ACV: An Arithmetic Circuit Verifier*

Yirng-An Chen
yachen@cs.cmu.edu

Randal E. Bryant
Randy.Bryant@cs.cmu.edu

Carnegie Mellon University, Pittsburgh, PA 15213

Abstract

*Based on a hierarchical verification methodology, we present an arithmetic circuit verifier ACV, in which circuits expressed in a hardware description language, also called ACV, are symbolically verified using Binary Decision Diagrams for Boolean functions and multiplicative Binary Moment Diagrams (*BMDs) for word-level functions. A circuit is described in ACV as a hierarchy of modules. Each module has a structural definition as an interconnection of logic gates and other modules. Modules may also have functional descriptions, declaring the numeric encodings of the inputs and outputs, as well as specifying their functionality in terms of arithmetic expressions. Verification then proceeds recursively, proving that each module in the hierarchy having a functional description, including the top-level one, realizes its specification. The language and the verifier contain additional enhancements for overcoming some of the difficulties in applying *BMD-based verification to circuits computing functions such as division and square root. ACV has successfully verified a number of circuits, implementing such functions as multiplication, division, and square root, with word sizes up to 256 bits.*

1 Introduction

The well-known division bug in Intel's Pentium processor [6] has illustrated the importance of proving the correctness of arithmetic circuit designs. It has brought industry and research attention to the verification of arithmetic circuits.

In an earlier paper [1], we showed that multiplicative Binary Moment Diagrams (*BMDs) provide a powerful method for verifying arithmetic circuits. *BMDs provide a canonical representation for "word-level" functions, mapping Boolean variables to numeric values. They can represent a number of arithmetic functions, such as multiplication and addition, in compact form. Our hierarchical methodology exploits the modular structure of arithmetic circuits, in which complex circuits are constructed from simpler ones, which themselves compute arithmetic functions. We verify that the individual modules compute their specified functions, compose these word-level functions according to the module interconnections, and verify that these compositions match the overall specification. In this earlier work, we successfully executed the steps to verify a number of multiplier circuits with word sizes up to 256 bits. These steps were directly encoded as a sequence of calls to our *BMD library routines.

In this paper, we describe an arithmetic circuit verifier ACV that works automatically from a description of the circuit in a hardware description language. Besides supporting the hierarchical verification methodology described in our earlier paper, ACV implements several extensions to the methodology, making it possible to verify a wider range of circuits, including ones for division and square root.

Since our earlier paper, several others have published related work. One drawback of a hierarchical approach is that users must define a partitioning of the circuit and provide word-level specifications for the modules. Hamaguchi and his colleagues[5] developed a method to construct a word-level, *BMD representation directly from a flat, gate-level circuit by composing the gate functions in reverse topological order. Their approach works reasonably well for correctly designed multiplier circuits, although requiring somewhat greater computing time and memory than ours. A small design error, however, can easily cause their method to blow up—not a desirable property for practical verification tool. Having specifications for both the overall circuit and the individual components makes it much easier to pinpoint a discrepancy between the circuit and its specification. In addition, for functions, such as divide and square root, our experiments indicate that the computing time for Hamaguchi's method grows exponentially with the word size, whereas our (extended) methodology can handle these functions.

Clarke, *et al* [3] extended BMDs to a form they call Hybrid Decision Diagrams (HDDs), in which a function may be decomposed with respect to each variable in one of four ways without edge weight in their representation. They also extended the symbolic model verifier(SMV) to handle word-level properties [4]. Using HDDs and extended SMV, they have verified a variety of circuits, including a radix-4 SRT divider. Their verifier represents the transition relation for the circuit using BDDs. Hence, it cannot directly handle circuits with complex combinational logic, such as array multipliers. A more recent version of their program [2] allows users to define a partitioning of the combinational logic and uses a variant of Hamaguchi's method to compose the circuit functions. With this user-specified partitioning, their method becomes very similar to ours.

Theorem provers can also be used to verify circuits hierarchically [7]. Compared with our approach, they must use much deeper hierarchies. For example, while verifying an adder, they first verify a one-bit adder cell and then verify the whole adder. This process can be quite tedious, especially when the circuit employs performance enhancements such as lookahead carry chains. Our approach can verify such components as adders directly.

Our verifier requires that both the circuit and its specification be given in a hardware description language, also called ACV, specifically tailored to the needs of our verification methodology. This language supports hierarchical definitions, where each module is composed structurally from other modules and gate-level primitives. In addition, a module can have a word-level specification, consisting of definitions of the numeric encodings of the inputs and outputs, as well as arithmetic expressions defining the functionality. Additional enhancements, described later in this paper, support extensions to our verifier for overcoming some of the limitations of *BMD-based verification. In particular, the language allows shifting the roles of inputs and outputs in the module hierarchy, introduction of auxiliary "pseudo"-inputs, specifying range constraints among module I/O signals, and cutting signals within modules to simplify their word-level representations.

The choice of whether to extend an existing HDL, e.g., by adding annotations to VHDL, or to design an entirely new language involve a variety of technical and sociological trade-offs. For this project, where we are more concerned with pushing the horizons of formal verification than with verifying existing circuits, we have followed the latter course. As future research, we are considering several techniques for working with more standardized circuit descriptions.

In the remainder of this paper we first give an overview of the ACV language and how it supports hierarchical verification. Then we describe several enhancements, using an SRT radix-4 divider circuit as a case study. Next, we show experimental results for a number of arithmetic circuits. We conclude with a brief discussion of future work.

* This research is sponsored by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the Advanced Research Projects Agency (ARPA) under grant number F33615-93-1-1330.

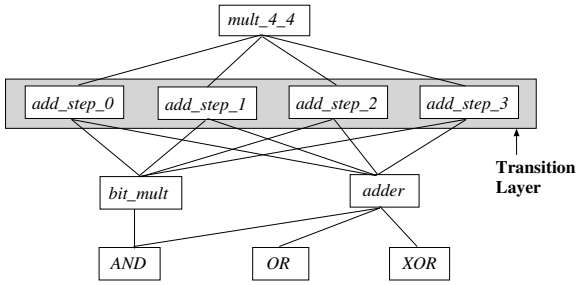

Figure 1: **Module hierarchy of 4×4 multiplier.** Each module in the transition layer is the last module with word-level specifications on the path down from the root.

2 Hierarchical Verification with ACV

In our earlier paper[1], we proposed *BMD-based hierarchical verification for verifying arithmetic circuits such as multipliers. Hierarchical verification is based on the principle that functions and circuits can be divided into sub-functions and sub-circuits which can be verified independently. Applying a divide-and-conquer method to verification, we first verify that the individual modules compute their specified functions, compose these word-level functions according to the module interconnections, and verify that these compositions match the overall specification.

To support this approach, we devised a hardware description language, called ACV, to describe circuits and their specifications in a hierarchical manner. Each module is composed structurally from other modules and primitive logic gates. In addition, a module can be given the word-level specification consisting of definitions of the numeric encodings of inputs and outputs, as well as the module functionality in terms of arithmetic expressions relating input and output values.

We use a 4×4 array multiplier to illustrate the ACV language and system. This multiplier can be represented by the module hierarchy shown in Figure 1. Readers can reference our earlier paper[1] for the detailed circuit design. We define the "transition layer", shown as the shaded box in Figure 1, as the collection of modules which are the last modules with word-level specifications on the paths down from the root. Modules in or above the transition layer must declare their word-level specifications, as well as their structural definitions. Modules below the transition layer just declare their structural definitions. Modules in the transition layer abstract from the bit-level, where the structure consists of logic gates (sub-module will be evaluated recursively), to a word-level representation, where the structure consists of blocks interconnected by bit-vectors encoding numeric values.

Figure 2 shows the ACV description of the top module of a 4×4 array multiplier. The definition of a module is encompassed between keywords "MODULE" and "ENDMODULE". First, the module name and the names of signals visible from outside of this module must be given as shown in the first line of Figure 2. The module is declared as $mult_4_4$ with three signals x, y, and p. Then, the width of these signals are declared in the VAR section. Both x and y are declared as 4 bits wide, and p are 8 bits.

```
MODULE mult_4_4(x, y, p)
VAR          p[8],x[4],y[4];
ENCODING     P = (unsigned) p;
             X = (unsigned)x;
             Y = (unsigned)y;
FUNCTION     P == X*Y;
VERIFY       P == X*Y;
ORDERING     x,y;
INTERNAL     s1[4],s2[6],s3[7];
STRUCTURE    add_step_0(y[0],X,s1);
             add_step_1(y[1],X,s1,s2);
             add_step_2(y[2],X,s2,s3);
             add_step_3(y[3],X,s3,p);
ENDMODULE
```

Figure 2: **ACV code for Module $mult_4_4$ of a 4×4 multiplier.**

For each module, section INTERNAL and STRUCTURE define the circuit connections among logic gates and sub-modules. The INTERNAL section declares the names and widths of internal vector signals used in the STRUCTURE section to connect the circuit. Vector $s1$, $s2$ and $s3$ are declared as 4, 6 and 7 bits, respectively. There are two types of statements in the STRUCTURE section. First, the assignment statements, shown in lines 1, 2, 4, 5 and 6 in the STRUCTURE section of Figure 4, are used to rename part of a signal vector, or to connect the output of a primitive logic gate. Second, the module instantiation statements, shown in the STRUCTURE section of Figure 2, declare which signals are connected to the referenced modules. Note that we do not distinguish inputs from outputs in module instantiation statements and module definitions. As we shall see, it is often advantageous to shift the roles of inputs and outputs as we move up in the module hierarchy. The ACV program will distinguish them during the verification process based on the information given in the specification sections.

To give the word-level specification for a module, sections ENCODING, FUNCTION, VERIFY and ORDERING are required in the module definition. The ENCODING section gives the numeric encodings of the signals declared in the VAR section. For example, vector p is declared as having an unsigned encoding and its word-level value is denoted by P. The allowed encoding types are: unsigned, two's complement, one's complement, and sign-magnitude. The FUNCTION section gives the word-level arithmetic expressions for how this module should be viewed by modules higher in the hierarchy. For example, if module $mult_4_4$ were used by a higher level module, its function would be to compute output P as the product of inputs X and Y. In general, the variable on the left side of "==" will be treated as output and the variables on the right side will be treated as inputs. The VERIFY section declares the specification which will be verified against its circuit implementation. In the multiplier example, the module specification is the same as its function. In other cases, such as the SRT divider example in next section, these two may differ to allow a shifting of viewpoints as we move up in the hierarchy. The ORDERING section not only specifies the BDD variable ordering for the inputs but also defines which signals should be treated as inputs during the verification of this module. The variable ordering is very important to verification, because our program does not currently do dynamic variable reordering.

The ACV program proceeds recursively beginning with the top-level module. It performs four tasks for each module. First, it verifies the sub-modules if they have word-level specifications. Second, it evaluates the statements in the STRUCTURE section in the order of their appearance to compute the output functions. For a module in the transition layer, this involves first computing a BDD representation of the individual module output bits by recursively evaluating the sub-module's statements given in their STRUCTURE sections. These BDDs are then converted to a vector of bit-level *BMDs, and then a single word-level *BMD is derived by applying the declared output encoding. For a module above the transition layer, evaluation involves composing the submodule functions given in their FUNCTION sections. Third, ACV checks whether the module specification given in the VERIFY section is satisfied, Finally, it checks whether the specification given in the VERIFY section implies the module function given in the FUNCTION section. A flag is maintained for each module indicating whether this module has been verified. Thus, even if a module is instantiated multiple times in the hierarchy, it will be verified only once.

For example, the verification of the 4-bit array multiplier in Figure 2 begins with the verification of the four add_step modules. For each

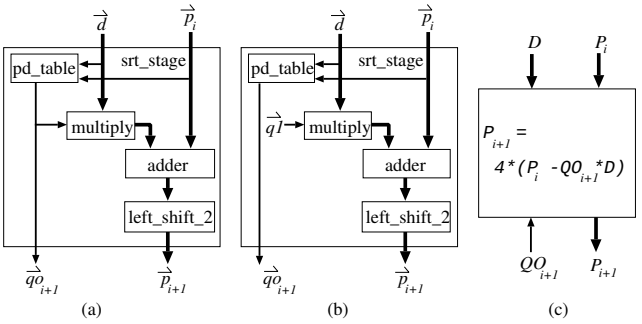

Figure 3: **Block level representation of SRT divider stage from different perspectives.** (a) The original circuit design. (b) The abstract view of the module, while verifying it. (c) The abstract view of the module, when it is referenced.

one, the structural definition as well as the structural definitions it references are evaluated recursively using BDDs to derive a bit-level representation of the module output. These BDDs are converted to *BMDs, and then a word-level *BMD is derived by computing the weighted sum of the bits. ACV checks whether the circuit matches the specification given in its VERIFY section. The specification of add_step module i is $Out = In + 2^i * y * X$, where In is a partial sum input (0 for i=0), y is a bit of the multiplier and Out is the output of the module.

Assuming the four add_step modules are verified correctly, ACV derives a *BMD representation of the multiplier output. It first creates *BMD variables for the bit vectors x and y (4 each), and computes *BMD representations of X and Y by computing weighted sums of these bits. It evaluates the add_step instantiations to derive a word-level representation of module output. First, it computes $s1$ by evaluating the FUNCTION statement $Out = y * X$ of module add_step_0 for the bindings $y = y_0$ and $X = X$. Then it computes $s2$ by evaluating the FUNCTION statement $Out = In + 2 * y * X$ of module add_step_1 for the bindings $In = s1$, $y = y_1$, and $X = X$. This process continues for the other two modules, yielding a *BMD for P equivalent to $P = (((y_0 * X) + 2 * y_1 * X) + 2^2 * y_2 * X) + 2^3 * y_3 * X$. Note that whether a module argument is an input or an output is determined by whether it has a binding at the time of module instantiation. ACV then compares the *BMD for P to the one computed by evaluating $X * Y$ and finds that they are identical. Finally, checking whether the specification in the VERIFY section implies the functionality given in the FUNCTION section is trivial for this case, since they are identical.

3 Additional Methodologies

We use radix-4 SRT division as an example to illustrate the use of the ACV language, and to explain several additional verification methodologies.

A divider based on the radix-4 SRT algorithm is an iterative design maintaining two words of state: a partial remainder and a partial quotient, initialized to the dividend and 0, respectively. Each iteration extracts two bits worth of quotient, subtracts the correspondingly weighted value of the divider from the partial remainder, and shifts the partial remainder left by 2 bit positions. The logic implementing one iteration is shown in Figure 3.a, where we do not show two registers storing partial remainder and partial quotient. The inputs are divisor $\vec{d}$ and partial remainder $\vec{p}_i$, and the outputs are the extracted quotient digit $\vec{qo}_{i+1}$ (ranging from -2 to 2) and the updated partial remainder $\vec{p}_{i+1}$. The PD table, used to look up the quotient digits based on the truncated values of the divisor and the partial remainder, is implemented in logic gates derived from a sum of products form. After the iterations, the set of obtained quotient digits is converted into the actual quotient by a quotient conversion circuit.

First, we prove the correctness of one iteration of the circuit. The specification is given in [6] and is shown as Equation 1. This specification states that for all legal inputs (i.e., satisfying the range constraint) the outputs also satisfy the range constraint, and that the inputs and outputs are properly related. This specification captures the essence of the SRT algorithm.

$$(-8D \leq 3P_i \leq 8D) \rightarrow \{(-8D \leq 3P_{i+1} \leq 8D) \wedge [P_{i+1} == 4(P - QO_{i+1} * D)]\} \quad (1)$$

This specification contains word-level function comparisons such as $\leq$ and == as well as Boolean connectives $\wedge$ and $\rightarrow$. In [3], a branch-and-bound algorithm is proposed to do word-level comparison operations for HDDs. It takes two word-level functions and generates a BDD representing the set of assignments satisfying the comparison operation. We adapted their algorithm for *BMDs to allow ACV to perform the word-level comparisons. Once these "predicates" are converted to BDDs, we use BDD operations to evaluate the logic expression.

If Equation 1 is used to verify this module, the running time will grow exponentially with the word size, because the time to convert output $\vec{p}_{i+1}$ in Figure 3(a) from a vector of Boolean functions into a word-level function grows exponentially with the word size. The reason is that $\vec{p}_{i+1}$ depends on output vector $\vec{qo}_{i+1}$ which itself has a complex function. We overcome this problem by cutting off the dependence of $\vec{p}_{i+1}$ on $\vec{qo}_{i+1}$ by introducing an auxiliary vector of variables $\vec{q1}$, shown in Figure 3(b). One can view this as a cutting of the connection from the PD table to the multiply component in the circuit design. Now, the task of verifying this module becomes to prove that Equation 2 holds:

$$(-8D \leq 3P_i \leq 8D \wedge QO_{i+1} == Q1) \rightarrow \{(-8D \leq 3P_{i+1} \leq 8D) \wedge [P_{i+1} == 4(P - Q1 * D)]\} \quad (2)$$

In the actual design, the requirement that $QO_{i+1} == Q1$ is guaranteed by the circuit structure. Hence Equation 2 is simply an alternate definition of the module behavior. By this methodology, the computing time of verifying this specification is reduced dramatically with a little overhead (the computing time of performing $QO_{i+1} == Q1$ and an extra AND operation). The major difference between this cutting methodology and the hierarchical partitioning is that the latter decomposes the specification into several sub-specifications, but the former only introduces auxiliary variables to simplify the computation. We can also apply this methodology to verify the iteration stage of such similar circuits as restoring division, restoring square root and radix-4 SRT square root.

Module srt_stage, shown in Figure 4, implements the function of one SRT iteration for a 6×6 divider using the ACV language. Vector variables, p, d, qo and $p1$ in Figure 4, represent signal vectors, $\vec{p}_i$, $\vec{d}$, $\vec{qo}_{i+1}$ and $\vec{p}_{i+1}$ in Figure 3(a), respectively. Their encoding and ordering information is given in the relevant sections. Modules $shifter$ and $negater$ implements module multiply in Figure 4(a). Since *BMDs can only represent integers, we must scale all numbers so that binary point is at the right. We specify one additional condition in the specification: that the most significant bit of the divider must be 1, by the term $D \geq 2$**5.

The support for our "cutting" methodology arises in several places. First, vector $q1$ is declared in the VAR and ORDERING sections with the same size as qo, and is therefore treated as a "pseudo input", i.e., an input invisible to the outside. Then, the equivalence of signals qo and $q1$ is declared in the EQUIVALENT section. The original signal qo must appear first in the pair. While evaluating the statements in the STRUCTURE section, ACV automatically uses $q1$'s value instead of qo's value for signal qo once signal qo has been assigned its value. For example, all appearances of signal qo after the pd_table instantiation in Figure 4 will use $q1$'s value (a *BMD $Q1$ using three Boolean variables) instead of its original value (a *BMD function

MODULE $srt_stage(p, d, qo, p1)$
VAR qo[3],$q1$[3],p[9],d[6],$p1$[9];
EQUIVALENT (qo,$q1$);
ENCODING P = (twocomp) p;
$P1$ = (twocomp)$p1$;
D = (unsigned)d;
QO = (signmag)qo;
$Q1$ = (signmag)$q1$;
FUNCTION $P1$ == 4*(P-QO*D);
VERIFY
($3*P \leq 8*D$ & $3*P \geq -8*D$ & $Q1$ == QO & $D \geq 2**5$)
$\rightarrow$ ($3*P1 \leq 8*D$ & $3*P1 \geq -8*D$ & $P1$ == 4*(P-$Q1$*D));
ORDERING $q1$,p,d;
INTERNAL ph[7],dh[4],t[9],nqd[9],r[10];
STRUCTURE $ph = p[2 .. 8]$;
$dh = d[1 .. 4]$;
$pd_table(ph, dh, qo)$;
$w1$ = qo[0];
$w2$ = qo[1];
neg = not(qo[2]);
$shifter(d, w1, w2, t)$;
$negater(t, neg, nqd)$;
$adder(p, nqd, neg, r)$;
$left_shift_2(r, p1)$;
ENDMODULE

Figure 4: **ACV code for Module srt_stage.**

of inputs P and D) when evaluating these statements. Finally, the encoding method of $q1$ is declared the same as q0 and Equation 2 is used in the VERIFY section instead of Equation 1.

Figure 5(a) shows the block level representation of a 6×6 SRT divider. Since module srt_stage performs a cycle of SRT division, we instantiate it multiple times, effectively unrolling the sequential SRT division into a combinational one, and compose them with another module $Conversion$ which takes the set of quotient digits generated from the stages and converts them into a quotient vector with an unsigned binary representation. The divider takes two inputs P and D, goes through 3 srt_stage and 1 $Conversion$ modules, and generates the outputs Q and R. Module $Conversion$ takes a set of quotient digits, generated from the srt_stages, and converts them into a vector in the unsigned binary form. Assume module $Conversion$ takes inputs $\vec{q_0}$, ..., $\vec{q_n}$, and produces the output $\vec{q}$. The specification of this module is $Q = Q_n + 4 * Q_{n-1} + ... + 4^n * Q_0$, where Q and Q_i are the word-level representations of $\vec{q}$ and $\vec{q_i}$, $0 \leq i \leq n$.

With the partitioning shown in Figure 5(a), we cannot directly apply hierarchical verification, because the outputs of module srt_stage do not have unique functional definitions. The redundant encoding of the quotient digits in the SRT algorithm allows, in several cases, a choice of values for the quotient digits. Fortunately, we do know the relation between inputs and outputs: $P_{i+1} = 4 * (P_i - QO_{i+1} * D)$. We exploit the fact that the correctness of the overall circuit behavior does not depend on the individual output functions, but rather on their relation. Therefore we can apply a technique similar to one used to verify circuits with carry-save adders[1] treating the quotient output as an input when this module is instantiated. Figure 3(c) shows this abstract view of the srt_stage module when it is referenced. The abstract view of the SRT divider is then changed as shown in Figure 5(b), and described in ACV as shown in Figure 6. The quotient output vectors $\vec{q2}$, $\vec{q1}$ and $\vec{q0}$ (denoted by $Q2$, $Q1$ and $Q0$ for the word-level representation) of three srt_stage modules are changed to pseudo inputs by declaring them in the VAR, ENCODING and ORDERING sections. With this additional information, the circuit is effectively changed from Figure 5(a) to Figure 5(b) without modifying the physical connections.

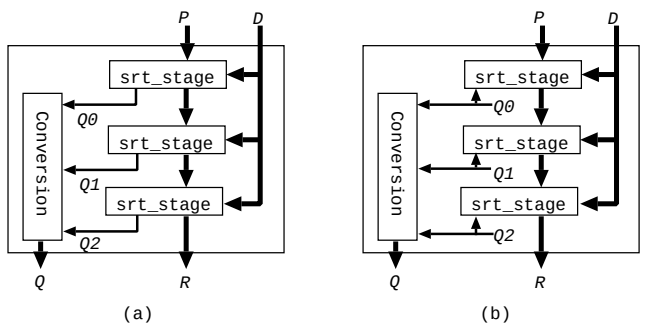

Figure 5: **Block level representation of a 6×6 SRT divider from two different perspectives.** (a) The original circuit design. (b) the abstract view of the module, while verifying it.

MODULE $srt_div_6_6(p, d, q, r)$
VAR p[6], d[6], q[6], r[9],$q0$[3],$q1$[3], $q2$[3];
ENCODING P = (unsigned) p;
D = (unsigned) d;
Q = (unsigned) q;
R = (twocomp) r;
$Q0$ = (signmag) $q0$;
$Q1$ = (signmag) $q1$;
$Q2$ = (signmag) $q2$;
FUNCTION R == 2**6 * P - 4*D*Q;
VERIFY ($3*P \leq 8*D$ & $3*P \geq -8*D$ & $D \geq 2**5$) $\rightarrow$
((2**6 * P) == 4*D*Q + R);
ORDERING $p, d, q2, q1, q0$;
INTERNAL $p0$[9],$p1$[9],$p2$[9];
STRUCTURE $p0$[0 .. 5]= p;
$p0$[6 .. 8]= 0;
$srt_stage(p0, d, q0, p1)$;
$srt_stage(p1, d, q1, p2)$;
$srt_stage(p2, d, q2, r)$;
$Conversion(q, q0, q1, q2)$;
ENDMODULE

Figure 6: **ACV description of Module $srt_div_6_6$.**

Assume both srt_stage and $conversion$ modules are verified. During verification of module $srt_div_6_6$, when ACV evaluates the first srt_stage statement, vector $\vec{q0}$ has its word value $Q0$ and is treated as an input to module srt_stage to compute the value of vector $p1$. Therefore, the value of vector $\vec{p1}$ is $4 * (P - Q0 * D)$ and this becomes an input to the second srt_stage. ACV repeats the same procedure for the other srt_stage statements to compute the value of R which now depends on P, D, $Q0$, $Q1$ and $Q2$. It also computes the value of Q, which depends on $Q0$, $Q1$ and $Q2$, from module $Conversion$. The specification of this 6×6 SRT Radix-4 divider we verified is: $(-8D \leq 3P \leq 8D \wedge D \geq 2^5) \rightarrow (P * 2^6 == 4 * Q * D + R)$. The constraints , $-8D \leq 3P \leq 8D$ and $D \geq 2^5$, required for the first srt_stage, specify the input range constraints. Under these input constraints, the circuit performs the division, specified by the relation $P * 2^6 = 4 * Q * D + R$. Since $Q0$, $Q1$ and $Q2$ can be arbitrary values, we cannot verify the divider's output range constraint: $-8D \leq 3R \leq 8D$. It can be deduced manually from the initial condition and the input and output constraints of the srt_stage modules.

When the output of one module is connected to an input of another,

Sizes	16x16	32x32	64x64	128x128	256x256
CSA	4.68(sec)	20.08	78.55	351.18	1474.55
	0.83(MB)	1.19	2.31	6.34	21.41
Booth	2.37	8.18	27.47	128.87	535.18
	0.77	1.09	2.12	5.94	20.41
BitPair	1.90	5.76	15.43	69.68	288.70
	0.74	0.93	1.53	3.56	11.12
Seq	1.08	2.41	5.30	14.35	36.13
	0.70	0.76	0.96	1.41	2.75

Table 1: **Verification Results of Multipliers.** Results are shown in seconds and Mega Bytes.

ACV does not currently check that the constraints of outputs in the former module implies the constraints on the inputs in the latter. These constraints are specified in the VERIFY section. For example, the output constraint of the first srt_stage should imply the input constraints of the second srt_stage. Our future work will include the automation of this conformance checking.

4 Experimental Results

All of our results were executed on a Sun Sparc Station 10. Performance is expressed as the number of CPU seconds and the peak number of megabytes (MB) of memory required.

Table 1 shows the results of verifying a number of multiplier circuits with different word sizes. Observe that the computational requirements grow quadratically, caused by quadratical growth of the circuit size, except Design "seq" which is linear. The design labeled "CSA" is based on the logic design of ISCAS'85 benchmark C6288 which is a 16-bit version of the circuit. Our verification of this circuit requires only 4.68 seconds. Compared with other multipliers, the verification of CSA multiplier is slower, because the verification of a carry-save adder is slower than a carry-propagate adder. The designs labeled "Booth" and "BitPair" are based on the Booth and the modified Booth algorithms, respectively. Verifying the BitPair circuits takes less time than the Booth circuits, because it has only half the stages. Comparing these results with the results given in [1], we achieve around 3 to 4 times speedup, because we exploited the sharing in the module hierarchy. For a 64×64 multiplier, Hamaguchi *et al.*[5] reported 22,340 seconds of CPU time on Sun Sparc 10/51 machine, but ACV only requires 27.47 seconds. In [2], Chen *et al.* reported 508 seconds to verify a 64 bit multiplier on a HP 9000 workstation with 256MB, which is at least 2.5 times faster than Sun Sparc 10, using HDDs and extended SMV with a variant of Hamaguchi's method. In general, compared with approaches with Hamaguchi's backward substitution method, our approach achieves greater speedup for the larger circuits. Design "Seq" is an unrolled sequential multiplier obtained by defining a module corresponding to one cycle of operation and then instantiating this module multiple times. The performance of Design "Seq" is another example to demonstrate the advantage of sharing in our verification methodology. The complexity of verifying this multiplier is linear in the word size, since the same stage is repeated many times.

Table 2 shows the computing time and memory requirement of verifying divider and square root circuits for a variety of sizes. We have verified divider circuits based on a restoring method and the radix-4 SRT method. For the radix-4 SRT divider, the computing time grows quadratically, because we exploit the sharing property of the design and apply hierarchical verification as much as we can. Chen *et al.* [2] reported 194 seconds and 18.8MBytes to verify a 64-bit sequential divider using extended SMV. Our result is better than their's, because we use edge weights in our *BMD representation, whereas HDDs do not. For both restoring divide and square root, the computing time grows cubically in the word size. This complexity is caused by verifying the subtracter. While converting the vector of BDD functions into word-level *BMD function for the output

Sizes	16x16	32x32	64x64	128x128	256x256
srt-div	16.25(sec)	23.58	40.40	109.63	398.68
	1.16(MB)	1.47	2.19	4.47	10.47
r-div	5.53	26.02	153.13	1131.82	8927.18
	0.71	0.89	1.56	4.22	15.34
r-sqrt	8.35	54.85	320.60	2623.11	20991.35
	0.77	1.12	3.12	14.97	98.31

Table 2: **Verification Results of Dividers and Square Roots.** Results are shown in seconds and Mega Bytes.

of the subtracter, the intermediate *BMD size and operations grow cubically, although, the size of final *BMD function is linear.

5 Conclusions and Future Work

We have presented a system to automatically verify arithmetic circuits described in a hardware description language. We also illustrated techniques to overcome problems of verifying circuits such as a radix-4 SRT divider. These methodologies are also applicable to other circuits such as restoring division and restoring square root. The experimental results demonstrate that ACV can efficiently handle a variety of circuits with large word sizes. We can replicate the Intel Pentium division bug and successfully verify the circuit with the correct PD table. Currently, we are working on the verification of a square root circuit based on the radix-4 SRT algorithm. We believe it can be verified by ACV.

As mentioned within the paper, there are several aspects of the ACV program that should be improved. Rather than requiring the user to specify a variable ordering for each module at or above the transition layer, we would like ACV to automatically choose an initial ordering from the specification given in the VERIFY section, and then improve this ordering dynamically. We must also automate the checking of input and output constraints among modules, and be able to deduce output range constraints by composing the constraints for the sub-modules. Finally, we plan to improve our ACV system to accept circuits with explicit registers, rather than requiring users to supply unrolled versions of sequential circuits.

References

[1] R. E. Bryant, and Y.-A. Chen, "Verification of arithmetic circuits with binary moment diagrams," *32nd Design Automation Conference*, 1995.

[2] Y.-A. Chen, E. M. Clarke, P.-H. Ho, Y. Hoskote, T. Kam, M. Khaira, J. O'Leary and X. Zhao, "Verification of all circuits in a floating-point unit using word-level model checking" *Proc. of The International Conference on Formal Methods in Computer-Aided Design*, 1996.

[3] E. M. Clarke, M. Fujita, and X. Zhao, "Hybrid Decision Diagrams Overcoming the limitations of MTBDDs and BMDs" *Proc. of International Conference on CAD*, 1995, pp. 159-163.

[4] E. M. Clarke, M. Khaira, and X. Zhao, "Word level model checking - Avoiding the Pentium FDIV Error," *33nd Design Automation Conference*, 1996.

[5] K. Hamaguchi, A. Morita, and S. Yajima, "Efficient construction of binary moment diagrams for verifying arithmetic circuits," *Proc. of International Conference on CAD*, 1995, pp. 78-82.

[6] H. P. Sharangpani, M. L. Barton, "Statistical analysis of floating point flaw in the Pentium processor(1994)," Intel Technical Report, Nov. 30, 1994.

[7] D. Verkest, L. Claesen, and H. DeMan, "A proof of the nonrestoring division algorithm and its implementation on an ALU," *Formal Methods in System Design*, Vol. 4, No. 1, January, 1994, pp. 5-32.