Evaluation of Embedded Processor Based BDD Implementation

Danko Ivošević and Vlado Sruk
Department of Electronics, Microelectronics, Computer and Intelligent Systems
Faculty of Electrical Engineering and Computing, University of Zagreb
Complete Address: Unska 3, Zagreb, 10000, Croatia
Phone: (+385) 1 6129 926 Fax: (+385) 1 6129 653 E-mail: danko.ivosevic@fer.hr

Abstract - Different strategies for implementation of computationally intensive applications in hardware are available today. The spectrum of implementations ranges from usage of standardized microprocessors to specially tailored hardware solutions. Available processor architectures range from general purpose type through processors with instruction set extensions to application-specific processors. On the other side, recent advances in design automation resulted in development of C-to-hardware compilers as a new strategy for application implementation in hardware.

In this paper, we present and elaborate characteristics of hardware implementations of Binary Decision Diagrams (BDDs) application, used in many research and development areas, and especially in formal verification and Computer-Aided Design (CAD) tools. For this application, processor architecture using C-to-hardware NISC toolset is manually tailored and compared with implementation approaches based on standard soft and hard processors. All these approaches are implemented and verified using FPGA Virtex-5 development board. Our results show that, besides code and compiler side optimizations, more significant improvements in total execution cycles count can be achieved when processor architecture side optimizations are included.

I. INTRODUCTION

Binary Decision Diagram [1] is a tree structured notation used for representation of Boolean functions [2]. The ability of storing Boolean expressions in data structure utilized in software production expands the spectrum of its application. Research and development of formal verification tools benefit from it in representation and manipulation of different functional behavior notations. The importance of BDDs also grows with software products growth since it is used in sequential systems and program analysis [3]. In CAD research area they are used in logic optimization and synthesis [4] and VLSI design [5].

Manipulation of a large BDD tree is a computationally intensive task. The size of the tree depends on variable ordering applied. There are numerous software and hardware BDD implementations that explore the problem and search for the method of BDD task acceleration. Because of the imposed algorithm complexity we have focused on evaluation of its potential hardware implementation in FPGA devices.

Development of specialized hardware solutions, with imposed functional, environmental and performance requirements, is always hosted by some kind of embedded environment. In such environments, different methodologies to real world application acceleration are at the disposal of the designers. The common choice is custom and semi-custom design relying on available custom developed functional cores. While having in mind the shortening of overall design time, the semi-custom design using customizable microprocessor cores seems as a suitable approach.

In this paper, we present our findings on different implementation approaches for a highly recursive application such as is BDD building application when using standard embedded processors and C-to-hardware tool with a possibility of processor architecture customization. The features of soft and hard processor cores and their final FPGA implementations' results are compared with FPGA implementation based on No-Instruction-Set Computer (NISC) toolset, developed at the UCI Center for Embedded Computer Systems [6]. The comparison is presented in terms of a cycle count required to complete the BDD building task.

In Section II, the related work on the BDD structure implementations and specific features of BDD manipulation algorithm are reviewed. Section III discusses details of the approaches implemented, the NISC based and the implementations relying on available embedded processor cores. Section IV evaluates and compares the results of each implementation with a more detailed analysis of the NISC based implementation features. Section V sets out the conclusions arising from the implementations' comparison.

II. BINARY DECISION DIAGRAMS IMPLEMENTATION

The theory of Binary Decision Diagram structure building complexity developed as VLSI designs and formal verification inputs experienced high-rate growth. Along with development of different implementations aiming to achieve the desired performance, the techniques for BDDs' optimizations in practice have been sought. The focus primarily stayed on the problem of variable ordering during the process of tree building. The order of entered variables influences the size of the tree as one logic function can be represented with different resulting BDDs. Since it was identified as the central problem of BDD complexity, significant efforts aimed at the BDD size reduction focused on the problem of variable ordering [7].

At the same time, general efforts aimed at improving performance led to miscellaneous approaches and implementations. There were parallelizing approach attempts [8], specially designed hardware [9], and other proposals that attempt to minimize time and memory resources consumption [10,11]. Since the topic has remained interesting until today and new reprogrammable implementation technology has appeared, the work on hardware acceleration exploiting FPGA devices in formal verification area has become a new potential direction [12]. Accordingly, the FPGA device is employed in this paper as the implementation platform for all the applied approaches.

The general BDD manipulation algorithm scheme is usually organized as a node table where each node is accessed by its hash value to improve the node search [13], Fig. 1. The algorithm builds the tree of nodes upon receiving new variables. With every new node upgrade on existing sub-trees of previously entered variables is performed, Fig. 2. When traversing the tree to evaluate logic expressions for a vector of values, or just searching for a node, the algorithm is moving down the tree with recursive routine calls. As the tree grows, the number of recursive calls increases. This emphasizes the dominant recursive nature of the algorithm.

Further important characteristic of this approach is an extensive usage of modulo operations to implement the required hash function. Having complex design, the divider operations can be expensive in terms of the required processing clock cycles and degradation of the overall performance.

Therefore, apart from the source code application characteristics, the performance is dependable on target architecture where the application is executed. As processor based implementations are applied, certain architecture details are closely related to performance. The conventional processors have fixed architecture with predefined instruction set. The performance improvement responsibility is on the compiler side which schedules the instructions from processor's instruction set with application of code optimizations. Opposite to that, the NISC architecture is customized by user and issued to the compiler which defines the instruction format and schedules appropriate execution cycles. There is no predefined instruction set so that the instruction format is dynamically created adaptively to a given architectural structure. This NISC capability gives it advantage over other processors in achieving shorter execution cycle sequences when implementing the same input code.

III. THE EVALUATED IMPLEMENTATIONS

In this section, we provide further information on NISC and other embedded processors along with the note on their execution environment.

No-Instruction-Set Computer has proved itself as the appropriate concept for a wide range of applications [14]. It directly exploits the nature of application written in C code on the architectural basis. The code and structural description of processor architecture are taken as inputs to NISC toolset compiler. The compiler produces the control words tailored for the architecture and scheduled according to functionality defined by the code. The appropriate implementation is formed as FPGA synthesizable RTL code, Fig. 3. The concept of control words formed according to the architecture has shown to be efficient in terms of execution time and power in examples of data and computationally intensive code [15]. As no instruction set was prepared in advance for some fixed processor architectures, the control words are longer than typical RISC or CISC instructions.

The generic NISC processor architecture consists of split control and data path, Fig. 4. The control unit consists of memory filled with control words and logic that picks the appropriate words during execution. These words applied to data path components, like busses, units, registers or data memory, realize the application's functionality.

The data path used in generic NISC implementation consists of Data Memory, Register File, and Comparator, ALU, Multiplier and Divider units connected with busses, wires and registers. The ALU unit performs additions, subtractions, shifts and all logical bitwise operations. Data and controller memories are instantiated by proprietary Xilinx cores, as well as a divider unit [16]. Other components are directly implemented inside FPGA logic resources during compilation.

For every application the block of control words produced by the compiler is stored in controller memory

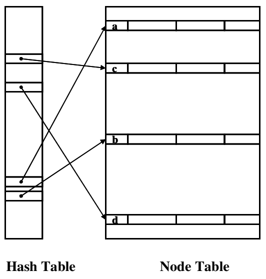


Fig. 1. Binary Decision Diagram nodes records organization

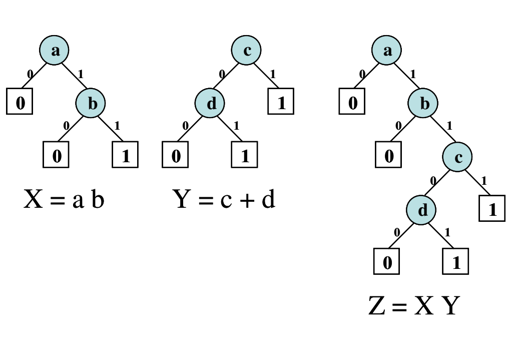
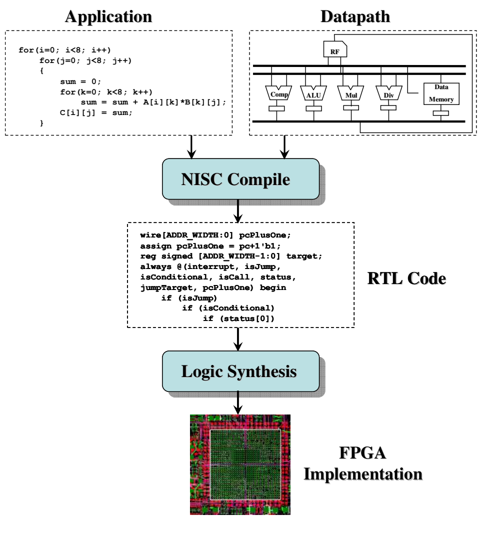


Fig. 2. Binary Decision Diagram composition



```
for(i=0; i<8; i++)
    for(j=0; j<8; j++)
    {
        sum = 0;
        for(k=0; k<8; k++)
            sum = sum + A[i][k]*B[k][j];
        C[i][j] = sum;
    }
```

```
wire[ADDR_WIDTH:0] pcPlusOne;
assign pcPlusOne = pc+1'b1;
reg signed [ADDR_WIDTH-1:0] target;
always @(interrupt, isJump,
isConditional, isCall, status,
jumpTarget, pcPlusOne) begin
    if (isJump)
        if (isConditional)
            if (status[0])
```

Fig. 3. NISC Toolset Flow

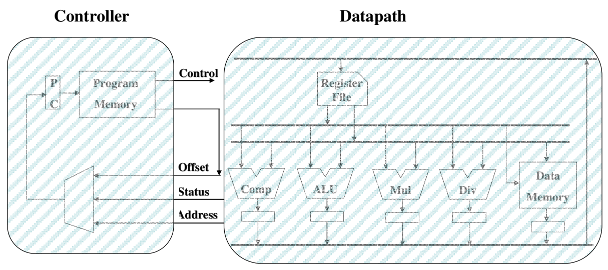


Fig. 4. Generic NISC architecture scheme: controller and data path

and scheduled in exact cycle order. The toolset in every cycle explicitly reports on all issued control signals and active data path parts. Such application execution profiling is presented in Section IV with comparative results of different implementations.

In this work, we benchmark three processor cores against NISC in cycle count metric before and after applying code and architectural optimizations: the Altium's TSK3000A soft processor [17], and Xilinx's MicroBlaze soft processor [18] and PowerPC 440 hard processor [19]. TSK3000A soft processor is 32-bit MIPS-like by its architecture and instruction set provided inside Altium's integrated development environment. MicroBlaze is a soft 32-bit RISC Harvard processor widely utilized in embedded processing solutions, and PowerPC 440 is a 32-bit hard processor core embedded within Virtex-5 FX devices. Designs based on either of the processor are easily built with Xilinx's Base System Builder which configures all processor, memory, bus and peripherals settings. The BDD functionality we implement by NISC and three other processor cores is extracted from BuDDy package written in C code [20].

IV. Case Study: Multiplier Combinatorial Circuit

As a case study, the structure used to evaluate and compare the results of the described implementations is $n\times n$ bits integer multiplier combinatorial circuit. The operands are of n bits and the product is of $2n$ bits wide. For the input operands' width n the application builds the appropriate circuit as an input to BDD building procedures.

The general scheme nomenclature for such multiplier circuit is expressed in equations (1)-(4), where the operands are denoted as x and y and the product as z.

The particular sums are $s_{k,i}, i,k \in [0,1,\ldots,n-1]$ with carries $c_{k,i}, i,k \in [0,1,\ldots,n-1]$, as in 3×3 bits multiplier scheme, Fig. 5.

The previous works [21,22] elaborated multiplier circuit structure as one with exponential growth of representing BDD nodes when raising bit width n. The scale of growth is dependable on the variable ordering applied. According to [21] for n bits size multiplier the lower bound of produced BDD nodes representing Boolean function at j - 1 or $2n - j$ – 1 product output position is 1.09^j.

$$z = x \cdot y \qquad (1)$$

$$x = \sum_{i=0}^{n-1} x_i \cdot 2^i \qquad (2)$$

$$y = \sum_{k=0}^{n-1} y_k \cdot 2^k \qquad (3)$$

$$z = \sum_{j=0}^{2n-1} z_j \cdot 2^j \qquad (4)$$

However, the growth of the BDD tree size for the algorithm implemented in this work appears to be with factor of 3, as our experiments have shown. The 3×3 bits multiplier circuit structure finally produces 106 BDD nodes with 990 division operations and 550 recursive procedure calls during circuit processing phase. It has been chosen as a case study for simplicity reasons, while scaling the problem on a larger multiplier structure would only emphasize the notions of the work.

In the following chapter, the results for the NISC based implementation are presented and discussed. After that, other implementations' results are shortly reviewed.

A. No-Instruction-Set Computer Based Implementation

The NISC toolset accepts the code and architecture description and, besides synthesizable RTL code, it produces the precise reports on control words construction and full translation of C code into constructed control words. By capturing controller memory address dump during execution and control words interpretation information, the execution is accurately profiled up to the level of the register-to-register transfer.

The analysis of the NISC based implementation using generic architecture reports little over 100,000 clock cycles where 96% of them are spent in recursive routines. As division operation is an "expensive" operation executed by outer 32-bit divider core instantiated in the design, its cycle consummation is high, i.e. 32% of total cycle count. Another point where the significant cycle number is spent is in entrance and exit parts of the routines. Every time the routine is called there is some processing on passing arguments and setting the routine's local context, i.e. the return address and caller's local context and local variables. Every time the routine is exited, the caller's local context is retrieved. When summing all cycles belonging to

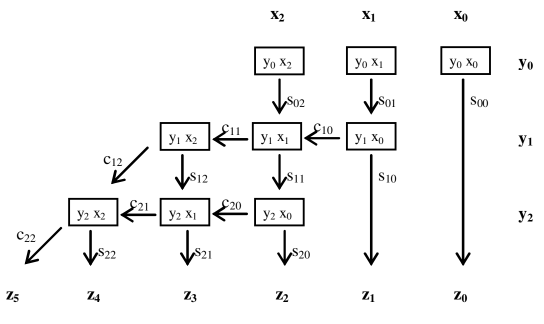


Fig. 5. Multiplier combinatorial circuit scheme (3x3 bits)

those specific parts of the routine processing, it appears that they consume 16% of all cycles. This is the consequence of numerous recursive calls as is the nature of the BDD manipulation algorithm.

According to profiling and performance analysis, we examined several code transformations and architectural refinements to decrease the total cycle count:

a. The hash functions for accessing nodes incorporate simple divisions by 2 which allowed replacement of 2/3 of all divisions with right shift operation.

b. For all multiplier sizes up to 7x7 bits the number of produced nodes is small enough so that replacement of the 32-bit divider core with 16-bit appeared to be allowed. The maximum divisor value is limited to the number of resulting BDD nodes as appropriate division operation is applied in nodes caching. In reality, the cache size is expected to be much lower than total number of nodes so that the divider core replacement done here is thus fully justified. As the divider latency is almost linear to input operands bit widths, the gain expected from this change is almost 50% in divider cycles.

c. The architecture was updated with another ALU, and, additionally, with forwarding paths from all computational units to the comparator. The additional ALU is intended for parallelization of routines' context handling during recursive calls where multiple additions to stack and frame pointers addresses in data memory are issued.

d. Forwarding paths to comparators were added since many branches dependable on temporary calculations had been noticed.

The summary of architectural changes is presented in Fig. 6 where the final data path is shown, while the comparison of cycle counts got from logic analyzer outputs for all implementations is shown in Fig. 7.

The first two refinements succeeded in the reduction of divider's cycles by almost 6 times while putting its total cycles' consummation below 10% of all execution cycles. At the same time, with decreasing the total number of cycles the impact of routines' entrance and exit cycles grew to over 20% of all cycles.

Further refinement of adding another ALU unit helped in a speedup of handling of routines' local contexts. When calculating the context frame, setting the new stack pointer and reserving the space for local variables in memory, the second ALU significantly contributed to execution parallelization.

All the considered optimizations contributed to an execution cycle count decrease by almost 40% while keeping division cycles below 10% and recursive influence on all routines' calls handling below 20%. For all implementations the dominantly recursive execution is confirmed as those routines account for 94-97% of all cycles.

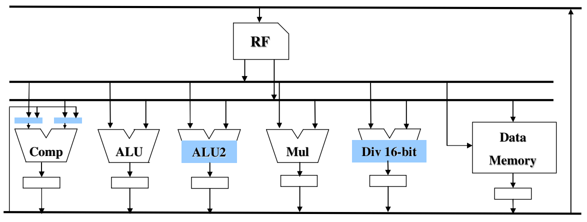


Fig. 6. NISC architectural changes proposed

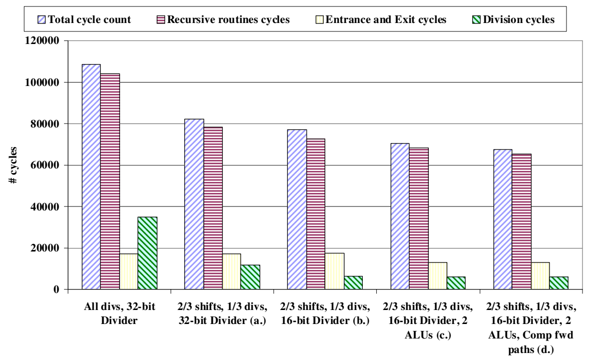


Fig. 7. Comparison of NISC based implementations

B. Implementations Comparison

The No-Instruction-Set Computer based implementation is compared to embedded processor based implementations. The optimizations applied for soft (TSK3000A, Microblaze) and hard (PowerPC) processor cores based implementations include code changes described in the previous chapter and different levels of compiler level optimizations. The difference between non-optimized and fully optimized implementations for all embedded processors and NISC are presented in Table I. There is a 35-55% range of improvements for implementations after applying optimizations. When compared to NISC, other implementations need 30-400% more cycles.

V. CONCLUSION

In this paper, strategies for FPGA based implementation of building Binary Decision Diagrams (BDDs) are explored. Three standard embedded microprocessor based implementations are compared against No-Instruction-Set Computer (NISC) processor customized implementation. We explore NISC concept applicability on recursive dominant type of application, as the BDD manipulation algorithm recursively traverses its own tree structure.

The different FPGA implementations are compared in clock cycle count metrics. For processor cores, we varied different optimization levels applied to the corresponding compiler. On the other hand for NISC, we introduced

TABLE I
CYCLE COUNTS COMPARISON

Implementation	Non-optimized	Optimized	Improvement
TSK3000A	135,209	88,553	34.5%
PowerPC440	744,673	329,754	55.7%
MicroBlaze 7.20b	190,507	121,390	36.3%
NISC	108,338	67,591	37.6%

several optimizations in source code and processor architecture changes. Since NISC compiler dynamically creates instruction format and its contents tailored for the provided processor architecture, its final implementation outperforms the others in the cycle count metric at least by 23%.

NISC advantage in customization of architecture and specifically the data path where particular computation units are refined or some new added enables straightforward architecture changes' implementation. Another important NISC toolset advantage is the possibility of definition of especially dedicated computational units that perform only ones specific operation, or a subset of operations.

Furthermore, we have noticed that after applying several code and architecture optimizations to NISC based implementation, the recursive nature of the algorithm becomes more dominant. In this manner, it limits performance improvements for all implementations. Therefore, efforts aimed at acceleration of such application in specialized hardware architectures, or in parallelizing approach through multiprocessing or distributed environments, have potential to provide an additional improvement.

The future research will focus on the target FPGA platform dependable metrics, such as are the achieved work frequencies, logic resource allocations and, consequently, the actual execution time and power dissipation for all proposed implementations.

ACKNOWLEDGMENTS

This work is supported by research grant No. 036-0362980-1929 from the Ministry of Science, Education and Sports of the Republic of Croatia and Unity through Knowledge Fund (UKF).

The work builds upon many years of research and experience of staff at the Center for Embedded Computer Systems (CECS), University of California, Irvine.

The authors would like to thank Valentina Zadrija for providing auxiliary measurements.

REFERENCES

[1] S. B. Akers, "Binary decision diagrams," *IEEE Transactions on Computers*, vol. 27, pp. 509–516, June 1978.

[2] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Transactions on Computers*, vol. 35, pp. 677–691, August 1986.

[3] O. Lhoták, "Program analysis using binary decision diagrams," Ph. D. thesis, School of Computer Science, McGill University, Montreal, Canada, January 2006.

[4] C. Yang and M. Ciesielski, "BDS: a BDD-based logic optimization system," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 21, pp. 866–876, July 2002.

[5] C. Meinel and T. Theobald, Algorithms and data structures in VLSI design - OBDD foundations and applications, 1st ed., Ed. Berlin, Germany: Springer-Verlag, 1998.

[6] "NISC Technology and Toolset home page," http://www.ics.uci.edu/ ~nisc/.

[7] B. Bollig and I. Wegener, "Improving the variable ordering of OBDDs is NP-complete," *IEEE Transactions on Computers*, vol. 45, pp. 993–1002, September 1996.

[8] T. Stornetta and F. Brewer, "Implementation of an efficient parallel BDD package," DAC, pages 641-644, 1996.

[9] T. Yoneda and T. Ishigaki, "Hardware acceleration for BDD manipulations," Tokyo Institute of Technology, Tokyo, TIT CS Tech. Rep. TR99-0015, 1999.

[10] S. Minato, "Streaming BDD manipulation," *IEEE Transactions on Computers*, vol. 51, pp. 474–485, May 2002.

[11] P.W. C. Prasad, A.Assi, M. Raseen and A. Harb, "Selective min-terms based tabular method for BDD Manipulations," WASET, pages 122-125, 2005.

[12] M. Safar, M. W. El-Kharashi and A. Salem, "An FPGA based accelerator for SAT based combinational equivalence checking," IWSOC, pages 419-424, 2005.

[13] K.S. Brace, R.L. Rudell and R.E. Bryant, "Efficient implementation of a BDD package," DAC, pages 40-45, 1990.

[14] M. Reshadi, B. Gorjiara and D. Gajski, "NISC technology and preliminary results," Center for Embedded Computer Systems, Irvine, CA, Tech. Rep. TR05-11, 2005.

[15] B. Gorjiara, M. Reshadi and D. Gajski, "Designing a Custom Architecture for DCT Using NISC Technology," ASP-DAC, pages 116-117, 2006.

[16] "Xilinx Divider Generator v3.0," http://www.xilinx.com/support/ documentation/ip_documentation/div_gen_ds530.pdf.

[17] "Altium TSK3000A 32-bit RISC Processor," http://www.altium.com/ files/learningguides/CR0121 TSK3000A 32 bit RISC Processor.pdf.

[18] "MicroBlaze – The Industry's Most Flexible Processing Solution," http://www.xilinx.com/publications/prod_mktg/MicroBlaze _Sell_Sheet.pdf.

[19] "PowerPC 440 home page," http://www.xilinx.com/support/ documentation/ipembedprocess_processorcore_ppc440.htm.

[20] "BuDDy - A Binary Decision Diagram Package home page," http://vlsicad.eecs.umich.edu/BK/Slots/cache/www.itu.dk/re search/buddy/index.html.

[21] R. E. Bryant, "On the complexity of VLSI implementations and graph representations of Boolean functions with application to integer multiplication," *IEEE Transactions on Computers*, vol. 40, pp. 205–213, February 1991.

[22] I. Grudenić, N. Bogunović, "BDD complexity analysis of multiplier circuits," MIPRO, pages 31-34, 2005.