iterative array multiplier for unsigned numbers, examine the 4×5 parallelogram shown in Figure 1. This circuit multiplies two `N = 5` wide operands, say `av` and `xv`. It adds the first two partial products (i.e. `av * xv(0)` and `av * xv(1)`) in row one after proper alignment. The

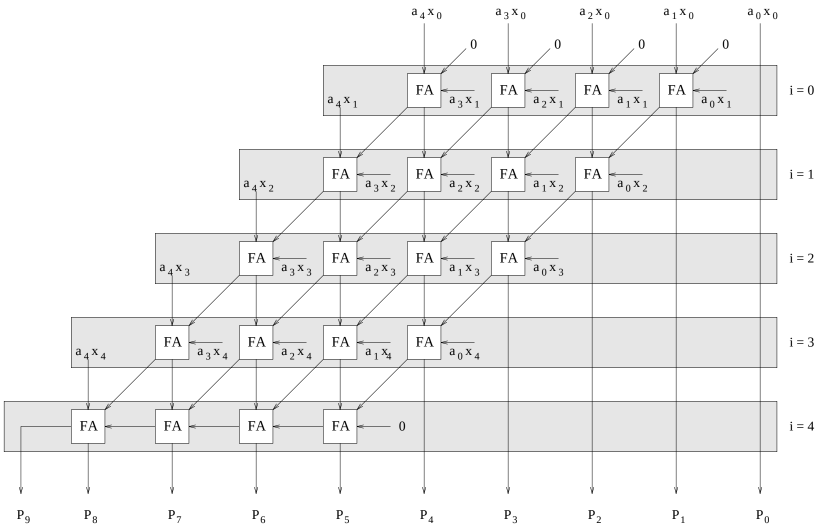

Figure 1: Iterative Array Multipliers

results of the first row are then added to `av * xv(2)` in the second row, and so on. The basic cell for such an array multiplier is a full adder `FA` accepting one bit of the new partial product, and a carry-in bit. In the first four rows there is no horizontal carry propagation. In other words, a *carry-save* type addition is performed in these rows, and the accumulated partial product consists of intermediate sum and carry bits. A horizontal carry propagation is allowed only in the last row. The last row of cells in this figure is a *ripple-carry adder*. Consequently, an iterative array multiplier circuit with operands of size `N` is abstracted in terms of a series of `(N - 1)` carry-save steps followed by a carry-propagate step. Block diagrams of these stages of the *vertical decomposition* or vertical refinement are depicted in Figure 2 together with word-level equations between appropriately shifted input and output vectors. The word-level condition `sv' =` 2^N `* cv + sv` of the carry-propagate step, for example, states that the (unsigned) interpretation of the output sum vector `sv'` equals the sum of the input sum vector `sv` and the carry-in vector `cv` shifted by `N` bits.

Moreover, looking at the circuit in Figure 1, every vertical stage can naturally be decomposed into different functional units. The *horizontal decompositions* of `csa_step(i)` and `cpa_step` are depicted in Figure 3. For every vertical stage, the decomposition consists of three functional units. The left-most unit simply computes the upmost bit of the next partial sum `av(N - 1) * xv(i + 1)`, the

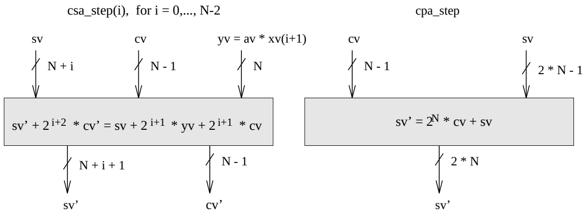

Figure 2: Vertical Refinement

right-most unit shifts through the lower `(i + 1)` bits of the current partial sum `sv`, and the central unit consists of a carry-save adder row of length `(N - 1)`. The final step of the multiplier is decomposed in a similar way (see Figure 3) with the central unit being a `(N - 1)`-bit carry-propagate adder row.

Informally, the correctness of the horizontal refinement of the carry-save step is obtained by the following equality reasoning:

$$\begin{array}{cl}
 & sv' + 2^{i+2}cv' \\
= & \{ \text{ decompositions } \} \\
 & 2^{N+i}sv'(N+i) + 2^{i+1}sv'(N+i-1, i+1) + sv'(i,0) + 2^{i+2}cv' \\
= & \{ \text{ algebra } \} \\
 & 2^{N+i}sv'(N+i) + 2^{i+1}(sv'(N+i-1, i+1) + 2cv') + sv'(i,0) \\
= & \{ \text{ functional behavior } \} \\
 & 2^{N+i}yv(N-1) + 2^{i+1}(sv(N+i-1, i+1) + cv + yv(N-2,0)) + sv(i,0) \\
= & \{ \text{ algebra } \} \\
 & 2^{N-1}2^{i+1}yv(N-1) + 2^{i+1}yv(N-2,0) + 2^{i+1}cv \\
 & +2^{i+1}sv(N+i-1, i+1) + sv(i,0) \\
= & \{ \text{ compositions } \} \\
 & 2^{i+1}yv + s^{i+1}cv + sv
\end{array}$$

The composition and decomposition steps above are based on the following arithmetic equations over unsigned interpretations of bit-vectors:

$$\begin{array}{rcll}
xv(i-1,0) + 2^{i}xv(n-1,i) & = & xv & \text{(composition)} \\
xv \ o \ yv & = & 2^{m}xv + yv & \text{(decomposition)}
\end{array}$$

Here, xv and yv are bit-vectors of length n and m, respectively, and o denotes concatenation of bit-vectors. The correctness proof for the horizontal refinement of the carry-propagate step works analogously. Finally, it is straightforward to prove, by induction, the overall correctness of iterative array multiplication (see also Section 6) from the word-level specifications of the vertical stages.

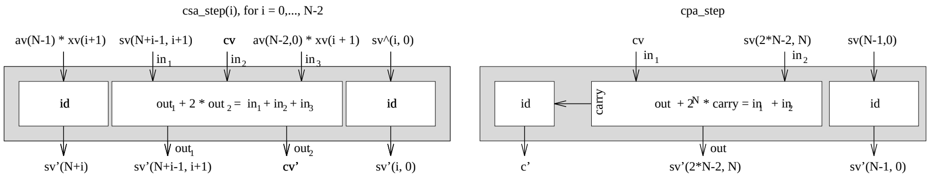

Figure 3: Horizontal Refinement

Altogether, this two-dimensional decomposition of the circuit leads to the following hierarchical strategy for proving the overall correctness of this multiplier:

- prove the correctness of gate-level implementations of carry-save and carry-propagate adders;
- assuming correct adder rows, prove the correctness of the horizontal decompositions;
- assuming the correctness of the horizontal decomposition prove the correctness of the vertical decomposition, and, consequently, the overall correctness of the circuit.

4 Implementation and Verification of Adders

This section describes gate-level implementations of iteratively generated `N`-bit carry-propagate and carry-save adder rows. Their arithmetic behavior is stated in terms of equations over word-level interpretations of the input and output signals. The proofs of these characteristic theorems are carried out automatically in PVS.

The theory `cpa` in $\boxed{2}$ describes the implementation and the corresponding correctness statement for the carry-propagate adder. This theory is parameterized with respect to the size `N` of the bit-vector inputs and output. It imports a theory `full_adder`[1] which contains a gate-level implementation and the arithmetic specification of a full adder with output `carry` bit and `sum` bit, and the theory `bv` which specificies the bit-vector type (`bvec[N]`) and related bit-vector functions. The carry bit that ripples through the full adders is specified recursively by means of the function `nth_cin`. The function `cpa` defines the carry

[1]The theories `full_adder`, `bit_mult`, and `csa` are not included in this paper for lack of space.

output to be the N-th carry, and the n-th sum output of the carry propagate adder is defined to be the sum output of the n-th full adder applied to the n-th bits of the input bit-vectors and the n-th carry.

```
                                                                      2
cpa[N: posnat]: THEORY
  BEGIN
   IMPORTING full_adder, bitvectors@bv[N], bitvectors@bv_nat

   xv, yv: VAR bvec[N]; n: VAR below[N]; bj: VAR upto[N]

   nth_cin(j, xv, yv):RECURSIVE bit =
    IF j = 0 THEN 0
    ELSE carry(FA(xv(j - 1), yv(j -1), nth_cin(j - 1, xv, yv))) ENDIF
   MEASURE j

   cpa(xv, yv): [# carry: bit, sum: bvec[N] #] =
    (# carry := nth_cin(N, xv, yv),
       sum   := (LAMBDA n: sum(FA(xv(n), yv(n), nth_cin(n,xv,yv)))) #)

   cpa_inv: LEMMA
    bv2nat_rec(j, sum(cpa(xv, yv)))
    = bv2nat_rec(j,xv) + bv2nat_rec(j,yv) - exp2(j) * nth_cin(j,xv,yv)

   cpa_char: THEOREM
    sum(cpa(xv, yv)) = xv + yv - exp2(N) * carry(cpa(xv, yv))
  END cpa
```

Having described an implementation, we now characterize its functionality. The theorem cpa_char in $\boxed{2}$ expresses the conventional correctness statement of an adder row on the *word-level*, and relates the gate-level implementation of cpa with the functional level. This characteristic theorem of cpa is an immediate consequence of the invariant cpa_inv in $\boxed{2}$, and the proof of this invariant proceeds by induction on the variable j using an induction scheme for the type upto[N].

```
(induct-and-simplify "j" :name "upto_induction[N]"
                         :exclude "FA"
                         :theories  "full_adder")
```

In this case, the use of the parameters exclude and theories yields a certain economy of proof construction, since they direct the induction strategy to apply the arithmetic characterization of the full adder as specified in the theory full_adder instead of blindly unfolding the definition of the function FA. It takes about 15 seconds to complete the proof on a Sun Sparcstation 10.

Correctness proofs for N-bit carry-propagate adders have been presented for many different logics and proving systems. Cyrluk et al. [CRSS94], for example, define a specialized proof strategy for proving a slight variant of the above carry-propagate adder and other hardware theorems. An alternative approach of synthesizing word-level properties of iteratively generated circuits from properties of the functional units has been proposed in [HDL90]. As shown above, however, the improved general-purpose induction strategies of the PVS prover already suffice to automatically prove the correctness of inductively generated carry-propagate adders.

In a similar way, one defines a carry-save addition row as a function `csa` such that the theorem `csa_char` holds.

3

```
csa(xv, yv, cv: bvec[N]): [# sum, carry: bvec[N] #] = ...

csa_char: THEOREM
 sum(csa(xv, yv, cv)) = xv + yv + cv - 2 * carry(csa(xv, yv, cv))
```

The PVS proof script of this theorem is identical with the proof strategy used above for the correctness result of carry-propagate addition.

5 Formalization of Horizontal and Vertical Composition

Formalization of the horizontal and the vertical refinements of the iterative array multiplier, as discussed in Section 3, are developed in this section.

The formalization of the carry-propagate step as depicted in Figure 3 is straightforward. The function `cpa_step` in 4 calls the carry-propagate adder on the carry vector `cv` and the upper half of the sum vector `sv`, and the resulting output sum vector is created by concatenating the carry output with the sum output and the lower half of the input sum.[2]

4

```
cpa_step((cv: bvec[N - 1]), (sv: bvec[2 * N - 1])): bvec[2 * N] =
  LET cpa = cpa(sv^(2 * N - 2, N), cv) IN
    carry(cpa) o sum(cpa) o sv^(N - 1, 0)
```

The formalization of carry-save steps `csa_step(i)` is more complicated and requires the concept of *dependent types*, since the types of both the input parameters and the result depends on the value of the index `i`.

[2]Here, a conversion from `bit` to `bvec[1]` is assumed.

5

```
csa_step(i:below[N - 1])
        ((sv: bvec[N+i]), (cv: bvec[N-1]), (yv:bvec[N]))
: [# sum: bvec[N + i + 1], carry: bvec[N - 1] #] =
  LET csa = csa(sv^(N + i - 1, i + 1), yv^(N - 2, 0), cv) IN
    (# sum   := yv(N - 1) o sum(csa) o sv^(i, 0),
       carry := carry(csa)                       #)
```

The function csa_step uses PVS's dependent typing features and the result of this function is a tuple consisting of output sum and carry-out vector. It is a straightforward transliteration of the block diagram of the carry-save step in Figure 3. Note also that the role of dependent types for hardware specification (and verification) has already been noted in [HDL90].

Finally, the carry-save and the carry-propagate stages are combined to form an iterative array multiplier in exactly the same way as described in Section 3.[3]

6

```
connect_csa((i: upto[N - 2]), (av, xv: bvec[N])):
 RECURSIVE [# carry: bvec[N - 1], sum: bvec[N + i + 1] #] =
 (IF i=0 THEN
    csa_step(0)(bit_mult(av, xv(0)), bvec0, bit_mult(av, xv(1)))
  ELSE
    csa_step(i)(sum(connect_csa(i - 1, av, xv)),
                carry(connect_csa(i - 1, av, xv)),
                bit_mult(av, xv(i + 1)))
  ENDIF)
MEASURE i

iam(av, xv: bvec[N]): bvec[2 * N] =
 cpa_step(carry(connect_csa(N-2)(av,xv)),sum(connect_csa(N-2)(av,xv)))
```

The iterative array multiplier iam in $\boxed{6}$ consists of (N - 2) stages of appropriately wired carry-save steps csa_step(i) followed by a final carry-propagate step. This finishes the formalization of this multiplier.

6 Correctness of the High-Level Design

The mechanized correctness proof of the iterative array multiplier is described in this section. This correctness proof is parameterized[4] with respect to the lower-level building blocks in Section 4 such as carry-save and carry-propagate adder rows and formation of partial products. It does not, however, refer to

[3] bvec0 denotes a constant 0 bit-vector of appropriate length.
[4] In a way described more clearly in Section 7.

the details of the gate-level implementations and only uses the characteristic behavioral theorems such as cpa_char in $\boxed{2}$ and csa_char in $\boxed{3}$.

The correctness proofs of the horizontal refinements need the following decomposition and composition lemmas as has been demonstrated in Section 3. Let n,m be positive natural numbers, and xv, yv be bit-vectors of sizes n and m, respecively, and i of type subrange[1,n-1]; then:[5]

7

```
decompose: LEMMA xv o yv = xv * exp2(m) + yv
compose  : LEMMA xv^(i - 1, 0) + xv^(n - 1, i) * exp2(i) = xv
```

First, the correctness of the horizontal refinement of the carry-save step is expressed by the following lemma:

8

```
csa_step_char: LEMMA
 FORALL ((i:below[N-1]),(sv:bvec[N+i]),(cv:bvec[N-1]),(xv:bvec[N])):
   sum(csa_step(i, sv, cv, xv))
   = sv + exp2(i + 1) * xv + exp2(i + 1) * cv
     - exp2(i + 2) * carry(csa_step(i, sv, cv, xv))
```

Unfortunately, the PVS prover needs some guidance to apply the compose lemma above in an appropriate way, since the rewrite engine of PVS does not apply *commutative-associative* matching; the appropriate matching terms, however, are typically found in independent branches of a nested +-structure. Altogether 15, mostly trivial, interactive steps are necessary to complete this proof.

Second, the behavioral specification given in Figure 2 for the carry-propagate step

9

```
cpa_step_char: LEMMA
  FORALL (cv: bvec[N - 1], sv: bvec[2 * N - 1]):
    cpa_step(cv, sv) = sv + cv * exp2(N)
```

is proved automatically. Skolemization followed by unfolding the definition cpa_step and repeated calls to the decision procedures discharge this proof

```
(then (skosimp*) (expand "cpa_step") (repeat (assert)))
```

after introducing lemmas compose, decompose, and the behavioral constraint cpa_char (see $\boxed{2}$) as rewrites to the prover.

Third and finally, the lemmas csa_step_char and cpa_step_char permit an easy induction proof of the behavioral correctness iam_char of the multiplier.

[5]Slight variants of these lemmas are stated and proved in the PVS bit-vector library.

```
                                                                        10
av, xv: VAR bvec[N]; i: VAR upto[N - 2]

iam_inv: LEMMA
 sum(connect_csa(i, av, xv))
  = av * bv2nat_rec(i+2,xv) - exp2(i+2) * carry(connect_csa(i,av,xv))

iam_char: THEOREM iam(av, xv) = av * xv
```

The induction scheme for proving `iam_inv` in $\boxed{10}$ is given by

```
(induct "i" :name "upto_induction[N - 2]")
```

Both the base case and the induction step are proved with a combination of calls to the decision procedures and rewriting.

7 The use of Generic Hardware Components

Behavioral specifications of many hardware components are sufficient to specify and reason about higher-level building blocks; other details are often irrelevant. Consequently, developments on higher levels abstract from specific gate-level implementations of the basic building blocks. In this way, *generic hardware components* allow verification of circuits to be done in a hierarchical fashion. It is demonstrated in this section how features of the PVS specification language such as parameterization of theories and predicate subtypes support this use of generic hardware components.

In our example, formalizations of higher-level designs related to the horizontal and vertical decomposition (see Section 3) can be developed in the theory `iam` in $\boxed{11}$. This theory is parameterized with respect to three *generic* hardware components; namely, a carry-save adder `csa`, a "carry-propagate" adder `cpa`, and the function `bit_mult` which computes a partial product.

Behavioral constraints associated with generic hardware components are specificed in PVS in the `ASSUMPTIONS` part, and any instantiation of the formal parameters incurs the obligation to prove all of the `ASSUMPTIONS`. Obviously, the last two generic hardware components in $\boxed{11}$ can be realized by the gate-level adder implementations in $\boxed{3}$ and $\boxed{2}$, respectively. A useful variation of the circuit in Figure 1, however, can be realized by replacing the final *ripple-carry adder* with a fast two-operand carry-lookahead adder, if a shorter overall execution time is desired. In this case, all developments within theory `iam` are "inherited" for this new circuit as long as the assumption `cpa_char` in $\boxed{11}$ can be shown to hold.

```
                                                                     11
iam[ (IMPORTING bitvectors@bv, ...)
 N       : upfrom(3),
 bit_mult: [bvec[N], bit -> bvec[N]],
 csa: [bvec[N-1], bvec[N-1], bvec[N-1] -> [# carry, sum: bvec[N-1] #]],
 cpa: [bvec[N-1], bvec[N-1] -> [# carry: bit, sum: bvec[N - 1]#]]
]: THEORY
 BEGIN
  ASSUMING
   xv, yv, cv: VAR bvec[N - 1]; av: VAR bvec[N]; x: VAR bit

   bit_mult_char: ASSUMPTION bit_mult(av, x) =av * x
   csa_char     : ASSUMPTION
      sum(csa(xv, yv, cv)) = xv + yv + cv - 2 * carry(csa(xv, yv, cv))
   cpa_char     : ASSUMPTION
      sum(cpa(xv, yv)) = xv + yv - exp2(N-1) * carry(cpa(xv, yv))
  ENDASSUMING

     (... see Sections 5 and 6 ...)
 END iam
```

Instead of using explicit ASSUMPTIONS constraints on the generic hardware components as shown above, one could also define predicate subtypes for each of these components and parameterize the theory iam in $\boxed{11}$ with respect to these subtypes. The type BIT_MULT in $\boxed{12}$, for example, characterizes the set of valid bit_mult functions.

```
                                                                     12
BIT_MULT[N: posnat]: THEORY
  BEGIN
   BIT_MULT: TYPE =
    { bit_mult: [bvec[N], bit -> bvec] |
         FORALL (xv: bvec[N], y: bit):  bit_mult(xv, y) = xv * y }
  END BIT_MULT
```

In PVS, theory parameterization must be used to define this family of types indexed over N.

```
                                                                     13
iam[ (IMPORTING BIT_MULT, ...)
 N: upfrom[3], bit_mult: BIT_MULT[N], csa: CSA[N], cpa: CPA[N]
]: THEORY
 BEGIN ... END iam
```

Using the type definition in $\boxed{12}$ and analogously defined types CSA[N] and CPA[N] the parameterized theory iam in $\boxed{11}$ is rewritten in a very concise way as shown in $\boxed{13}$.

8 Related Work

The well-publized FDIV error of initial releases of the Pentium floating-point divider sparked renewed interest in the verification of arithmetic circuits [Bry95, CB96, KS96, RSS96, ML96, CGZ96, LO95, AS95, KS96]. For the large number of recent publications in this area, we restrict ourselves to a comparison to work that we think is most closely related to this paper.

Chen and Bryant [BC94, CB96] use *Binary Moment Diagrams* (BMDs) to verify iterative multipliers for word sizes up to 256 in a hierarchical fashion. At the lower levels, a set of component modules are described at both the bit-level (as combinational circuits) and at the word-level (as algebraic expressions). They verify that the bit-level implementation of each block implements its word-level specification. At the higher levels, the system is described as an interconnection of components having word-level representations, and the specification is also given at the word-level. The verification task makes sure that the composition of the block functions corresponds to the system specification. The verification is, after the manual decomposition of the problem and a suitable choice for the variable ordering, fully automatic. The array multiplier they are verifying is similar to the one in Figure 1, and their hierarchical decomposition of this multiplier is analogous to the one presented in this paper. The verification approach based on BMDs, however, only supports the verification of fixed-sized circuits, while our approach is based on inductive theorem proving and proves, once and for all, the correctness of circuits of arbitrary size. Moreover, circuit descriptions in their specification language ACV are still rather low-level, since this language does not allow to express parameterized designs and sequencing features.

Kapur and Subrahmaniam [KS96] mechanically verify multiplier circuits in RLL by developing a common top-level specification that abstracts multiplication in terms of a component that computes the partial sums and another that adds these partial sums to compute the final product; a similar decomposition of multipliers that mimic the usual process of longhand multiplication also been been proposed by [Chi92]. The top-level specification in terms of conditional equations is parameterized over circuit-specific details such as the number of partial sums computed in each stage and the adder interconnection. Kapur and Subrahmaniam "instantiate" this top-level scheme with various multipliers such as Wallace tree and an array multiplier. Bit-vectors are encoded in [KS96] as linear lists. One consequence of this encoding are side conditions about the length of bit-vector representation in their specifications. This complication is avoided in our approach by using bit-vectors of fixed-sized length, since, in this case, illicit expressions are already detected by the type-checker. Moreover, the specifications in this paper reflect the structure of the underlying circuit and the sizes of the data-path more closely than the ones given in [KS96]; one reason for this is the represention of right-shifts in [KS96] by appending trailing zeros to the bit-vectors. Also, the functional representation of the array multiplier in this paper allows one to generate verified gate-level implementations.

The construction of the correctness proofs in [KS96] requires a minimum of user interactions (such as manual orientation of some equations) and are very fast; this is mainly due to efficient rewriting and the discovery of appropriate induction schemes by cover set induction. With our functional approach of generating the circuits by means of recursive functions, however, selection of appropriate induction schemes was always straightforward.

9 Concluding Remarks

A two-dimensional high-speed multiplier circuit has been specified and formally verified in a hierarchical way using the PVS system. It took about 2 weeks for an — back then — inexperienced PVS user to complete this case study. Altogether, the specifications of this multiplier consist of 5 separate theories with about 220 lines of code.

This verification exercise demonstrates the value of an expressive specification language in mechanized verification. In particular, the use of dependent types together with predicate subtypes and theory parameterization allows the writing of specifications that are clear, concise, and generic. Such high-level descriptions not only minimize the pitfalls of introducing errors in initial design specification but also open the door to using these specifications as design documents. Moreover, the modular specification approach and the use of generic hardware components supports modular proof development and has the advantage that some variations of a particular circuit design can be verified by just redoing one part of the proof. It is, for example, a simple matter to substitute the carry-propagate adder used in this paper with a (faster) carry-lookahead adder.

The construction of the proof requires about 50 user interactions, most of which are as trivial as introducing rewrites to the prover or explicitly calling the decision procedures. All the lemmas about unsigned interpretations of bit-vectors that were required to finish the proof could already be found in the bit-vector library; the main lemmas in $\boxed{7}$, however, had to be restated in a different form to make the PVS correctness proof "more automatic".

Altogether, the combination and tight integration of term rewriting with efficient decision procedures in PVS proved to be a useful workhorse, since most theorems could be proved fairly automatically. Only the proofs at the higher-levels needed some guidance. It is conjectured, however, that a PVS system extended with standard theorem proving techniques such as associative-commutative term rewriting (and perhaps a slight variation of the current induction strategy) could prove all the theorems in this paper fully automatically.

Another promising approach of automating this and other hardware-related proofs is to extend PVS with efficient decision procedures for theories about bit-vectors of fixed, but arbitrary size N. Moreover, further work is needed to demonstrate the general applicability of the hierarchical decomposition of two-dimensionally generated circuits into vertical and horizontal stages as proposed in this paper.

References

[AH96] R. Alur and T.A. Henzinger, editors. *Computer-Aided Verification, CAV '96*, volume 1102 of *Lecture Notes in Computer Science*, New Brunswick, NJ, July/August 1996. Springer-Verlag.

[AS95] M.D. Aagaard and C.J.H. Seger. The Formal Verification of a Pipelined Double-Precision IEEE Floating-Point Multiplier. In *Proc. of ICCAD'95*, pages 7–10. IEEE Computer Science Press, 1995.

[BC94] R.E. Bryant and Y.A. Chen. Verification of Arithmetic Circuits with Binary Moment Diagrams. Technical Report CMU-CS-94-160, School of Computer Science, Carnegie Mellon University, 1994.

[Bry94] R.E. Bryant. Verification of Arithmetic Functions with Binary Moment Diagrams. Technical Report CMU-CS-94-160, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, 1994.

[Bry95] R.E. Bryant. Bit-Level Analysis of an SRT Divider Circuit. Technical Report CMU-CS-95-140, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, April 1995.

[CB96] Y.A. Chen and R.E. Bryant. ACV: An Arithmetic Circuit Verifier. In *IEEE/ACM International Conference on Computer Aided Design (ICCAD-96)*, San Jose, CA, nov 1996. to appear.

[CGZ96] E. M. Clarke, S. M. German, and X. Zhao. Verifying the SRT division algorithm using theorem proving techniques. In Alur and Henzinger [AH96], pages 111–122.

[Chi92] S.K. Chin. Verified Functions for Generating Signed-Binary Arithmetic Hardware. *IEEE Transactions on Computer-Aided Design*, 11(2):1529–1558, December 1992.

[CRSS94] D. Cyrluk, S. Rajan, N. Shankar, and M. Srivas. Effective Theorem Proving for Hardware Verification. In R. Kumar and Th. Kropf, editors, *Theorem Provers in Circuit Design*, number 901 in Lecture Notes in Computer Science, pages 203–222, 1994.

[HDL90] F.K. Hanna, N. Daeche, and M. Longley. Specification and Verification Using Dependent Types. *IEEE Transactions on Software Engineering*, 16(9):949–964, September 1990.

[Kor93] I. Koren. *Computer Arithmetic Algorithms*. Prentice-Hall, 1993.

[KS96] D. Kapur and M. Subramaniam. Mechanically verifying a family of multiplier circuits. In Alur and Henzinger [AH96], pages 135–146.

[LO95] M. Leeser and J. O'Leary. Verification of a Subtractive Radix-2 Square Root Algorithm and Implementation. In *Proc. of ICCD'95*, pages 526–531. IEEE Computer Society Press, 1995.

[ML96] P.S. Miner and J.F. Leathrum, Jr. Verification of IEEE compliant subtractive division algorithms. In M. Srivas, editor, *Formal Methods in Computer-Aided Design (FMCAD '96)*, Palo Alto, CA, November 1996. To appear.

[ORSvH95] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal Verification for Fault-Tolerant Architectures: Prolegomena to the Design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[RSS96] H. Rueß, N. Shankar, and M. K. Srivas. Modular verification of SRT division. In Alur and Henzinger [AH96], pages 123–134.

Acknowledgments This case study has been conducted while visiting the formal verification group at SRI International, Menlo Park. I would like to thank S. Owre, J. Rushby, N. Shankar, and M. Srivas for all the support and useful feedback, and H. Pfeifer for proof-reading.