Mechanically Verifying a Family of Multiplier Circuits *

Deepak Kapur M. Subramaniam

Computer Science Department
State University of New York
Albany, NY 12222
kapur@cs.albany.edu, subu@cs.albany.edu

Abstract. A methodology for mechanically verifying a family of parameterized multiplier circuits, including many well-known multiplier circuits such as the linear array, the Wallace tree and the 7-3 multiplier is proposed. A top level specification for these multipliers is obtained by abstracting the commonality in their behavior. The behavioral correctness of any multiplier in the family can be mechanically verified by a uniform proof strategy. Proofs of properties are done by rewriting and induction using an automated theorem prover *RRL* (Rewrite Rule Laboratory). The behavioral correctness of the circuits is established with respect to addition and multiplication on numbers. The automated proofs involve minimal user intervention in terms of intermediate lemmas required. Generic hardware components are used to segregate the specification and the implementation aspects, enabling verification of circuits in terms of behavioral constraints that can be realized in different ways. The use of generic components aids reuse of proofs and helps modularize the correctness proofs, allowing verification to go hand in hand with the hardware design process in a hierarchical fashion.

1 Introduction

There has been a great deal of interest in verifying properties of hardware circuits at the input-output level. Many papers on this topic have appeared in conference proceedings and journals[10], to cite a few [3, 5, 8, 11, 6, 17]. Different approaches have been proposed in the literature, notably among them state-based approaches and the use of model checkers [5, 3], induction-based approaches adapted from software verification [8, 12] and finally approaches based on modeling hardware circuits using higher-order logics [6, 11].

Despite this widespread interest, verification efforts involving multiplier circuits have been few in comparison[14, 4, 13]. The state based approaches and model checking that employ binary decision diagrams (BDDs) or some variant of these, do not perform well on multiplier circuits due to the associated state explosion (see further discussion on this in the next section on related work). It is possible to verify the correctness of multipliers using theorem provers and proof checkers but such efforts have also been limited as they are ad hoc in nature and require considerable user ingenuity.

* Partially supported by the National Science Foundation Grant no. CCR-9308016.

The focus of this paper is on the use of an automated theorem prover for mechanically verifying parameterized multiplier circuits. A methodology for specifying and verifying a family of parameterized multipliers circuits is described. The behavioral correctness of many well-known multiplier circuits such as the linear array, the Wallace tree and the 7-3 multipliers, with respect to addition and multiplication over numbers, is mechanically verified using an automated theorem prover *RRL* (*Rewrite Rule Laboratory*)[15].

We first develop a common top level equational specification for a family of multiplier circuits by abstracting the commonality in behavior. A multiplier circuit is abstracted in terms of two components: a component that computes the partial sums, called the ***partial sum computation*** component and another that adds these partial sums to compute the final product, called the ***partial sum addition*** component. We then describe a uniform approach for mechanically verifying the correctness of any multiplier in the family using *RRL*. It is shown that the correctness of any multiplier circuit in the family can be mechanically established from the behavioral correctness of the partial sum computation component and that of the partial sum addition component. The correctness of these components follow from the correctness of the adder circuits used in them.

The proposed approach is highly generic – not only abstracting over the word size of multiplier circuits but also abstracting the common behavior of a variety of different multiplier circuits. The proofs of correctness are obtained for multiplier circuits of arbitrary word size. Secondly, seemingly different multiplier circuits share a common specification and proof of correctness using the same lemmas, with only a few different definitions for each multiplier circuit.

A major complaint against the use of theorem provers and proof checkers for hardware verification has been the semi-automatic nature of these systems. Verification efforts using these systems involve considerable user ingenuity. We believe that a common top level proof for a family of multiplier circuits with well-characterized intermediate lemmas that are independent of the underlying prover, is a step in addressing this issue. It is shown that the intermediate lemmas used in the proofs of correctness of multipliers reported here correspond to formulas that specify the input-output behavior expressed in terms of numbers, of different components of the circuits. Such lemmas, we speculate, can be generated systematically from the structure of the circuits.

In our specifications and proofs of various multiplier circuits, we abstract the adders in terms of generic hardware components with associated behavioral constraints. The correctness of the multiplier circuits is first established in terms of such generic components. It is shown later how a particular adder can realize a generic component by demonstrating that the adder satisfies the behavioral constraints of the generic component. Such a view provides a clear separation between specification and implementation aspects. The use of generic components aids reuse of proofs and modularize the correctness proofs, allowing verification to go hand in hand with the design process in a hierarchical fashion. Such modularization of proofs is crucial for any verification methodology to effectively scale up to larger and more complex hardware circuits.

Let us briefly review main aspects of different multiplier circuits considered in this paper. A linear array multiplier performs the multiplication of two n bit numbers in linear time by computing the partial sums corresponding to the given numbers and adding the partial sums together to obtain the required result. Addition of partial sums is done by considering one partial sum at a time. Wallace in [20] introduced a multiplication scheme, which has popularly come to be known as the Wallace tree multiplier, for multiplying two n bit numbers in logarithmic time. Improved performance is achieved in the Wallace tree multiplier by considering three partial sums for addition together. The multiplication scheme due to Wallace was generalized and improved upon by Dadda in [7] leading to a rich family of multipliers called the *Dadda multipliers*. In these multipliers, larger than three partial sums are taken up for addition at a particular time. Considering larger number of partial sums does not improve the asymptotic complexity but considerably reduces the number of stages required for multiplication resulting in reduced wiring delays. The 7-3 multiplier used in *IBM RS/6000* is based on this observation and has been attributed [9] as one of the important features that contributes to its good performance.

Most of these multiplier circuits are based on the grade school principle of multiplying any two given n bit numbers–computing the partial sums and adding the partial sums to obtain the required result. This basic underlying principle is often not evident in commonly found descriptions of these circuits. The computation of partial sums is done in the same manner in these circuits, and these circuits differ only in the number of partial sums that they consider for addition at any particular time. A common top level specification for the family of multiplier circuits based on this observation is developed in Section 3. In section 4, the behavioral correctness of different multiplier circuits with respect to addition and multiplication over numbers are presented. The use of generic hardware components in specifying and verifying different multiplier circuits using *RRL* is discussed in section 5.

2 Related Work

Among the various approaches employed for hardware verification, the state based approaches based on symbolic manipulation of boolean functions using binary decision diagrams *BDDs* [3] are perhaps the most popular for verifying hardware circuits of fixed word size (non-parametric circuits). A circuit is specified using a boolean function that can be succinctly represented using a *BDD*. Further *BDD*s provide a fast mechanism for comparing boolean functions. Even for linear circuits, in which the output is a linear function of the inputs, this approach has two major limitations: (i) it is unclear how circuits of arbitrary word size can be verified, and (ii) verification is limited to showing that a circuit implements a boolean function, and not a function on numbers.

It is well-known that for many important boolean functions, especially the ones for multiplication, that grow exponentially with the word size, the state-based approaches are less attractive for verification. Bryant and Chen recently introduced a new data structure *Multiplicative Binary Moment Diagram (BMD)* for modeling the functionality of circuits in terms of data at the word level [4].

Using this approach, a number of integer multiplier designs with word sizes up to 256 bits have been verified. However, such verifications are not fully automatic as Bryant and Chen in [4] state:

....the overall circuit is divided into components, each having a word level specification. Verification involves proving 1) that each component implements its word level specification and 2) that the composition of the word level component functions matches the specification.....

The approach advocated in this paper using a theorem prover *RRL* for verifying multiplier circuits is similar to the one suggested using BMDs. We decompose a multiplier circuit into two components, and establish the number-theoretic correctness of the individual components. The overall proof then follows by the composition of these two components. The automated proofs obtained using our theorem prover *RRL* do not entail any additional overheads. Due to the generality afforded by theorem provers like *RRL*, it was further possible to obtain common proof for a family of multiplier circuits of arbitrary sizes (parametric circuits) which would be infeasible otherwise.

Approaches based on theorem provers and proof checkers have been widely used to verify hardware circuits. Most of this effort has focussed on verification of different forms of processors [11, 17, 8], different forms of ALUs [19, 8] or has been used for the verification of adder circuits [19, 8, 16, 12]. In [14], a Braun Multiplier is formally specified using the Boyer-Moore logic and some properties about this specification are proven using *Nqthm* [1].

In [18], a framework for synthesizing a variety of hardware circuits including the carry save and Wallace tree multipliers is proposed. Higer order metafunctions with different circuit interconnection structures such as the carry save array and the Wallace tree as inputs are **manually** transformed to realize multipliers at the gate level. The correctness of the circuits is established by reasoning about the behavior of these metafunctions and the associated transformations using the automated theorem prover *HOL*. We are unaware of other mechanical verification efforts where the correctness of multipliers such as the Wallace tree multiplier or the 7-3 multiplier have been mechanically established with minimal user guidance using a uniform framework such as ours.

3 Specifying a family of Multiplier Circuits

A common, top level equational specification for a family of multiplier circuits is developed in this section. The Wallace tree multiplier is used as an example to illustrate the methodology. The overall structure of the Wallace tree multiplier can be described diagrammatically as in Fig. 1.

Given bit vectors x and y of equal length, a Wallace tree circuit first computes a list of partial sums ($P1, \cdots, P8$ in Fig. 1) using a function such as *psum-all*. Each partial sum in the list is a bit vector that corresponds to a single bit of x and is obtained by shifting y appropriately. The partial sums in the list are then added together by adding in parallel three partial sums at a time. Addition of any three partial sums is typically done using a carry save adder(CSA) that has three bit vectors as its inputs and produces a pair of bit vectors as its output.

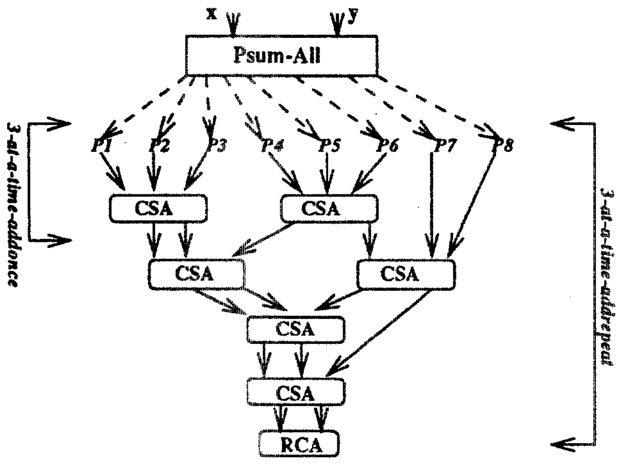

Fig. 1.

The outputs correspond to the bitwise sum and the bitwise carry of the inputs.[2] The parallel addition of three bit vectors at a time is repeated on the outputs of the carry save adder until we have only two bit vectors left. The final product is obtained by using a ripple carry (RCA in Fig. 1.) or a carry lookahead adder.

Specifying Partial Sums Computation in RRL: A bit vector is modeled in *RRL* as a list of bits (with 0 denoting bit zero and 1 denoting bit one) with *nl* and *cons*. A list of bit vectors is modeled as a list of lists with *lnl* denoting the empty list of lists and *consl* that adds a list to a list of lists.

Contrary to the usual convention, we assume that the bits increase in order from left to right i.e., the bit vector 01 stands for $0 * 2^0 + 1 * 2^1 = 2$.

The partial sum, *psum* corresponding to a single bit `x1` of `x` is the same as `y` if `x1` is 1; otherwise, it is the zero bit vector of the same length as `y`[3]:

```
psum(x1, y) := cond(x1 = 0, mkzero(y), y),
```

where `mkzero` generates a zero bit vector of the same length as its input.

The list of partial sums corresponding to all the bits of `x` is computed by applying the function `psum` pointwise to each bit of `x` and shifting `y` to the right by appending a *trailing* zero.

```
psum-all(nl, y) := lnl
psum-all(cons(x1, x), y)  := consl(psum(x1, y), psum-all(x, cons(0, y)))
```

Specifying Partial Sums Addition in RRL: In a Wallace tree circuit, each level in the tree in Fig. 1. contains a list of bit vectors that have to be added to produce the final result. The root contains the list of partial sums corresponding to each bit of the bit vector `x`. The successive levels of the tree are repeatedly constructed until there are less than three bit vectors at any given level(equations

[2] Further details on the specification of the carry save adder are given in section 5.

[3] We follow the convention of typesetting RRL specifications and italicizing other logical formulas.

1, 2 and 3 below). In the case of two bit vectors, addition using a ripple carry adder, `rca`, is performed(equation 3 below).

The Wallace tree multiplier is specified by `3-mult` below. The trace of a computation of `3-mult` on input vectors of a specific length corresponds to a specific circuit.

```
3-mult(cons(x1, nl), y)          :=  psum(x1, y)
3-mult(cons(x1, cons(x2, nl)), y) :=
  rca(0, pad(1, psum(x1, y)), psum(x2, cons(0, y)))
3-mult(cons(x1, cons(x2, cons(x3, x))), y)  :=
  3-repeat(psum-all(cons(x1, cons(x2, cons(x3, x))), y)).
```

The function `3-repeat` repeatedly takes 3 bit vectors and add them; it is specified:

```
3-repeat(lnl)                          := nl
3-repeat(consl(x1,lnl))                := x1
3-repeat(consl(x1,consl(x2, lnl)))     := rca(0, pad(1, x1), x2) if
                                          (len(pad(1, x1)) = len(x2))
3-repeat(consl(x1, consl(x2, consl(x3, x)))) :=
  3-repeat(3-once(consl(x1, consl(x2, consl(x3, x))))),
```

where `len` denotes the length of a list. The function `pad(m, x)` produces a bit vector by appending `m` *leading* zeroes to the bit vector `x`. Bit vectors are typically padded by leading zeroes in these specifications so that the input bit vectors to the adders are of equal length. The last equation (equation 4) computes the bit vectors at the successive level by the function `3-once`.

The function `3-once` is defined on a list of bit vectors. If the input list contains less than three bit vectors(equations 1, 2 and 3 below), then the bit vectors in the input list are carried over to the output list. Otherwise, the bit vectors in the input list by considering bit vectors in groups of three and adding such groups in parallel using a carry save adder, `csa`, (equation 4 below). The outputs of the `csa`'s and the bit vectors in the input list that were not considered for addition together, constitute the bit vectors of the output list.

```
3-once(lnl)                                 := lnl
3-once(consl(x1, lnl))                      := consl(x1, lnl)
3-once(consl(x1, consl(x2, lnl)))           := consl(x1, consl(x2, lnl))
3-once(consl(x1, consl(x2, consl(x3, x)))) := consl(fst(z1), consl(snd(z1)
                                               3-once(x))) if
  (z1 = carrysave-adder(pad(2, x1), pad(1, x2), x3)) and
  (len(pad(2, x1)) = len(x3)) and (len(pad(1, x2)) = len(x3))
```

3.1 A common top level specification for multipliers in RRL

Circuits that perform multiplication of two n bit numbers by first computing the partial sums and then adding these partial sums constitute a rich family of multipliers based on the number of partial sums that they consider for addition at a particular time. Any multiplier of this family can be specified in *RRL* using the same top level specification as that of the Wallace tree multiplier.

Consider a multiplier circuit defined by the function `k-mult` in which k ($k \geq 1$) partial sums are added together at any time. The multiplier can be abstracted in terms of two hardware components that are cascaded together. The first of these components performs ***partial sum computation*** with two bit vectors as its inputs and produces a list of bit vectors as its output. The second of these components performs ***partial sum addition*** with a list of bit vectors as its input and produces a bit vector as its output.[4]

The partial sum computation component of the multiplier is specified by the functions `psum` and `psum-all`. The partial sum addition component is specified in terms of: `k-repeat` which adds k partial sums repeatedly, and `k-once` for adding partial sums at one level until there are fewer than k partial sums left.

The function `k-once` is defined in the same way as `3-once`. The function leaves the input list of bit vectors invariant if the list contains less than k (more precisely, maximum of $k-1$ and 1) bit vectors. Otherwise, a suitable adder is used to add the bit vectors in the list k at a time with the outputs of the adder constituting the bit vectors to be added in the next round. The definitions of the functions `k-mult` and `k-repeat` can be generalized from the definitions of `3-mult` and `3-repeat` respectively in a similar fashion. [5]

4 Establishing the correctness of multipliers in RRL

In this section we discuss how the behavioral correctness of multiplier circuits can be automatically established using *RRL*. *RRL* is a theorem prover based on rewriting techniques and induction. The main inference steps used in *RRL* are (i) contextual simplification using rewrite rules, (ii) case analysis, (iii) decision procedures for data types with free constructors, propositional calculus and quantifier-free Presburger arithmetic for reasoning about numbers, and (iv) proofs by well-founded induction. *RRL* implements many heuristics to select the order of application of these inferences. For more details on *RRL* the reader is referred to [15].

Consider a multiplier specified by `k-mult` that performs multiplication of its two input bit vectors x and y by considering k, $k \geq 1$, partial sums for addition at a time. To establish the correctness of this circuit with respect to multiplication over numbers, conversion functions from bit vectors and list of bit vectors to numbers are needed. The function `bton` converts a bit vector to the number it represents (recall that the first bit is the least significant bit).

```
bton(nl) := 0,
bton(cons(x1, x)) := cond(x1 = 0, 2 * bton(x), 1 + (2 * bton(x)).
```

Given a list of bit vectors as input, the function `btonlist` below defines a linear addition of numbers corresponding to each of the bit vectors.

```
btonlist(lnl)  :=  0,      btonlist(consl(x, y)) := bton(x) + btonlist(y).
```

[4] k itself can be treated as a parameter while adding the partial sums. Such a specification and the correctness proof can be found in *ftp.cs.albany.edu/pub/subu/Multipiers.* The specification uses generic adders (discussed in section 5). Instantiating such adders requires discharging assumptions on lengthes of the lists of bit vectors in terms of k and would be discussed in the expanded version of this paper.

[5] The complete specifications of the linear array, the Wallace tree and the 7-3 multiplier as done in *RRL* along with the *RRL* transcripts of their correctness proof are also available by anonymous ftp.

The correctness of a multiplier `k-mult` is stated in *RRL* as:

```
Kmult-thm: bton(k-mult(x, y)) == bton(x) * bton(y) if (len(x) = len(y)).
```

The basic strategy employed for proving the above theorem is simple. It involves characterizing the input-output behavior of the partial sum computation component and the partial sum addition component of the multiplier with respect to numbers, and then showing that cascading these two components leads to the desired overall behavior. It is shown that *i)* multiplying the numbers corresponding to the input bit vectors of the partial sum computation component is the same as number obtained by the linear addition of the list of partial sums output by this component. *ii)* And, the number corresponding to the bit vector output by the partial sum addition component is the same as the number corresponding to the linear addition of the list of partial sums input.

The same strategy can be used to prove the correctness of the correctness of any multiplier in the family of multipliers (for any fixed k). Linear addition of partial sums serves as a common denomination for any k and the addition of k partial sums together can always be reduced to linear addition.

Speculating the Intermediate Lemmas

Intermediate lemmas capturing the behavior of each of the component circuits are first established. For instance, lemma `L1` below states that the ripple carry adder correctly implements addition over numbers (needed for the final stage).

```
L1: bton(rca(0, y, z))  == bton(y) + bton(z) if (len(y) = len(z)).
```

There is a similar lemma for carry-save adders (lemma `L4` in section 5). Lemma `L2` captures the correctness of the behavior of the partial sum addition component; it states the number corresponding to the output bit vector is precisely the one obtained by adding numbers corresponding to the list of input bit vectors. Finally, `L3` relates the number corresponding to the list of bit vectors output by the partial sum computation component to the product of the numbers corresponding to its two input bit vectors.

```
L2:  bton(3-repeat(x)) == btonlist(x).
L3: btonlist(psum-all(x, y)) == bton(x) * bton(y)
```

Each of these lemmas can be verified completely automatically in *RRL* by the cover set induction method [15] and the associated heuristics.

We believe that each of the above intermediate lemmas can be speculated from the structure of the multiplier circuit. Lemmas relate the input-output behavior of components of a multiplier circuit with respect to numbers. For each component in the circuit, the number corresponding to its output bit vector (or a list of vectors) is related to the numbers corresponding to its input bit vectors. This important issue of generating intermediate lemmas from the circuit structure needs further investigation; the approach based on generating lemmas from the component behavior seems to be very promising.

For instance, the Wallace tree multiplier can be viewed as a linear composition of the ripple carry adder(`rca`), the partial sum addition (`3-repeat`), and the partial sum computation (`psum-all`) components. The main theorem `3mult-thm` can be expressed as:

```
bton(x) * bton(y)  = bton(rca(0, 3-repeat(psum-all(x, y)))),
```

by identifying the list of bit vectors output by `3-repeat` with the two input bit vectors of `rca`. The number theoretic correctness of the circuit `3mult-thm` can be reduced to the number theoretic correctness of each of these components relating their corresponding inputs and outputs. These correspond to the intermediate lemmas `L1-L3`. Lemma `small L4` can be speculated from the use of `3-once` in the iterative component `3-repeat`.

Establishing the correctness of the Wallace tree multiplier in RRL:

Below, we briefly review the proof of Wallace tree multiplier as obtained in *RRL* using the above-discussed strategy using lemmas `L1, L2, L3, L4`. Other proofs are similar.[6]

The correctness of the Wallace tree multiplier is stated in *RRL* as:

```
3mult-thm: bton(3-mult(x, y)) == bton(x) * bton(y) if (len(x) = len(y)).
```

The above theorem was proved in *RRL* by induction. Induction scheme based on the definition of the function `3-mult` is automatically chosen by the heuristics implemented in *RRL* without any user guidance. Here is the *RRL* transcript.

```
Let P(x): bton(3-mult(x, y)) == (bton(x) * bton(y)) if (len(x)  = len(y))
Induction will be done on x in 3-mult(x, y), with the scheme:
[1] P(cons(x1, nl))                        [2] P(cons(x1, cons(x2, nl)))
[3] P(cons(x1, cons(x2, cons(x3,  x))))
```

The subgoal corresponding to [1] is easily established by case analyses based on the definition of `psum`, using the definitions of `3-mult` and `bton` for simplification. The case analyses is automatically recognized by *RRL* based on the definition of `psum` given in terms of the ternary predicate *cond*.

The subgoal [2] follows from lemma `L1` (ensuring that the ripple carry adder correctly implements addition over numbers). The proof of the subgoal [3] is also direct from lemmas `L2, L3`, thus completing the proof of `3mult-thm` by induction.

The correctness of any other multiplier in the family of multipliers such as the linear array or the 7-3 multiplier can be performed in *RRL* using the same top level proof as that of the Wallace tree multiplier given above. For instance, the correctness of a linear array multiplier is proved in *RRL* using three lemmas which are exactly the same as `L1 - L3` with the lemma `L2` defined in terms of functions `1-repeat` instead of the function `3-repeat`. The correctness proof of the 7-3 multiplier also follows from the lemmas `L1 - L3` with the lemma `L2` defined in terms of the functions `7-repeat` instead of `3-repeat`.

5 The use of Generic Hardware Components

While proving the correctness of different multipliers , the specifications and the associated correctness proofs of the adders are duplicated. Such duplication can be avoided by noting that specifications of the input-output behavior of the adders is sufficient to reason about different multipliers; other details of adders

[6] Detailed proof transcripts are available via anonymous ftp from *ftp.cs.albany.edu: pub/subu/Multipliers.*

are irrelevant. So adder circuits are abstracted by generic hardware components with behavioral constraints. The correctness proof of multipliers is first performed in terms of these generic components. The generic components are then realized by specific adders that satisfy the associated behavioral constraints.

To specify and reason over generic hardware components, *RRL* has been extended along the lines of [2] to allow function instantiations and for handling theories. The behavioral constraints associated with a generic component are specified in *RRL* as equations(possibly conditional) using ? = to indicate that the equation is a behavioral constraint. For instance, a carry save adder can be specified in *RRL* in terms of the generic component g32-adder as:

```
bton(fst(g32-adder(x, y, z))) + bton(snd(g32-adder(x, y, z)))  ?=
bton(x) + bton(y) + bton(z) if (len(x) = len(y) = len(z)).
```

The behavioral constraints introduced on these generic components are oriented into rewrite rules by *RRL* and are subsequently available for simplification.

5.1 Realizing the generic components : Carry Save Adder

To complete correctness proofs of different multipliers, the generic components used are realized by specific adders that satisfy the associated behavioral constraints. In this section we use the correctness proof of a carry save adder as an example to realize the generic component g32-adder. The other generic components used in the proofs of the multiplier circuits have been realized similarly using *RRL*. For details refer to [12].

A carry save adder has three bit vectors of equal length as its inputs and outputs two bit vectors corresponding to the bitwise parity and the bitwise sum of its inputs. It is specified in *RRL* as:

```
csa(x, y, z) :=  pairl(paritylst(x, y, z), cons(0, majoritylst(x, y, z)))
                    if (len(x) = len(y) and (len(y) = len(z)),
```

where pairl given two bit vectors constructs a pair of bit vectors. The function paritylst, computes the bitwise parity of its three inputs, and the function majoritylst computes their bitwise majority. These functions can be easily defined by invoking parity and majority functions on bits.

The correctness of the carry save adder can be stated as:

```
L4: bton(x) + bton(y) + bton(z) == bton(paritytlst(x, y, z)) + bton(cons(0,
    majoritylst(x, y, z))) if (len(x) = len(y)) and (len(y) = len(z)).
```

The above formula is proved directly in *RRL* by induction using the scheme based on the definition of paritylst.

The component g32-adder can be realized by the carrysave adder, csa in *RRL* using the *instantiate* directive with a set of *function replacements* such as ((g32-adder csa),...). Based on these function replacements the behavioral constraints are suitably instantiated by *RRL* and the instantiated formula is treated as a proof obligation which must be discharged from the properties of the realization.

6 Conclusion

A number of well-known multiplier circuits such as the linear array, the Wallace tree and the 7-3 multiplier employed in *IBM RS/6000* have been verified using the automated theorem prover *RRL*. It has been shown that by abstracting the commonality in behavior, a family of multiplier circuits can be specified using a common top level specification. Such a specification was used to illustrate a common top level correctness proof for the family of multiplier circuits. The basic strategy employed in performing these correctness proofs is simple, and it leads to concise proofs with a handful of meaningful lemmas that are independent of the underlying prover. It should be possible to duplicate these proofs using other provers which support capabilities similar to those implemented in *RRL*.

Circuit	*Comm. Defs*	*Comm. Lemm.*	*Spec. Defs*	*Spec. Lemm.*	*Time*
Linear Array			2	0	2.48
Wallace Tree	12	5	2	0	2.45
7-3			2	0	6.22

The intermediate lemmas used in these proofs correspond to the input-output behavior of the various components of the multiplier circuit. Speculation of such lemmas can be done by the user in a routine manner. The use of generic components to segregate the specification and implementation aspects was advocated. The use of such generic components lead to concise proofs and help reuse of proofs. It was also demonstrated that generic components lead to modular proof development in a hierarchical fashion analogous to the design process.

The specification and the correctness proofs of the Wallace tree multiplier were attempted first in *RRL* and it took less than a week. This time is inclusive of our attempts to familiarize ourselves with the multiplier itself. The subsequent multipliers were formalized and their correctness proof was proved in a couple of days. The statistics for the various correctness proofs obtained using *RRL* are given in the table. *RRL* is implemented in Common Lisp and the timings are on a Sun Sparc 5 station(64Mb memory). The proofs of the linear array and the Wallace tree multiplier can be performed in *RRL* within 5 secs. The time required for the 7-3 multiplier is larger due to extensive contextual rewriting required for establishing the appropriateness of the lengthes of seven bit vectors. There are no specific intermediate lemmas needed in the proofs. For each multiplier circuit, only two definitions specific to the circuit are needed.

The results of our initial experiments, in terms of adder circuits [12] and multiplier circuits performed in *RRL*, are encouraging, and they lead us to believe that *RRL* is well-suited for reasoning about the properties of hardware descriptions using recursive equations that can be oriented into rewrite rules. Particularly, *RRL* can be used for verifying properties of parameterized circuits, which cannot be handled by state based approaches, as well as for structuring proofs of larger circuits in terms of proofs of their component circuits. Further, circuit properties are verified in terms of the arithmetic functions they compute in contrast to boolean functions. The major stumbling block in the use of theorem provers is perhaps the need for intermediate lemmas. As shown for adder and multiplier circuits, these lemmas correspond to capturing the arithmetic function of the component circuits; generation of such lemmas, we speculate, can be automated.

References

1. R.S. Boyer and J. Moore, *A Computational Logic Handbook.* New York: Academic Press, 1988.
2. R.S. Boyer, J. Moore and M. Kaufmann "Functional Instantiation in Nqthm", CLI Inc. Tech. Report.
3. Bryant R.E., "Graph-based Algorithms for boolean function manipulation", *IEEE trans. on Computers*, C-35(8), 1986.
4. R. E. Bryant, and Y.-A. Chen, "Verification of Arithmetic Functions with Binary Moment Diagrams", Tech. Rep. CMU-CS-94-160, June 1994.
5. J. R. Burch, E.M. Clarke, K. L. Mcmillan and D.L. Dill, "Sequential Circuit Verification using symbolic model checking", in proceedings of *Twenty seventh ACM/IEEE Design Automation Conference*, 1990.
6. A.J. Camilleri, M.J.C. Gordon and T.F.Melham, "Hardware verification using higher-order logic"", *HDL Descriptions to Guaranteed Correct Circuit Designs*, D. Borrione (editor) pp. 43-67, N.Holland, Amsterdam 1987.
7. L. Dadda "Some Schemes for parallel multipliers," *in Computer Arithmetic Vol. I*, E.E. Swartzlander Jr. (editor), IEEE Computer Society Press, 1990.
8. W.A. Hunt., "FM8501: A verified Microprocessor", Ph.D thesis, UT Austin, 1985.
9. R.K.Montoye, E. Hokenek and S.L.Runyon, "Design of the IBM RISC System/6000 floating-point execution unit," *IBM Journal*, Vol. 34, No. 1, 1990.
10. "Theorem Provers in circuit design",*IFIP Transactions*, V. Stavridou, T.F. Melham, R.T.Boute (eds.) N.Holland 1992.
11. J. Joyce, G. Birtwistle and M. Gordon, "Proving a computer correct in HOL", Tech. Report 100, Computer Lab. University of Cambridge 1986.
12. D. Kapur and M. Subramaniam, "Mechanical Verification of Adder Circuits Using Powerlists," CS.Tech. Report, Dept. of CS Suny Albany, November 1995.
13. R.P. Kurhshan, L. Lamport, "Verification of a Multiplier: 64 Bits and Beyond," *Fifth Intl. Conf. on CAV*, C. Courcoubetis (editor), LNCS 697, July 1993.
14. L. Pierre, "VHDL Description and Formal Verification of Systolic Multipliers," *in Proc. of CHDL*, D. Agnew and L. Claesen (eds.) N. Holland 1993.
15. D. Kapur, and H. Zhang, "An overview of Rewrite Rule Laboratory (RRL)," *J. of Computer and Mathematics with Applications*, 29, 2, 1995, 91-114.
16. D. Cyrluk and S. Rajan and N. Shankar and M. K. Srivas, "Effective Theorem Proving for Hardware Verification", Proc. 2nd *conference on theorem provers in circuit design*, R. Kumar and T. Kropf (eds.), Sept. 1994.
17. M. Srivas and M. Bickford, "Formal Verification of a pipelined microprocessor.", *IEEE Software*, Sept. 1990.
18. Shui-Kai Chin, "Verified Functions for Generating Signed-Binary Arithmetic Hardware", *IEEE trans. on Computer Aided Design*, Vol. 11, No. 12, Dec. 1992.
19. D. Verkest, L. Claesen, and H. De Man, "Correctness Proofs of Parameterized Hardware Modules in the Cathedral-II Synthesis Environment", *EDAC'90*, Glasgow, Scotland, March 1990.
20. C.S. Wallace, "A Suggestion for a fast multiplier," *in IEEE Trans. Electron. Comput.*, EC-13:14-17, 1964.