Formal Verification of Word-Level Specifications

Stefan Höreth

Siemens Corporate R&D
D-81730 Munich / Germany
and
Darmstadt University of Technology
Dept. of Electrical & Computer Engineering
http://www.rs.e-technik.tu-darmstadt.de/ sth

Rolf Drechsler

Institute of Computer Science
Albert-Ludwigs-University
79110 Freiburg i. B., Germany

drechsle@informatik.uni-freiburg.de

Abstract

Formal verification has become one of the most important steps in circuit design. In this context the verification of high-level Hardware Description Languages *(HDLs), like VHDL, gets increasingly important.*

In this paper we present a complete set of datapath operations that can be formally verified based on Word-Level Decision Diagrams *(WLDDs). Our techniques allow a direct translation of HDL constructs to WLDDs. We present new algorithms for WLDDs for modulo operation and division. These operations turn out to be the core of our efficient verification procedure. Furthermore, we prove upper bounds on the representation size of WLDDs guaranteeing effectiveness of the algorithms. Our verification tool is totally automatic and experimental results are given to demonstrate the efficiency of our approach.*

1 Introduction

Nowadays modern circuit design can contain several million transistors. For this, also verification of such large designs becomes more and more difficult, since pure simulation can not guarantee the correct behavior and exhaustive simulation is too time consuming.

But many designs have very regular structures, like ALUs, that can be described easily on a higher level of abstraction. E.g. describing (and verifying) an integer multiplier on the bit-level is very difficult, while the verification becomes easy, when the outputs are grouped to a build a bit-string. Recently, several approaches to formal circuit verification have been proposed that make use of these regularities [1, 12, 3]. All these approaches have in common that they are based on *Word-Level Decision Diagrams* (WLDDs), i.e. graph based representations of functions (similar to BDDs [4]) that allow to represent functions with a Boolean range and an integer domain. Examples of WLDDs are e.g. EVBDDs [20], MTBDDs [9, 2], *BMDs [6], HDDs [10], and K*BMDs [14]. In the meantime WLDDs have been integrated in verifications tools [1, 8] and are also used for symbolic model checking [11, 7]. In [19] HDDs have been applied to verification of circuits at the register transfer level. WLDDs are a tool for bridging the gap between verification of high-level *Hardware Description Languages* (HDLs), like VHDL, and the netlists consisting of basic gates, like AND and OR. But so far for many HDL commands no effective way of translation into a WLDD is known.

In this paper we present a complete set of datapath operations that can be formally verified based on *Word-Level Decision Diagrams* (WLDDs). Our verification techniques allow to directly translate a HDL constructs to WLDDs. The key to this transformation are two new algorithms for modulo operation and division. Even though the operations have exponential worst case behavior we show by some experiments that these algorithms can handle functions with up to several hundred variables, while previously known algorithms fail for more than 16 bits. For some important functions often occurring in high-level descriptions we prove polynomial upper bounds on the representation size of the WLDD.

For each HDL operation we describe the main ideas and report some experiments. Finally, a case study on verifying a BCD-to-binary converter shows how the different components can be combined. We succeeded in automatically verifying this circuit, while other approaches, e.g. based on *BMDs only, fail.

The paper is structured as follows: In Section 2 WLDDs are introduced. In Section 3 arithmetic functions are described that often occur in high-level descriptions of circuits and their size is estimated. In Section 4 datapath operations are discussed. An experimental study is given in Section 5. Finally, the results are summarized.

2 Word-Level Decision Diagrams

In this section notations and definitions are given that are important for understanding the paper. We give a brief overview on *Decision Diagrams* (DDs). (For more details see [18, 13].)

All DDs are graph-based representations, where at each (non-terminal) node labeled with a variable x a decomposition of the function represented by this node into two subfunctions (the *low*-function and the *high*-function) is performed. In the following, we assume that the underlying graph is *ordered* and *reduced*, i.e. variables occur in the same order on all paths in the DD and functions represented by nodes of the graph are unique.

For bit-level DDs the following three decompositions have been considered:

$$\begin{array}{rclll} f & = & \overline{x}\, f_{low} \oplus x\, f_{high}, & & \text{Shannon } (S) \\ f & = & f_{low} \oplus x\, f_{high}, & & \text{positive Davio } (pD) \\ f & = & f_{low} \oplus \overline{x}\, f_{high}. & & \text{negative Davio } (nD) \end{array}$$

Function f is represented at node v, while f_{low} (f_{high}) denotes the function represented by the *low*-edge (*high*-edge) of v. $\oplus$ is the Boolean Exclusive OR operation. The recursion stops at terminal nodes labeled with 0 or 1. If at a node a decomposition of type S (D) is carried out this node is called a S-node (a D-node). If only decompositions of type S are applied the resulting DD is a BDD [4], while in OKFDDs [15] all three are allowed.

In this paper, we consider the same three decompositions for word-level functions, i.e. functions of the form $f : \mathbf{B}^n \rightarrow \mathbf{Z}$:

$$\begin{array}{rcl} f & = & (1-x) \cdot f_{low} + x \cdot f_{high}, \\ f & = & f_{low} + x \cdot f_{high}, \\ f & = & f_{low} + (1-x) \cdot f_{high}. \end{array}$$

The notation S, pD and nD is used analogously to the bit-level. x still denotes a Boolean variable, but the values of the functions are integer numbers and they are combined with the usual operations (addition, subtraction, and multiplication) in the ring **Z** of integers. Which decomposition is used, i.e. bit- or word-level, becomes clear from the context. To simplify the notation in the following and to avoid different cases for all decompositions, we use the notation

$$f = d_{low}(x) \cdot f_{low} + d_{high}(x) \cdot f_{high},$$

where $\cdot$ and $+$ is the multiplication and addition, respectively, over a domain D, $d_{low}, d_{high} : \mathbf{B} \rightarrow \mathbf{B}$, and $f_{low}, f_{high} : \mathbf{B}^{n-1} \rightarrow D$. It is easy to see that all decompositions above can be formulated using this generalized form, if d_{low}, d_{high}, f_{low}, and f_{high} are chosen appropriately.

Decomposition types are associated to the n Boolean variables $x_1, x_2, \ldots, x_n$ with the help of a *Decomposition Type List* (DTL) $d := (d_1, \ldots, d_n)$ where $d_i \in \{S, pD, nD\}$, i.e. d_i provides the decomposition type for variable x_i ($i = 1, \ldots, n$).

Based on the notations and definitions above we now introduce the functions represented by an edge in a DD. The edge function f_e is obtained from the function of the node through multiplication or addition of integer values.

For MTBDD [9, 2], BMD [6], HDD [10], EVBDD [20], *BMD [6] and K*BMD [14], the corresponding functions f_e are given in Table 1 (a, m are integer numbers).

Table 1. Functions represented by edges

graph type	edge function
MTBDD, BMD, HDD	$f_e = f$
EVBDD	$f_e = a + f$
*BMD	$f_e = m \cdot f$
K*BMD	$f_e = a + m \cdot f$

Edge-values are obtained from the node representation during the graph reduction phase. Note that all remaining DDs from Table 1 are obtained by further restricting the K*BMD reduction rules and DTL.

3 Representation Size of Arithmetic Functions

High-level circuit descriptions allow the use of buses. By this, Boolean variables are grouped, if they belong together. The big advantage of WLDDs is that they allow to directly make use of this grouping, while the direct correlation gets lost in bit-level DDs, like BDDs. Obviously, the smaller the representation is, the faster are the algorithms. This becomes even more important, if algorithms with exponential worst case behavior are used.

For this, we first consider arithmetic operations of functions that are defined over Boolean variables. Let $Var = \{x_1, \ldots, x_n\}$ be a set of Boolean variables:

1. $X = \sum_{i=0}^{n-1} x_i 2^i$
2. $X + Y$
3. $X \cdot Y$
4. $X^2, X^3, \ldots, X^c$ (c constant)
5. c^X (c constant)

While these functions are the basic operations for most others, they are studied in more detail in the following.

Depending on the WLDD-type the representation size largely varies (see below). The best results so far have been obtained for *BMDs: For $X, X+Y, X \cdot Y, c^X$ *BMDs have linear size [5]. These results directly transfer to K*BMDs. The situation becomes more complex, when functions of type X^c (c constant) are considered.

We first prove an upper bound for function X^c for BMDs. Obviously, the given bound holds for *BMDs and K*BMDs as well.

Theorem 1 *The BMD for function X^c has at most*

$$\sum_{i=0}^{c} \binom{n}{i}$$

nodes using the variable ordering $x_{n-1}, \ldots, x_0$.

Proof: Before we consider the BMD representation we start with some general considerations that will be used in the following:

$$(a+b)^d = \sum_{i=0}^{d} \binom{d}{i} a^{d-i} b^i \tag{1}$$

It now easily follows from Equation (1):

$$\begin{aligned}(a+b)^d - b^d &= (\sum_{i=0}^{d} \binom{d}{i} a^{d-i} b^i) - b^d \\ &= (\sum_{i=0}^{d-1} \binom{d}{i} a^{d-i} b^i) + b^d - b^d \\ &= \sum_{i=0}^{d-1} \binom{d}{i} a^{d-i} b^i \end{aligned} \tag{2}$$

Notice that the exponent of the polynomial decreases by one, i.e. from d to $d-1$.

We now make use of these equations, when we have a closer look at the influence of the BMD decomposition on polynomials:

$$f = f_{low} + x_i f_{high}$$

Here, f_{low} represents the function, if variable x_i is set to zero, i.e. $f_{low} = f_{x_i=0}$. f_{high} is given by $f_{x_i=1} - f_{x_i=0}$. If in the following we decompose the function starting from the highest coefficient in the polynomial towards the lower coefficients, i.e. $x_{n-1}, x_{n-2}, \ldots$, the function represented by the *low*-edge of node v computes the same polynomial as v, with the only difference that coefficient x_i has been set to zero.

The case for the *high*-edge is more difficult: We have to subtract the polynomials for the case of $x_i = 1$ and $x_i = 0$. But these polynomials differ only in a constant factor. Thus, Equation (2) can be applied and it directly follows that by each use of the *high*-edge the exponent of the polynomial represented by the corresponding node is decreased by one. After c *high*-edges the polynomial assumes a constant value, but this is represented as a terminal node in a BMD. Thus, for X^c we only have to count the number of paths from the root of the BMD that pass at most c *high*-edges. But as a straightforward computation shows this number is given by:

$$\sum_{i=0}^{c} \binom{n}{i}$$

□

As mentioned before the bounds given in the theorem directly transfer to *BMDs and K*BMDs. For *BMDs the bound for $c = 2$ can (asymptotically) not further be improved [5]:

Remark 1 *A *BMD for X^2 has a quadratic number of nodes.*

In the following two theorems we show that better upper bounds can be given for K*BMDs.

Theorem 2 *The K*BMD for function X^2 has at most $O(n)$ nodes using the variable ordering $x_{n-1}, \ldots, x_0$.*

Proof: We show the decomposition on the top level of the function. Then the generalization for all variables x_i becomes obvious. We start with function

$$f = (\sum_{i=0}^{n-1} x_i 2^i)^2.$$

This function is decomposed to the *low*-edge

$$f_{low} = (\sum_{i=0}^{n-2} x_i 2^i)^2$$

and the *high*-edge

$$f_{high} = 2^{2n-2} + 2^n \sum_{i=0}^{n-2} x_i 2^i.$$

As in the proof of the theorem above the exponent in the sum is reduced by one by the *high*-edge. Again, the case for the *low*-edge is trivial. If the function on the *high*-edge is decomposed again by a Davio decomposition, we obtain

$$2^{2n-2} + 2^n \sum_{i=0}^{n-3} x_i 2^i.$$

But due to the additive and multiplicative edge values this function becomes isomorphic to the *high*-successor of the *low*-edge, while the *high*-edge points to a constant value. All in all, the number of nodes per level is bounded by two. (A more detailed analysis shows that the exact number is given by $2 \cdot n - 2$). □

This result shows that K*BMDs are the only DD-type presented so far for hardware verification, that can represent all functions considered in [5] in linear size (see Table 2). The representation size becomes extremely important for WLDDs, since most operations have exponential worst case behavior. Thus, keeping the (final) representation small enables us to define more efficient algorithms.

Finally, we show that the result of Theorem 1 can also be improved for K*BMDs for $c = 3$.

Theorem 3 *The K*BMD for function X^3 has at most $O(n^2)$ nodes using the variable ordering $x_{n-1}, \ldots, x_0$.*

Proof: A detailed analysis similar to the one of the proof above shows that (starting from the third level) per level one additional node is created. Thus, the total number of nodes becomes quadratic in the number of variables. (The exact number is given by $(n^2 + n - 4)/2$). □

All in all, it turns out there exist WLDD-types that can efficiently represent arithmetic operations in polynomial size (by polynomials of low degree), while other types fail.

Table 2. Representation sizes of different DD-types for arithmetic functions

DD-type	X	$X+Y$	$X \cdot Y$	X^2	c^X
MTBDD	exp	exp	exp	exp	exp
EVBDD	lin	lin	exp	exp	exp
BMD, HDD	lin	lin	quad	quad	exp
*BMD	lin	lin	lin	quad	lin
K*BMD	lin	lin	lin	lin	lin

4 Word-Level Verification

In this section we define a set of datapath operations that allow to effectively verify high-level HDLs, like VHDL. For this, first two operations are introduced, i.e. modulo operation and division. Recently, it has been proved that none of the "usual" WLDD-types can represent the division function efficiently [21]. Nevertheless, our algorithms for these closely related operators work very well in practice. (All experiments in this section have been carried out on a SUN UltraSPARC-170 workstation with 256 MByte of main memory.)

4.1 Modulo Operation

Modulo arithmetic based on powers of two is frequently used in specifications of datapaths. But as described above, division (and modulo) is a "hard" problem for WLDDs. A straightforward approach to compute modulo would be to recursively apply Shannon decompositions. But a limitation of this approach when using WLDDs is that the range of functions often becomes prohibitively large.

In the sequel, we present an algorithm for modulo arithmetic, that often avoids explicit enumeration of function values. We make use of the two properties of modulo arithmetic:

$$\begin{aligned}(a+b)\%n &= (a\%n+b)\%n \\ &= (a\%n+b\%n)\%n \\ (a\cdot b)\%n &= (a\%n\cdot b)\%n \\ &= (a\%n\cdot b\%n)\%n.\end{aligned}$$

Here $\%$ denotes the modulo operation, $a, b \in \mathbf{Z}$, and $n \in \mathbf{N}$.

The algorithm consists of two steps (only the main idea is given in the following, due to page limitation; for more details see [17]):

1. Terminal cases are checked based on a "conservative" estimate for function ranges. We make use of the algorithm for range estimation as described in [10].

2. If step 1. fails, an estimate $f\%_{\approx}g$ is computed, by carrying out the modulo operation on f_{low} and f_{high}.

   If $g : \mathbf{B}^n \to \mathbf{N}$ is independent of variable x, it holds:

$$\begin{aligned}f\%g &= (d_{low}(x)\cdot f_{low} + d_{high}(x)\cdot f_{high})\%g \\ &= (d_{low}(x)\cdot(f_{low}\%g) \\ &\qquad + d_{high}(x)\cdot(f_{high}\%g))\%g \\ &= (f\%_{\approx}g)\%g \\ &= (d_{low}(x)\cdot(f_{low}\%_{\approx}g) \\ &\qquad + d_{high}(x)\cdot(f_{high}\%_{\approx}g))\%g\end{aligned}$$

   Then again step 1. is applied to the WLDD for $f\%_{\approx}g$ until some terminal cases are reached.

   If g depends on x, both algorithms, i.e. exact computation based on Shannon decomposition and estimate are applied recursively.

As an important special case this algorithm also includes the modulo operation with a constant function g. Then for computing $f\%_{\approx}g$ only the WLDD for f has to be traversed, and the operation has to be applied to the terminal nodes. Afterwards, range estimation on the simplified WLDD frequently leads to an early termination.

Remark 2 *For WLDDs using additive and multiplicative edge values for constant functions $g > 0$ we proceed as follows:*

$$(a+m\cdot f_v)\%g = (a\%g+(m\%g)\cdot f_v)\%g$$

Then the modulo operation only has to be computed for the simplified function $a\%g + (m\%g)\cdot f_v$.

Experiment We consider modulo addition based on WLDDs:

$$(\sum_{i=0}^{n-1} 2^i x_i + \sum_{i=0}^{n-1} 2^i y_i)\%2^n \tag{3}$$

We represent the formula by a K*BMD with only pD decomposition using an interleaved variable ordering.

The results of our approach in comparison to the conventional approach based on Shannon expansion for varying bit-length are given in Table 3. Even though the size of the output function grows only linear with the bit-length, the straightforward approach fails for more than 16 bits, while our algorithm can handle the function with 512 bits (and 1024 variables) in less than 300 CPU seconds.

4.2 Division

Based on the modulo operation described above, we now give an algorithm for computing the division on WLDDs.

The basic idea of the algorithm is to first subtract the remainder of the division from the dividend and then to compute the result:

$$f/g = (f - f\%g)/g.$$

If g is independent of variable x, it holds:

$$\begin{aligned}f/g &= (f-f\%g)/g = f'/g \\ &= (d_{low}(x)\cdot f'_{low} + d_{high}(x)\cdot f'_{high})/g \\ &= (d_{low}(x)\cdot(f'_{low}/g) \\ &\qquad + d_{high}(x)\cdot(f'_{high}/g)).\end{aligned} \tag{4}$$

Table 3. Modulo operation

bit-length	4	8	16	32	64	128	256	512
time (Shannon) [s]	0,1	0,2	18,5	>1h	>1h	>1h	>1h	>1h
time (mod) [s]	0,1	0,1	0,1	0,2	0,6	2,7	21,4	275,9
size [nodes]	16	36	76	156	316	636	1276	2556
max. size [nodes]	20	40	80	160	320	640	1280	2560

Table 4. Division with non-constant divisor

bit-length n	4	8	16	32	64	128
divisor [nodes]	6	14	30	62	126	254
max. size [nodes]	17	40	88	184	376	760
time [s]	0,1	0,1	0,1	0,4	3,9	170,5

Otherwise the division is computed by carrying out a Shannon expansion for its arguments $(f - f\%g)$ and g, respectively. (Again, the algorithm can be simplified for constant functions $g > 0$ and for WLDDs making use of edge-values.)

In some cases division can also be computed efficiently when the divisor is not constant. This is often the case, if dividend and divisor are monotonous and if they are defined over the same set of variables.

Experiment Consider the division

$$\frac{a+1}{a^2+2a+1}, \quad a = \sum_{i=0}^{n-1} 2^i a_i.$$

The expressions $a+1$ and a^2+2a+1 are given as K*BMDs consisting of pD-nodes only. For $a = 0$ the result becomes 1. In all other cases it becomes 0. The K*BMD grows linearly with the bit-length.

This is "obvious", but hard to handle using DDs. E.g. BDDs fail, since they can not represent multiplication efficiently. Applying the standard methods (see e.g. [5]) all input combinations have to be considered resulting in an exponential runtime.

Using the algorithm described above also large bit-length can be handled efficiently (see Table 4). A prerequisite for this are the efficient representations of e.g. a^2 as proven in Section 3.

4.3 Datapath Operations

Based on the algorithm described so far in combination with the results presented in [16] we can now efficiently describe a large set of datapath operations for HDLs, like VHDL (see Table 5). `a, a0, a1` denote bit-vectors of length n, that are given by integer encodings $a, a0, a1$. `b` is a single bit represented by the Boolean function b. `n, k, l` (n, k, l) are natural numbers.

```
equ(inc(ac,2*n),
 cat(inc(selslice(ac,0,n-1),n),
  adc(selslice(ac,n,2*n-1),0,
   equ(inc(selslice(ac,0,n-1),n),0)),n))
```

Figure 1. Example of datapath operation

Notice that the operations often combine Boolean and integer expressions. This is taken into account by using Boolean and integer graph types. In the implementation of the hybrid DD package from [16], e.g. the parity function `odd(a)` uses a WLDD to represent the integer function a, while the result is represented by a Boolean graph type, i.e. an OKFDD or a BDDs.

Experiment Consider the datapath operation in Figure 1. It will be checked whether incrementing register `ac` (of bit-length $2n$) can be done by splitting it into two words of length n and then performing the operation accordingly.

The implementation given in Figure 1 is **faulty**, since a carry might be generated during addition `adc`. In our experiment all word-level operations are carried out on K*BMDs and for all Boolean expressions BDDs are used. The BDD for the first occurrence of function `equ` represents the complete set of possible values of register `ac`, for which the operation is implemented correctly. (It is easy to see that the BDD only needs $2n + 1$ nodes.)

In Table 6 again the runtimes and the maximum graph sizes during the computation are given. The main problem in this case is the computation of the division and modulo operation in functions `selslice` and `inc`, respectively. Notice that the addition `adc` is again a hybrid operation, i.e. between K*BMDs and BDDs. Even though, most of the word-level operations have exponential worst case behavior it turns out that in most practically relevant cases

Table 5. HDL and their implementation by word-level operations

HDL	word-level operation	interpretation
`a`	$\sum_{i=0}^{n-1} 2^i \cdot a_i$	integer encoding
`adc(a0, a1, b)`	a0+a1+b	addition with carry
`cat(a0, a1, n)`	$2^n \cdot a1 + a0$	concatenation
`fae(b, n)`	$\sum_{i=0}^{n-1} b \cdot 2^i$	fanout
`selel(a, k)`	$odd(a/2^k)$	bit selection
`selslice(a, k, l)`	$(a\%2^{l+1})/2^k$	bit-slice
`inc(a, n)`	$(a+1)\%2^n$	increment modulo
`dec(a, n)`	$(a-1)\%2^n$	decrement modulo
`rsh(a, b, n)`	$(a\%2^n)/2 + 2^{n-1} \cdot b$	shift right
`lsh(a, b, n)`	$((a \cdot 2) + b)\%2^n$	shift left
`rol(a, n)`	$(a \cdot 2)\%2^n + (a\%2^n)/2^{n-1}$	rotate left
`ror(a, n)`	$(a\%2^n)/2 + 2^{n-1} \cdot odd(a)$	rotate right
`equ(a0,a1)`	$equ(a0, a1)$	equivalence
`gt(a0,a1)`	$gt(a0, a1)$	greater than
`IF b THEN a0; ELSE a1; FI`	$b \cdot a0 + (1-b) \cdot a1$	conditional

Table 6. Verification of datapath operation

bit-length $2n$	16	32	64	128	256	512
time [s]	0,3	0,6	1,3	4,3	21,4	161,2
max. size [nodes]	54	155	315	681	1387	2811

the runtimes are very small. (If a correct implementation is considered the runtimes of our algorithm are in the same range.)

5 A Case Study

Finally, we describe the complete automatic formal verification of a 10-decade BCD-to-binary converter. (Minor details are left out due to page limitation.) Following the *Texas Instrument TTL Data Book for Design Engineers* the specification is given by:

> The BCD-to-binary function of the SN54184 and SN74184 is analogous to the algorithm:
>
> a. Shift BCD number right one bit and examine each decade. Subtract three from each 4-bit decade containing a binary value greater than seven.
>
> b. Shift right, examine, and correct each shift until all converted decades contain zeros.

One possible formulation of this algorithm in a more formal way is given in Figure 2. We compare the HDL description to an implementation composed of subcircuits of type SN74184. As can be seen the HDL description makes use of several operations introduced in the previous sections, like addition, multiplication, greater than.

```
i:=0; d[n]:=0;
DO
 b[i] := odd(d[0]);
 FOR j:=0 TO n-1 DO
  d[j] := d[j]/2 + 2ˆ3 * odd(d[j+1]);
 FOR j:=0 TO n-1 DO
  d[j] := d[j] - 3 * gt(d[j], 7);
 i := i+1;
UNTIL (d[0]=0 & ... & d[n-1]=0);
```

Figure 2. Algorithmic specification

For all Boolean functions we used BDDs and all word-level operations are carried out using K*BMDs. The decomposition types and the variable ordering are not predetermined: they are dynamically found using DTL-sifting [18].

On a SUN UltraSPARC-170 workstation 30 MByte of main memory were needed. For the transformation of the specification to WLDDs about 11 CPU minutes were needed. Then the BDD for the implementation is constructed. The circuit consists of 82 TTL elements (corresponding to about 5000 two-input gates). The BDDs for the outputs are constructed in less then 1 CPU minute. Furthermore, also *Don't Cares* are considered, i.e. only “valid” input combinations are used.

All in all, the verification could be completed (including computation of specification and implementation) in less than 18 CPU minutes using 97 MByte of main memory. 60% of the runtime was used for dynamic minimization based on DTL-sifting and the maximal number of nodes during the run was 1.5 million.

Finally notice that in contrast the verification of the specification against the circuit using *BMDs only failed. This further underlines the importance to hybrid structures in verification.

6 Conclusions

In this paper we presented a complete set of datapath operations that can be formally verified based on *Word-Level Decision Diagrams*. Our techniques allow a direct translation of HDL constructs to WLDDs. The sizes of WLDDs for important arithmetic functions have been estimated and we have studied manipulation algorithms for WLDDs for modulo operation and division. Based on these core operations, we have shown by several experiments the feasibility of our approach.

In a case study we showed how a specification and its implementation could be automatically verified using formal techniques. Alternative approaches based e.g. on *BMDs could not complete the verification within several hours, while the whole process took less than 18 CPU minutes using our techniques.

References

[1] L. Arditi. *BMDs can delay the use of theorem proving for verifying arithmetic assembly instructions. In *FMCAD*, pages 34–48, 1996.

[2] R.I. Bahar, E.A. Frohm, C.M. Gaona, G.D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebraic decision diagrams and their application. In *Int'l Conf. on CAD*, pages 188–191, 1993.

[3] C.W. Barrett, D.L. Dill, and J.R. Levitt. A decision procedure for bit-vector arithmetic. In *Design Automation Conf.*, June 1998.

[4] R.E. Bryant. Graph - based algorithms for Boolean function manipulation. *IEEE Trans. on Comp.*, 35(8):677–691, 1986.

[5] R.E. Bryant and Y.-A. Chen. Verification of arithmetic functions with binary moment diagrams. Technical report, CMU-CS-94-160, 1994.

[6] R.E. Bryant and Y.-A. Chen. Verification of arithmetic functions with binary moment diagrams. In *Design Automation Conf.*, pages 535–541, 1995.

[7] Y. Chen, E. Clarke, P. 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. In *FMCAD*, pages 389–403, 1996.

[8] Y.-A. Chen and R.E. Bryant. ACV: an arithmetic circuit verifier. In *Int'l Conf. on CAD*, pages 361–365, 1996.

[9] E. Clarke, M. Fujita, P. McGeer, K.L. McMillan, J. Yang, and X. Zhao. Multi terminal binary decision diagrams: An efficient data structure for matrix representation. In *Int'l Workshop on Logic Synth.*, pages P6a:1–15, 1993.

[10] E.M. Clarke, M. Fujita, and X. Zhao. Hybrid decision diagrams - overcoming the limitations of MTBDDs and BMDs. In *Int'l Conf. on CAD*, pages 159–163, 1995.

[11] E.M. Clarke and X. Zhao. Word level symbolic model checking - a new approach for verifying arithmetic circuits. Technical Report CMU-CS-95-161, 1995.

[12] D. Cyrluk, O. Möller, and H. Rueß. *An Efficient Decision Procedure for the Theory of Fixed-Sized Bitvectors*, volume 1254 of *LNCS*. Computer Aided Verification, 1997.

[13] R. Drechsler and B. Becker. *Binary Decision Diagrams - Theory and Implementation*. Kluwer Academic Publishers, 1998.

[14] R. Drechsler, B. Becker, and S. Ruppertz. The K*BMD: A verification data structure. *IEEE Design & Test of Comp.*, pages 51–59, 1997.

[15] R. Drechsler, A. Sarabi, M. Theobald, B. Becker, and M.A. Perkowski. Efficient representation and manipulation of switching functions based on ordered Kronecker functional decision diagrams. In *Design Automation Conf.*, pages 415–419, 1994.

[16] S. Höreth. Implementation of a multiple-domain decision diagram package. In *CHARME*, Chapman & Hall, pages 185–202, 1997.

[17] S. Höreth. *Effiziente Konstruktion und Manipulation von binären Entscheidungsgraphen*. Ph.D. thesis at Technische Universität, Darmstadt, 1998.

[18] S. Höreth and R. Drechsler. Dynamic minimization of word-level decision diagrams. In *Design, Automation and Test Europe*, pages 612–617, 1998.

[19] G. Kamhi, O. Weissberg, and L. Fix. *Automatic Datapath Extraction for Efficient Usage of HDD*, volume 1254 of *LNCS*. Computer Aided Verification, 1997.

[20] Y.-T. Lai and S. Sastry. Edge-valued binary decision diagrams for multi-level hierarchical verification. In *Design Automation Conf.*, pages 608–613, 1992.

[21] C. Scholl, B. Becker, and T.M. Weis. Word-level decision diagrams, WLCDs and division. In *Int'l Conf. on CAD*, 1998.