Automatic Datapath Extraction for Efficient Usage of HDD

Gila Kamhi, Osnat Weissberg, Limor Fix
Ziv Binyamini, Ze'ev Shtadler
Design Technology
Intel, Haifa, Israel
gkamhi@iil.intel.com

Abstract

Hybrid Decision Diagrams (HDD) have been proven in Intel to be an important enabler for the formal verification of datapath intensive circuits and in particular the verification of arithmetic units. However, extensive user interaction with the formal verification tool was required in order to use the HDD technology efficiently. The user had to analyze the circuit and its specification and manually partition the signals and operations into control and datapath.

In this paper, we will demonstrate how we have made use of the automatic datapath extraction techniques widely used in the synthesis world in order to efficiently integrate HDDs to an SMV-based formal verification system. The intention of this paper is to illustrate how existing technology can help improve the usability and productivity of the formal verification process and enable efficient integration of new technology, in our case HDDs.

The system described in this paper, *Prover*, statically analyzes the model to be verified and partitions the representation of the logic to HDDs and Binary Decision Diagrams (BDDs). Moreover, the partitioning algorithm decides which vector operations will be represented more efficiently as word-level (i.e. using HDD) versus bit-level (i.e using BDD).

The new methodology of integrating HDD into the formal verification process increases the productivity of the verification process. At the same time, experiments with *Prover* show that verification is (both computation and memory usage wise) as efficient as the previously known manual method.

1 Introduction

The Binary Decision Diagram (BDD) technology is a key enabler to several VLSI-CAD solutions. However, since BDDs are inefficient in dealing with datapath intensive circuits, the synthesis and formal verification of these circuits still challenge the VLSI-CAD community.

BDD-based approaches cannot handle, particularly, the verification of arithmetic functions such as multiplication and division, mainly because the BDD representations for these functions grow exponentially relative to the bit size. Bryant and Chen [1] have addressed the limitation of BDDs by introducing Binary Moment Diagrams (BMDs). BMDs and their extensions, e.g.*BMDs, enable compact word-level representation of arithmetic functions including multipliers. Although BMDs can represent datapath portion of the logic efficiently, the control portion of the logic can still be represented more efficiently using BDD or MTBDD (multi-terminal BDD).

Clarke and Zhao [2] have introduced Hybrid Decision Diagrams (HDDs) which represent logic as a combination of MTBDDs and BMDs. Previous applications of

HDDs in Intel's tools [3] required the user to manually provide the information on how to partition HDDs into MTBDDs and BMDs. In most cases, a design and a verification expert had to manually extract the datapath. This approach was time-consuming, unproductive and often involved a third party's understanding of the design. Therefore, we believe that automatic identification and partitioning of datapath and control logic is essential for the efficient embedding of HDDs in the industrial formal verification tools.

State-of-the-art architectural and datapath synthesis systems automatically identify the Hardware Design Language (HDL) statements describing datapath and replace the corresponding logic with optimized predefined macros that fit the description. The encapsulation of datapath functions in the synthesis tools is mainly done to optimize the datapath portion of the logic taking into consideration timing and area constraints. In this paper, we demonstrate how we have made use of the automatic datapath extraction techniques widely used in the synthesis world to efficiently integrate HDDs to an SMV-based [16] formal verification tool. Our intention is to illustrate how existing technology can help improve the usability and productivity of the formal verification process and enable efficient integration of new technology, in our case HDDs.

This paper presents two capabilities of Intel's formal verification tool, *Prover* : the automatic encapsulation of datapath and HDD partitioning. The HDD partitioning algorithm decides which datapath bits will be encoded as BMDs and which as BDDs. This decision is not trivial since some of the datapath bits are still more efficiently represented as BDDs. *Prover* accepts as input a high-level hardware specification language and generates an hierarchical intermediate netlist. The intermediate representation contains an additional layer of hierarchy for datapath operations. The partitioning algorithm of *Prover* when fed the intermediate netlist builds and analyzes a two-level Data Flow Graph (DFG) of the circuit. One level of the DFG represents the bit-level connectivity of the model and the second level represents the word-level connectivity. The algorithm decides which components of the circuit will be more efficiently represented using BMD and which using MTBDDs and BDDS and generates word-level Symbolic Model Language (SML) for both the model and its specification. Using *Prover* we have successfully verified circuits for division and square root computation that are based on SRT algorithm used by Intel's *Pentium*® microprocessor. The novelty of *Prover* is that it reduces the required user expertise and intervention compared to other formal verification systems.

This paper is organized as follows: Section 2 contains a brief overview of Intel's formal verification tool, *Prover*. Section 3 describes how the datapath statements in the model and specification languages are automatically extracted. Section 4 discusses the efficiency and deficiency of HDDs and especially BMDs to represent logic. In Section 5, we describe the partitioning algorithm used to partition the logic into BDDS and HDDs and HDDs into MTBDDs and BMDs. Section 6 presents results obtained using *Prover* to verify complex real-life arithmetic properties including the floating-point division (FDIV) in *Pentium*® processor. We compare our results to the manual datapath extraction experiments.

2 The System

Prover, Intel's formal verification tool, is based on two major components: the formal specification language and the symbolic model checking engine. Our specification language, called Formal Specification Language (FSL), is a linear-time temporal language that allows the specification of a complex behavior as a series of timed expressions. The model building stage of *Prover* synthesizes the register-transfer-level (RTL) model and the specification to a gate-level intermediate format [8]. *Prover* performs property-specific model extraction to prune the parts of the model that are irrelevant to the property and automatically generates extended SML as input to its SMV-based [16] model checking core. The temporal properties expressed in FSL are translated internally to SML checkers and simple CTL formulas.

Prover's model checking core implements the word-level model-checking algorithm of Clarke and Zhou [2]. However, we allow word-level expressions both in the model description and in the specification. As a result, the intermediate functions needed for the construction of the transition relation and the specification are represented using HDD while the final representation of the transition relation and the specification are in BDDs.

Embedding HDDs[1] in *Prover*'s verification engine [3] made it possible to handle circuits containing both control logic and wide datapaths. Recently, we have automated the use of HDDs in *Prover* by integrating an automatic datapath extraction mechanism to the model building stage of the tool. The synthesizer automatically encapsulates arithmetic functions in the model and the specification. A partitioning algorithm decides which signals and which operations will be more efficiently implemented using BDDs and which using HDDs. Moreover, within the HDD representation, the algorithm decides on the partitioning between the MTBDDs and the BMDs.

3 Encapsulation of Datapath Operators

The logic of a hardware system can be classified as (pure) control, (pure) datapath, or (mixed) control/datapath. The control portion of a hardware system consists of a set of interacting FSM's which depending on data values and states, produce a set of control signals for the datapath. The datapath consists of functions and registers which, based on control signals, operate on data. The data often consists of

1. The reader is referred to Section 4 and [1, 2, 3, 5, 12] for detailed description of BDDs, MTBDDs, BMDs and HDDs.

integers. The memory acts as a container for values, and communicates with datapath.

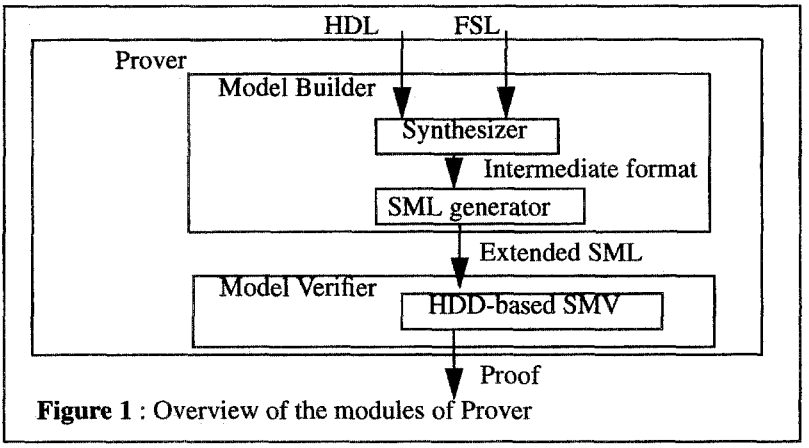

Figure 1 : Overview of the modules of Prover

The BDD-based approaches fail to represent efficiently wide datapath operators. BMDs and extensions to BMDs, e.g. *BMDs, enable very compact word-level representation of arithmetic functions including multipliers. The symbolic model checking engine of *Prover* is HDD-based. The model under test can be represented as a combination of MTBDDs and BMDs. The motivation is to represent control logic as BDD/MTBDDs and datapath logic as BMDs. The automatic extraction of datapath [7] and control logic is essential for the success of the hybrid approach taken in HDD where MTBDDs and BMDs coexist.

In the model-building stage, *Prover* encapsulates the HDL and FSL operators related with datapath and maps the isolated operators to generic predefined subcircuits. The predefined subcircuits are generic in the sense that all the subcircuits that describe a plus operation of any width will have only two inputs and one output. The bit-width of the input and output signals varies depending on the size of the operation. In other words, the generic parameter of the predefined subcircuits is the bit-width of the operation.

Prover, currently automatically encapsulates the arithmetic operations, e.g. multiplication. In addition to the arithmetic operators, the relational operations, i.e. comparison, equality and inequality operations, and other vector operations, e.g. concatenation, are automatically identified. Furthermore, the automatic encapsulation mechanism in *Prover* extracts memories, and control logic, e.g. multiplexers, that controls the datapath. For example, "if", and "case" expressions in HDL and FSL are preserved by mapping them to predefined generic multiplexer templates of the respective width.

The encapsulated operators generate an extra hierarchy in the intermediate netlist generated by the synthesizer of the model builder (See Figure 1). Information on the functionality (e.g. addition, subtraction) of these implicit tool-generated subcircuits is stored as attributes. The SML netlist generator of *Prover*, when fed the intermediate netlist, decides which isolated operators can be more efficiently

represented by BMDs and generates an extended SML netlist which is fed to the verification core. The partitioning algorithm used in the SML generator is explained in more detail in Section 5. The SML netlist contains information on which signals are to be represented as BMDs by means of the new constructs that have been added to the language. The verification core of *Prover* identifies the datapath circuitry in the SML netlist and represents the relevant portions of the logic as BMDs.

The benefits of a datapath extraction mechanism are numerous. The idea of automatically extracting the datapath operators and mapping them to predefined generic modules can be extended. For example, it would be quite reasonable to create a library of optimized standard arithmetic functions (addition, multiplication, relational operators) and re-use the size-wise optimized (BDD or extensions to BDD) modules. The portion of the logic (e.g. memory) which cannot be represented efficiently neither by BDDs nor by BMDs can be treated as a black box.

4 Efficiencies and Deficiencies of BMDs versus BDDs

Ordered binary decision diagrams (BDDs) [5, 12] are a canonical efficient representation for Boolean formulas. A BDD representing a formula is a directed acyclic graph (DAG) with a unique order on the occurrence of variables from root to leaf. Multi-terminal BDDs (MTBDDs) have a similar structure; however instead of having boolean leaves they have integer leaves. Both BDD and MTBDD representation of certain arithmetic functions such as multiplier requires exponential resources. Bryant and Chen [1] have shown that BMD can represent compactly certain arithmetic functions which have exponential MTBDD representation.

HDDs (BMDs and MTBDDs) can represent functions that map boolean bit vectors to integers; whereas BDDs can represent only boolean functions. An array or bit vector of state variables represented as a HDD is referred to as a word [1, 2, 3]. The value of the word is the value of the integer represented by the bit vector. BMD representation is more compact for some arithmetic functions which have exponential size if represented as MTBDDs. In general, BMDs and BDDs have comparable size for boolean functions, however, our experience shows that BMDs are less compact in these cases. Similar results were reported for some boolean functions in [15].

The method of building BMD from an arithmetic expression, e.g. multiplication, can blow up exponentially in the building phase of a single bit [6]. Therefore, two verification methods for multipliers based on Multiplicative Binary Moment Diagrams (*BMDs) have been introduced. Bryant has proposed a hierarchical approach in which the building blocks of the multiplier being verified, for instance a carry-save adder, are checked against their *word-level* specification. Once the individual sub-blocks have been verified, they are composed to yield the function of the entire block. The shortcoming of this approach is that it assumes that the specification has the same hierarchy as the implementation, and that the

corresponding blocks in the specification and the implementation are equivalent which is not always the case. Hamaguchi [9] proposed a method to overcome this limitation which is a non-hierarchical variation of Bryant's scheme which compute BMD from outputs to inputs instead of inputs to outputs and verify successfully wide-size, e.g. 64-bit, multipliers. However, if there are errors in the circuits, BMD can easily blow up and the verification program may not terminate, since these circuits represent different logic functions from multiplier which can have exponential sizes of BMD. In [10, 11], BDD-based techniques that overcome this limitation of the BMD-based methods are presented.

In *Prover,* the decision on which HDL and FSL operations are to be translated to *word-level* SML operators and consequently to be represented as HDD by the verification core is based on mainly the efficiency of HDD in comparison with BDD to represent these operations. As mentioned above, the BMD representation of single bit/bits of an arithmetic expression may be huge, even when the BMD representation of the whole expression is very compact. Therefore, expressions involving relations among single bits of arithmetic sub-expressions are not good candidates to be represented as BMD.

Moreover, the relational operations, e.g., inequalities, equalities, can be represented more efficiently by BDD than BMD. Clarke and Zhao [2] present an algorithm that can substantially reduce the cost of computing arithmetic relations between *word-level* functions. However, our experience shows that the BDD representation of the relational operations is more efficient than the BMD representation.

In Section 5, we will present the algorithm currently used by *Prover* in order to decide which portion of the logic will be represented as BDD and which portion of the logic will be represented as BMD.

5 Automatic Partitioning

The algorithm for partitioning the logic into BDD (*bit-level*) and BMD (*word-level*) representations within the HDD is mainly based on the efficiency of HDD (BMD/ MTBDD) compared to BDD for representing datapath operations.

The partitioning algorithm in the first stage decides which vectors will be represented as words and which operations will be represented as *word-level* operators. These vectors and operations will be represented as HDD. The rest of the logic will be represented as BDD. The BMD/MTBDD partitioning within HDD is performed by marking the datapath variables in HDD as data variables; thus, causing the datapath operations to be represented as BMD.

Based on the encapsulation of the datapath operators described in Section 3, a two-level Data Flow Graph (DFG) of the model is built. One level of the DFG represents the *bit-level* connectivity of the entire model. That is, given a signal bit *sig* one can extract from the DFG information on the signals in the *fan-in* and *fan-out* of *sig*. The second level of the DFG represents the *word-level* connectivity of the model. That is, given a word (vector) *vec* one can extract from the DFG the *fan-in* and *fan-out* information on *vec*. The *word-level* connectivity graph also contains

information on the *fan-in* and *fan-out* of each vector operation: the operands are the *fan-in* and the result is the *fan-out*.

The *bit-level* and *word-level* connectivity graphs are used to detect the case where a single bit is extracted from the result of a vector operation. In other words, if we compute an intermediate *word-level* expression and represent it using BMD it will be inefficient to extract from it the BDD/BMD representation of one of its bits. For example, the intermediate expression *temp* below

```
temp := word1 + word2;
sig := temp[3];
```

will be represented as BDD instead of BMD since we need to extract bit 3 out of *temp*. On the other hand, we may choose to represent an operation using BMD even in the case this operation is more efficiently represented using BDD. The reason for such a decision lays again in the context of the operation.

The BDD representation of some bit vector operations, e.g. concatenation, is more efficient than the BMD representation. Bit vector operators and relational operators, e.g. <=, >=, are mapped to their corresponding *word-level* SML operators if and only if the operators in the *fan-in* cone of these operators are arithmetic operators (+,-,*). For example, the concatenation operation (&) in the assignment below has to be represented at *word-level*, in order to represent the multiplication operation (*) at *word-level*.

```
c[3n+2:0] := (a[n:0] * b[n:0]) & d[n:0];
```

In other words, to represent the multiplication expression `(a[n:0] * b[n:0])` as a BMD, we choose to perform the concatenation operation (&) on BMDs even though it is more efficient to perform it on BDDs.

The partitioning algorithm first analyzes the DFG and marks every vector and every datapath operator as either *bit-level* or *word-level*. After the preliminary marking of all vectors and operations as *bit-level* or *word-level*, the DFG is scanned again and a final decision on which vectors to represent as words (HDDs) and which as BDDs is made. The outcome of the algorithm is a list of vectors and operations that need to be represented at *word-level* and a list of signal bits that need to be specified as data variables.

The partitioning algorithm is depicted below. In summary, initially (Step 1) the algorithm marks as many vectors and operators as possible as *word-level*. Then (Step 2), the algorithm reclassifies the word-level vectors and operators by propagating backwards the need to represent some vectors at *bit-level* instead of *word-level* and the need to represent datapath operators at *bit-level* instead of *word-level*. Finally (Step 3), marks all bit-vector and relational operators as *bit-level*, if it is not necessary for them to be *word-level*.

The Partitioning Algorithm:

Step 1 (Initial marking):

Step 1.1 (Mark datapath operators):
For each datapath operator ***op***,
If no single bit of fan-out vectors of ***op*** is referenced (read) by other expressions, then mark ***op*** as ***bit-level***.
Otherwise, mark ***op*** as ***bit-level***.

Step 1.2 (Mark datapath vectors - words):
For each vector ***vec***,
If the fan-out of every bit of ***vec*** is equal to the fan-out of ***vec***, then mark ***vec*** as ***word-level***.
Otherwise, mark ***vec*** as ***bit-level***.

Step 2 (Propagating backward):

For each datapath operator ***op*** marked as ***bit-level***,
For each datapath vector ***vec*** in the fan-in of ***op***,
Mark ***vec*** as ***bit-level***
For each datapath operator ***op'*** in the fan-in of ***vec***,
Mark ***op'*** as ***bit-level***

Step 3 (If necessary, mark bit-vector and relational operations as ***bit-level*** - correction):

For each ***word-level*** bit-vector and relational operation ***op***
Recursively mark ***op*** as bit-level if none of its fan-in vectors is ***word-level***
If the fan-out vector ***vec*** of ***op*** is ***word-level***, then mark ***vec*** as ***bit-level*** and recursively apply step 3 on all operations ***op'*** which are in the fan-out of ***vec***

6 Experimental Results

Intel's new generation microprocessors are being massively verified using the automatic datapath extraction algorithm integrated into *Prover*. Below is a simple example that demonstrates *Prover*'s capability to verify complex arithmetic specifications.

The division and square-root algorithms similar to the ones in *Pentium*® processor that we have re-verified are described in several published papers [2, 3, 4, 13]. For

the completeness and readability of this paper we include below the description of the division algorithm adopted from [3]. The square-root operation shares common datapath with the division operation; thus, we only illustrate here the capability of our system to verify complex arithmetic algorithms by concentrating mainly on the division algorithm. The number of state variables after the property-specific extraction in *Prover* is in the order of 300-400. During the re-verification process we did not exceed the memory requirements to verify these properties. On a HP or IBM machine with about 100MB, these properties have been verified. The verification of all the properties took less than 30 minutes.

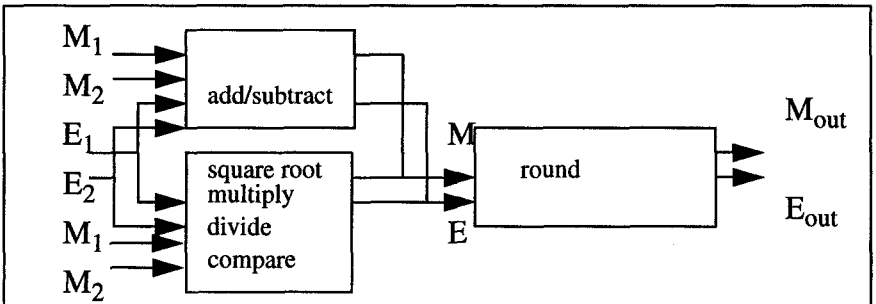

Figure 2: Block Diagram of FPU

The floating-point unit (FPU) of the processor under test uses a radix-4 SRT division algorithm. Since the SRT division algorithm is iterative, the loop invariant verification technique described in [3] is used.[1] Given the mantissa *d* (from M1 in Figure 2) of the dividend and the mantissa *b* (from M2 in Figure 2), the radix-4 SRT algorithm iteratively computes a partial remainder r_i and a quotient digit q_i. The partial remainder r_0 is initialized to *d/4* and the quotient digit q_0 is initialized to zero. Each iteration the algorithm gets the quotient digit from the lookup table and subtracts q_i. *b* from the partial remainder r_i that has been shifted left by 2 bits. In other words, $r_{i+1} = 4.r_i - q_i.b$. The algorithm terminates when enough quotient bits have been computed. Suppose that the quotient digits are within the range {-n, -n+1, . . . , -1 , 0, 1, . . . , n-1, n} for some positive *n*. Then a radix-4 SRT division algorithm is guaranteed to be correct if both of the following properties are true in each division loop [14]:

$$r_{i+1} = 4.r_i - q_i.b$$
$$|r_i| <= n.b/3$$

The loop invariant INV_i that we want to verify is the conjunction of the two properties above. We want to verify that the invariant INV_0 is true initially and also $INV_i => INV_{i+1}$.

1. The correctness of this property decomposition was only manually proved.

In the previous experiments, the loop invariants INV_i were expressed in extended SML. For example, for any constant *n*, the second property of INV_i was specified as follows.

```
AG(3.r_i <= n.b) & (0 <= 3r_i + n.b)
```

Additionally, in order to represent the arithmetic expressions (e.g. $3.r_i$) internally as words, the bit-level signals had to be grouped to form an array. The grouping had to be specified explicitly in extended SML. In order to represent the words that will be represented as HDD in the formal verification engine as BMD instead of MTBDD, the state variables in the fan-in cone of the operands of the expression (e.g. r_i) had to be specified as data variables. In the case of the property above, if the state variables in the fan-in cone of r_i are not specified as data variables, the expression $3.r_i$ will be represented as an MTBDD instead of a BMD. Since the MTBDD representation of multiplication operation has exponential size relative to the bit size and the bit size of r_i exceeds 64, this expression is not representable as MTBDD. In short, a failure in the specification of data variables and the grouping of bits to words may cause the system to blow up, and a lot of user expertise, and knowledge on BMD and MTBDD technology were required to manually specify the properties in extended SML.

Prover, in order to prevent the generation of inefficient SML and reduce the work that needs to be performed by the user, automates the process of extended SML generation. The user is required to write the specifications in FSL. In the case of the CTL property specified above, the user instead of writing an extended CTL/SML property writes the semantically equivalent FSL correspondent of the property above. The user will not need to decide which operations and which vectors will be represented at *word-level*. The decision to represent vectors as BMD/MTBDD is done internally in the model builder stage of *Prover* (See Figure 1). The model builder automatically generates for the FSL checkers extended SML checkers.

The experimental results demonstrate that the verification of these properties by *Prover* is as efficient (with respect to the computation time and memory usage) as the method that requires extensive user guidance.

7 Conclusions

In this paper, we have illustrated how automatic datapath extraction techniques can help improve the productivity of the formal verification process by reducing the need for user intervention and expertise required in the current HDD-based applications. Furthermore, in order to prove the efficiency and applicability of this technique to verify arithmetic circuits of industrial size and complexity, we have re-verified some of the selected floating-point operations (e.g. FDIV, FSQRT) in the floating point unit (FPU) of Intel's *Pentium*® microprocessor. The division and square-root algorithms of the FPU have been verified using manual extraction of datapath in [3]. Our system reduces extensively the user intervention and expertise that was needed to verify these operations by integrating an automatic datapath extraction mechanism to Intel's formal verification system, *Prover*. The experimental results demonstrate that the verification of these properties by *Prover*

is as efficient (with respect to the computation time and memory usage) as the method that requires extensive user guidance.

By automating the process of extended SML generation, we have broadened the usage of the HDD technology. The manual specification and identification of all the datapath operations in industrial size models is not humanly possible. Therefore, *Prover* automatically detects the datapath operations in the model, in addition to the specification, that can be more efficiently represented by HDD. Additionally, the specification of the properties in a high-level linear-time language, FSL, that is more close to the hardware than CTL, reduces the risk of verifying specifications that do not specify exactly what the user meant to verify. In other words, *Prover* by enabling the user to specify the properties in a high-level specification language ensures that he/she specifies what he/she really meant to verify.

Prover with the new high-level specification entry and automatic datapath extraction mechanism is being successfully used to verify Intel's new generation microprocessors. Lately, complex arithmetic operations in an Intel's next generation micro-processor have been successfully verified using *Prover*. The techniques specified in this paper promote the mass usage of formal verification in Intel's design environment.

8 Acknowledgments

We would like to thank Dany Khabaza, Roni Rosner, and Andreas Tiemeyer for their continuous involvement in the development of *Prover*. Orit Kedem has helped a great deal in the verification of the floating-point operations using *Prover*. We would also like to thank our colleagues at Intel Development Labs for providing us the test cases for the experiments reported in this paper.

9 References

[1] R.E.Bryant and Y.A. Chen.Verification of Arithmetic Functions with Binary Moment Diagrams. In Proceedings of the 32nd ACM/IEEE Design Automation Conference, IEEE Computer Society Press

[2] E.Clarke, X.Zhao. Word Level Symbolic Model Checking, CMU-CS-95-161

[3] Y.Chen, E.Clarke, Pei-Hsin Ho, Y. Hoskote, T. Kam, M. Khaira, J. O'Leary, X. Zhao. Verification of All Circuits in a Floating-Point Unit Using Word-Level Model Checking, In Proceedings of the International Conference on Formal Methods in Computer-Aided Design, November 1996

[4] R.E.Bryant. Bit-level Analysis of an SRT Divider Circuit. Technical Report, Carnegie Mellon University, 1995

[5] R.E.Bryant. Graph-based Algorithms for Boolean Function Manipulation. IEEE Transactions on Computers, c-35(8):677-691, Aug. 1986

[6] Laurent Arditi. *BMDs Can Delay the Use of Theorem Proving for Verifying Arithmetic Assembly Instructions, In Proceedings of the International Conference on Formal Methods in Computer-Aided Design, November 1996

[7] R.Hojati, R.K.Brayton. Automatic Datapath Abstraction In Hardware Systems, In Proceedings of the International Conference on Computer-Aided Verification Conference, 1995

[8] A.Aziz et.al. HSIS: A BDD-Based Environment for Formal Verification, In Proceedings of the 31st Design Automation Conference, IEEE Computer Society Press

[9] K.Hamaguchi, A. Morita, and S.Yajima. Efficient Construction of Binary Moment Diagrams for Verifying Arithmetic Circuits. In Proceedings of the International Conference on Computer-Aided-Design, pages 78-82, San Jose, CA, November 1995.

[10] K.Ravi, A.Pardo, G.Hachtel, F.Somenzi. Modular Verification of Multipliers. In Proceedings of the International Conference on Formal Methods in Computer-Aided Design, Palo Alto, CA, November 1996

[11] M.Fujita. Verification of Arithmetic Circuits by Comparing Two Similiar Circuits. In Proceedings of the International Conference on Computer-Aided Verification, 1996

[12] K.S.Brace, R.L.Rudell, and R.E.Bryant. Efficient Implementation of a BDD Package. In Proceedings of the Design Automation Conference, pages 535-541, San Francisco, CA, June 1995.

[13] E.M.Clarke, M. Khaira, and X.Zhao. Word Level Model Checking - A New Approach for Verifying Arithmetic Circuits. In Proceedings of the 33rd ACM/IEEE Design Automation Conference. IEEE Computer Society Press, June 1996.

[14] D.E.Atkins. Higher-radix Division Using Estimates of the Divisor and Partial remainders. IEEE Transactions on Computers, C-17(10):925-934, October 1968.

[15] R.Enders. Note on the Complexity of Binary Moment Diagram Representations. unpublished paper, Siemens AG, Munich Germany, 1994.

[16] K.L.McMillan. Symbolic Model Checking, Kluwer Academic Publishers.