Automated modeling of custom processors for DCT algorithm

D. Ivošević and V. Sruk
Faculty of Electrical Engineering and Computing / Department of Electronics, Microelectronics, Computer and Intelligent Systems, University of Zagreb, Croatia
danko.ivosevic@fer.hr

Abstract - The needs for automated digital system design rise with constant technology improvements and time-to-market shortening. High-level synthesis tools cope with this problem by raising the design specification to a higher level. We have implemented the methodology of custom processor automated modeling for DCT algorithm. This algorithm is often used in signal and image processing applications. At front end, the methodology assumes C code input specification as it is the case of many high-level synthesis tools. C programming language popularity offers the applicability for a broader spectrum of users. At back end, logic synthesis tools produce the FPGA implementation of the algorithm. The results are evaluated in terms of execution cycles required for completing the algorithm and processor's datapath components allocations, and compared to previous work.

I. INTRODUCTION

Moore's Law [1] has accurately predicted the growth in IC complexity in the last 40 years and it has remained a reliable method of calculating future trends and the pace of innovation. It states that the capability of technology rises twice every 18 months, while the hardware design productivity grows by 1.6 times in the same period. In the same time there is a software productivity gap as the needs for software support became higher with the potentials of hardware. The actual software productivity grows twice every 5 years, and the requirements are much higher as they rise twice every 10 months [2].

The opinions are that the key to bridging the software gap lies in increased development of EDA (Electronic Design Automation) tools for IP (Intellectual Property) cores production that will help to close the hardware gap. This concept of hardware-software co-design is closely and thoroughly investigated in embedded system design especially in the last two decades. As stated in [3], the embedded design primarily consists of hardware and software synthesis, and the synthesis of their interface and communication channels.

The synthesis of hardware in a more traditional design view starts by writing the Hardware Description Languages (HDLs) code, i.e. VHDL or Verilog. This code has to be written by logic synthesis standards, timing constraints, functional and interface requirements. It is a feasible task when there is plenty of time available, but time-to-market pressures cause that only some requirements (functional at first) are fulfilled. In High-Level Synthesis (HLS) approach the design specification is raised to the style of higher level programming languages, i.e. C and C++ or SystemC. It is also referred to in the literature as C synthesis, or algorithmic synthesis as it is an automated process of hardware design from algorithmic specification. This process analyzes the input code and produces RTL schedule led by the architectural constraints. The logic synthesis process translates the RTL description into an integrated circuit. Lifting the specification level means better control over design, optimizations and verification at RTL level by means of specialized tools. In the mid 90s when it emerged, the C synthesis appeared to be successful industrial solution. The key was in the popularity, abstraction level and flexibility of C programming language and its integration of design flows and modeling of the time. The ratings of programming languages' popularity [4,5] show that C and C++ with usage rate of approximately 25% range among the top 3 languages. On the other hand, hardware description languages have a usage rate below 2%.

The high-level synthesis is targeted for ASIC and FPGA based designs. There are several tools that take C input specification and produce RTL code, and use logic synthesis tools to produce target implementation [6]. They are all conceptually similar but can differ in the scope of optimizations they perform on input code and the allowed level of user interventions. One of them, No-Instruction-Set Computer (NISC) is developed at the UCI Center for Embedded Computer Systems [7]. It is a C-to-Verilog compiler that assumes user customizable processor architecture as input, besides the input C code. In a previous work [8], we showed the principle of applying the code and architectural optimizations using the NISC toolset on a case of Binary Decision Diagram (BDD) structure manipulation.

The code optimizations techniques were extensively investigated in compiler design topic research and some of the high-level synthesis tools [9]. The architectural customizations were investigated to a lesser extent. High-level synthesis tools usually produce low-level design implementation without its microprocessor level interpretation, but there are processor implementations with configurable instruction sets, i.e. Application-Specific Instruction-Set Processors (ASIPs) [10].

The FPGA device as implementation platform usually has a low operational frequency, but the design can be customized to fully extract parallelism. The actual

comparisons of FPGA, Graphic Processor Unit (GPU) and general purpose CPU show a high performance of FPGA based implementations in image processing tasks [11].

In this work, we have implemented the methodology of processor architecture design automation according to C code input requirements, as introduced in [3]. We have analyzed it on DCT algorithm case and presented the results in execution cycle count metric. The strength of the approach is in a rapid design development and the style of specification familiar to the huge population of users. The methodology assumes the input code as it is and does not apply any compiler-style optimizations, but fully customizes the architecture. Thus, the emphasis is on optimizations that are closer to the implementation platform while still preserving the processor style of execution. The hard-to-solve problem of HLS flow optimizations is broken into two stages: one that presents the execution engine, i.e. the processor, and the other that maps it on the implementation platform. The FPGA implementation platform is flexible enough to be considered as a natural target of such rapid design development style. The available NISC toolset is the prototype of the tool that implements the design flow.

In the following Section II there is a short outline of the related work on DCT implementations and the NISC concept. Section III describes the methodology of custom processor design, while Section IV shows its application to DCT algorithm. Section V discusses the results, and Section VI provides the final observations on the applied methodology and the presented results.

II. Related Work

A. No-Instruction-Set Computer

The related work on processor architecture customization is based on NISC toolset. The concept of this toolset allows full customization of datapath and program words that make up the control unit. The customization of datapath assumes that a particular functional unit is responsible only for a single operation or combination of operations. Thus the minimization of datapath resources is supported. The program words are constructed according to the contents of datapath. The architecture is translated into synthesizable Verilog code targeted for FPGA implementation.

B. Manual Design

Previous work [14] uses NISC toolset to explore the design space for discrete cosine transform. The transform was calculated using two consecutive 8×8 matrix multiplications, Listing 1.

The code and architectural exploration have been performed synchronously, and respective designs have been synthesized for Virtex-II FPGA device. The evaluation metrics included the execution time by means of the number of execution cycles and clock frequency, power and energy consumption and area occupation.

The architecture exploration started with NISC predefined architecture constructed as general-purpose MIPS style datapath and it underwent through multiple transformations. For the MIPS style datapath the DCT code required over 10,000 execution cycles. The C code was then transformed to increase the parallelism in the code. The inner-most loop was unrolled, two outer loops merged and the functional units simplified, i.e. addition and multiplication were converted into OR and AND operations, Listing 2.

Parallel with that, architectural transformations were conceived to be mapped to the code transformations, Fig. 1. The *OR-ALU* units' chain is instanced to accomplish the *A* and *B* array addressing (odd lines starting with *Line05* to *Line19* in Listing 2) in one cycle. The read values (*aL*, *bL*) have been propagated to *Mul-Adder* units cascade to sum the products of arrays' values (even lines starting with *Line06* to *Line20* in Listing 2).

```
Line01: for(i=0; i<8; i++)
Line02: {
Line03:   for(j=0; j<8; j++)
Line04:   {
Line05:      sum = 0;
Line06:      for(k=0; k<8; k++)
Line07:      {
Line08:         sum = sum + A[i][k]*B[k][j];
Line09:      }
Line10:      C[i][j] = sum;
Line11:   }
Line12: }
```

Listing 1. DCT code

```
Line01: ij=0;
Line02: do {
Line03:   i8 = ij & 0xF8;
Line04:   j = ij & 0x7;
Line05:   aL = *(A+(i8|0)); bL = *(B + (0|j));
Line06:   sum = aL * bL;
Line07:   aL = *(A+(i8|1)); bL = *(B + (8|j));
Line08:   sum += aL * bL;
Line09:   aL = *(A+(i8|2)); bL = *(B + (16|j));
Line10:   sum += aL * bL;
Line11:   aL = *(A+(i8|3)); bL = *(B + (24|j));
Line12:   sum += aL * bL;
Line13:   aL = *(A+(i8|4)); bL = *(B + (32|j));
Line14:   sum += aL * bL;
Line15:   aL = *(A+(i8|5)); bL = *(B + (40|j));
Line16:   sum += aL * bL;
Line17:   aL = *(A+(i8|6)); bL = *(B + (48|j));
Line18:   sum += aL * bL;
Line19:   aL = *(A+(i8|7)); bL = *(B + (56|j));
Line20:   *(C + ij) = sum + (aL * bL);
Line21: } while(++ij!=64);
```

Listing 2. Unrolled DCT code

Such synchronized set of transformations shortened the schedules to approximately 2900-3500 cycles while increasing the clock frequency and shortening the overall execution time. The occupation of Virtex-II FPGA device implementation area was also minimized. The occupation of the area was achieved owing to dedicating the functionality of functional units to the required minimum of operations. For instance, the *OR* unit is dedicated only to logical OR operation while *ALU* unit is dedicated only to adding and logical AND operations. In addition, the data memory (*DMem*) is instanced with minimum size

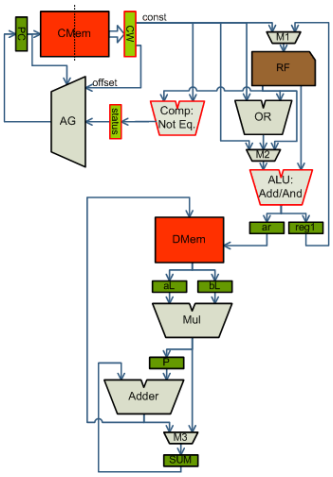

Figure 1. Manually designed architecture for unrolled DCT code ([14])

required by *A*, *B* and *C* arrays, and the control memory (*CMem*) is made up only of control words that implement the provided C code.

C. Automated Design

As for NISC based design automation, previous work [15] attempted to automatically derive the datapath from C code. It extracted the initial datapath for maximum performance in accordance with C code properties, and then optimized it under imposed resource constraints. These constraints were expressed in terms of the numbers of allowed functional units and register files ports. The key strengths of such approach are a rapid design development with scalable input C code specifications and a quality of designed cores comparable to manual designs. The methodology of similar approach is shown in next two sections on DCT case.

III. METHODOLOGY

A. Traditional High-Level Synthesis Flow

Traditional HLS flow has a number of common steps: lexical processing, algorithm optimizations, control and data dependencies analysis, technology library processing, resource allocation, operation scheduling, functional units and register bindings as well as output processing [16, 17].

The lexical processing translates the source code into internal representation. The source-code optimizations that follow have much in common with high-level and parallelizing compiler optimizations. After that, the control and data dependencies analysis identifies inputs and outputs of the operations and their data dependencies in the algorithm. The time component is not considered in this task, but is introduced during the technology library processing. The libraries representing specific hardware design technology are processes focusing on functional, timing and resource occupation features. The resource allocation phase identifies the set of functional units required for design implementation. The operation scheduling converts the algorithm into Finite State Machine (FSM). Based on the existing data dependencies and library-defined functional unit latencies the algorithm operations are scheduled by states, i.e. execution cycles operated by clock signal. The principles of scheduling can be diverse, depending on delays, resource limitations and pipelining capabilities. Basically, there is time-constrained (TC) scheduling and resource-constrained (RC) scheduling. The following phases bind the operations to functional unit items from the library and the variables to registers. The final step outputs the RTL code that implements the finite state machine, and is structured according to the logic synthesis optimization levels or the readability of the code.

B. Automated Processor Modeling Flow

The design flow applied in this work has some common HLS phases, but the main difference is that it outputs the processor architecture datapath, Fig. 2. The datapath is optimized according to the input application characteristics. Such architecture is purposed as input for NISC toolset which will be used as the compiler to RTL level.

The C code is converted into Control and Data Flow Graph (CDFG) [18]. Such representation exposes control and data dependencies present in the code. The CDFG representation relies on three-address code notation. Each line of the code is converted into one or more three-address code statements. During this process new, temporary, variables are introduced to preserve the data flow consistency. We use SPARK parallelizing compiler [19] to get CDFG from C code. The structure of the code is represented by basic blocks constructs of the CDFG. Basic blocks are units of code consisting of three-address code statements without control dependencies involved. The general structure of the input code is therefore represented as a set of basic blocks divided by control dependencies. For instance, the branch and loop constructs are transformed as shown in Fig. 3. The boxes represent the basic blocks with an appropriate set of three-address statements. The bodies (*if_body*, *else_body*, *loop_body*) of those constructs are separated from initialization (*loop_initialization*), increment (*loop_increment*) and test conditions (*branch_condition*, *loop_condition*). The basic blocks that conclude these constructs (*branch_end*, *loop_end*) are kept only to preserve the structure of the CDFG.

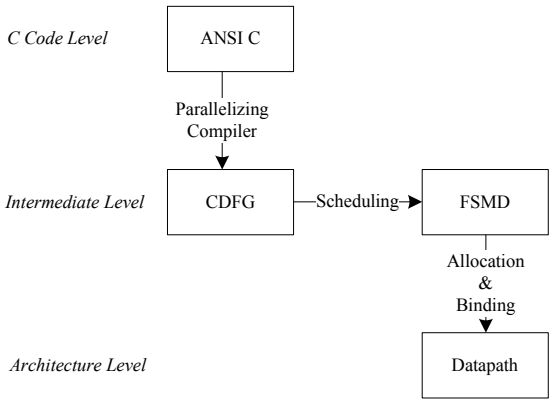

Figure 2. Path of transformations from C code to processor architecture

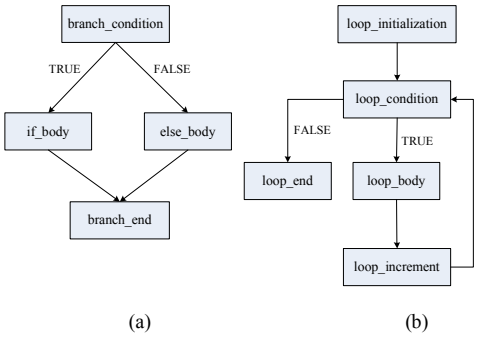

Figure 3. Control flow representation of (a) branch and (b) loop constructs

The data dependencies are localized inside the basic blocks. Thus, the scheduling of the three-address code is performed separately for each basic block. The applied principle of scheduling is As-Late-As-Possible (ALAP) list scheduling algorithm with a constraint on the number of data memory ports. This ALAP principle appears to expose more parallelism than As-Soon-As-Possible (ASAP) principle. The constraint on the data memory ports number is assumed by the fact that this number is usually limited for FPGA implementations. In NISC based implementations, the proprietary Xilinx IP cores with such limitations are used for memory blocks instantiation. The product of scheduling is Finite State Machine with Data (FSMD) representation where three-address statements are scheduled by states holding both information of operation performed and variables involved in the operation. The analysis of lifetimes of variables and usages of operations by states optimizes the allocations of registers and functional units and, consequently, accomplishes their bindings to variables and operations. The final architecture is composed by incorporation of particular basic blocks architectures.

IV. CASE STUDY: DCT ALGORITHM

A discrete cosine transform (DCT) is a mathematical form used in coding signals and images [12,13]. It is used in JPEG image compression and MPEG video compression standards. The most common variant is the 2-dimensional transform, or type-II DCT that takes the image digitized to pixels as input.

As in typical designs the image is sub-divided to 8x8 blocks of pixels, we also use DCT source introduced in Section II. Fig. 4 shows the CDFG representation of Listing 1 DCT code. There are three nested loops. The outer loop (*i* indexed in Listing 1) is closed within set of basic blocks denoted as BB1 to BB13 and is iterated 8 times. The first inner loop (*j* indexed in Listing 1) is closed within basic blocks BB3 to BB11 and is iterated 64 times, while the innermost loop (*k* indexed in Listing 1) is closed within basic blocks BB6 to BB8 and is iterated 512 times. The methodology described in Section III can be applied on all basic blocks to create the custom processor datapath. However, the basic blocks with a higher impact on the customization of the architecture are those with more computation enclosed and those that are iterated more times. The quantity of computation can be expressed in terms of the number of FSMD scheduled states.

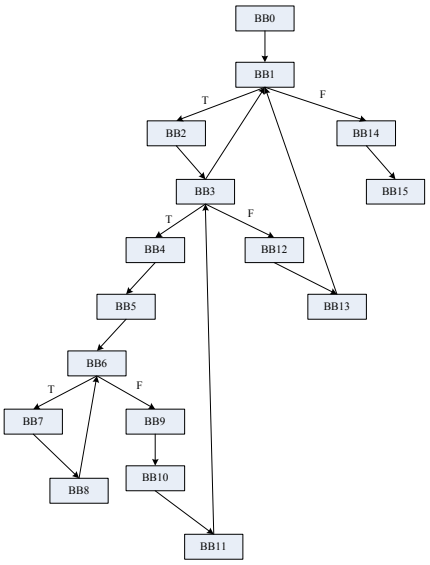

Figure 4. Control flow of DCT code

The basic block with the most computation is BB7 as it is scheduled in 7 or 8 FSMD states. It represents the body of the innermost loop (*Line08*):

```
sum = sum + A[i][k]*B[k][j]
```

Basic block BB10 is scheduled in 4 FSMD states and represents the middle loop body other than the innermost loop (*Line10*):

```
C[i][j] = sum
```

Their contents in CDFG view are presented in Fig. 5. All other basic blocks are too simple to analyze. They consist only of one assignment, increment or comparison operation, and they are therefore scheduled in 1 FSMD state. Their demand on the datapath is one comparator and one adder.

As presented in Fig. 5, the temporary variables *Tx* are used to break the complex expressions and complete the form of three-address statement. The schedules of basic block BB7's CDFG (in Fig. 5(a)) are listed in Fig. 6 under data memory ports constraints. On assumption of 1-port data memory the code is scheduled in 8 states (a), and on assumption of 2-port data memory the code is scheduled in 7 states (b).

The allocation and binding phase picks appropriate functional units and bind them to three-address statements. For schedule in Fig. 6(a) there are maximums of 2 additions and 1 multiplication operation per state which is reflected in the demand for two adders and one multiplier instance in the datapath. In Fig. 6(b), the schedule is shorter, but 2 adders and 2 multipliers are required to encompass it. All variables are bound to registers taking care that variables that are used in the same states do not share a register. For instance, variables *T3*, *j*, *A* and *T7* are used only as inputs for additions in state S3 in Fig. 6(a), and are not supposed to share the register. On the contrary, variable *B* is used only in state S4 and is allowed to share a register with any variable used in state S3.

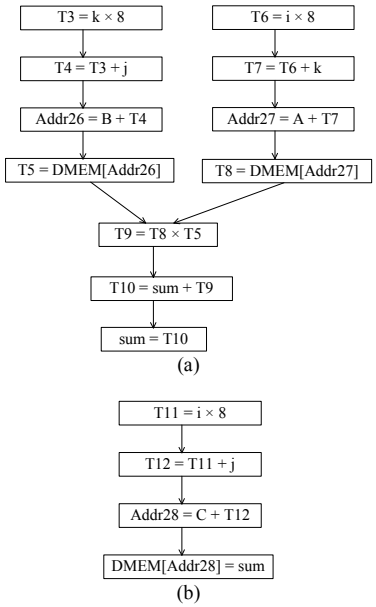

Figure 5. Data flow of basic blocks: (a) BB7, (b) BB10

State	Scheduled statements
S1	T6 = i × 8
S2	T3 = k × 8 T7 = T6 + k
S3	T4 = T3 + j Addr27 = A + T7
S4	Addr26 = B + T4 T8 = DMEM[Addr27]
S5	T5 = DMEM[Addr26]
S6	T9 = T8 × T5
S7	T10 = sum + T9
S8	sum = T10

(a) 1-port data memory

State	Scheduled statements
S1	T3 = k × 8 T6 = i × 8
S2	T4 = T3 + j T7 = T6 + k
S3	Addr26 = B + T4 Addr27 = A + T7
S4	T5 = DMEM[Addr26] T8 = DMEM[Addr27]
S5	T9 = T8 × T5
S6	T10 = sum + T9
S7	sum = T10

(b) 2-port data memory

Figure 6. FSMDs for basic block BB7 of DCT code

The schedule of BB10's CDFG (in Fig. 5(b)) is not dependent on data memory ports constraint. It is always scheduled in 4 states and demands 1 adder and 1 multiplier instances. These demands are the subset of basic block BB7's demands and thus are satisfied if the datapath is fully customized to BB7. The demands of other basic blocks are also encompassed within those of BB7 with the exception of a comparator functional unit demanded by branch_condition basic blocks BB1, BB3 and BB6.

V. Results and Discussion

The datapath for Listing 1 DCT code with the assumption of 1-port data memory is formed as shown in Fig. 7. There are 2 adder instances, 1 multiplier and 1 comparator functional unit instances. All adder and multiplier instances are not used simultaneously so there is one combined adder/multiplier unit. The optimizations during allocation and binding phase derived such a datapath to save implementation platform resources. There are 4 registers with contents listed in Table I. For instance, the multiplication scheduled in state *S1* is executed by the functional unit *FU1*. Variable *i* is read from register *Reg2*, constant *8* is read from register *Reg1*, and the result is stored as variable *T6* in register *Reg2*. By taking into consideration the schedules and iteration counts of basic blocks there is a total of 6314 execution cycles of DCT code with this datapath which is almost 40% less than for MIPS style datapath used in [14].

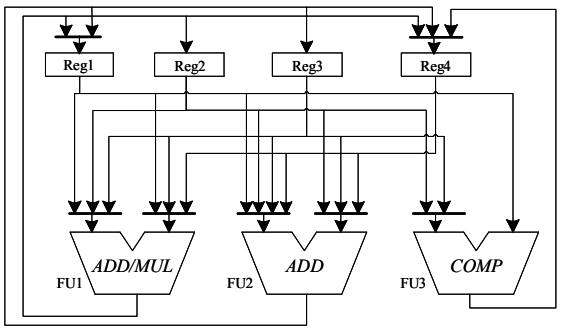

Figure 7. The datapath for DCT code (with 1-port data memory)

The datapath that is produced for DCT with the assumption of 2-port data memory has two combined adder/multiplier functional units and a comparator. The difference from the one with 1-port data memory constraint is in another multiplier instance combined with adder of *FU2*. With this enhancement the DCT is executed in 5802 cycles which is close to 45% improvement over MIPS style datapath used in [14].

The Unrolled DCT case in Listing 2 has much more parallelizing opportunities than the original DCT code and the resulting custom datapaths much differ under different data memory ports constraints. There is only one loop so CDFG has one complex basic block that represents the loop body (lines from *Line03* to *Line20* in Listing2), while the other basic blocks represent loop initialization, condition test and increment.

The preliminary results show higher occupations of datapath components than for manual design [14] as there were no such constraints. Table II shows a rough comparison of manual (Fig. 1) and both cases of automated designs. It is expressed in allocations of datapath components, while the estimation of implementation resources occupation is currently beyond the scope of this work. The 2-port memory allows more

TABLE I. Contents of Registers for Fig. 7 Datapath

Register	Variables and constants
Reg1	*A*, 8, *Addr26*, *Addr27*, *T10*, *T11*
Reg2	*B*, *i*, *T3*, *T6*, *T8*, *T9*
Reg3	*sum*, *j*, *k*, *T5*, *T12*
Reg4	*C*, *Addr28*, *T2*, *T4*, *T7*

TABLE II. COMPARISON OF MANUAL AND AUTOMATED DESIGNS FOR UNROLLED DCT CASE

Evaluation metric		Manual design [14]	Automated design	
			1-port Data Memory	*2-port Data Memory*
Resource occupation	#FUs	5	6	8
	#Regs	8	16	20
	#Conns	14	52	61
# Exec. cycles		3040	1665	1217

parallelized execution which is reflected in a higher number of functional units in datapath and in a lower number of execution cycles. For 1-port data memory case one pass of loop body is scheduled in 24 states, and for 2-port data memory it is scheduled in 17 states. That is reflected in the number of total execution cycles. The total cycle count for 1-port memory is 1665, and for 2-port memory it is 1217, which is 40-60% less than for manual design [14]. The tool produced presented designs in a few seconds, while for manual design we do not have the exact information.

VI. CONCLUSION

In this paper, we have presented automated design of custom processor architecture for Discrete Cosine Transform - DCT algorithm. Such architecture is purposed as input for the No-Instruction-Set computer concept of high-level synthesis where the design is synthesized according to processor architecture and input C code application.

Previous work elaborated the design space exploration by looking into the features of input C code and manual customization of processor architecture. Here, we have automated the construction of processor with a specific tool. The tool analyzes the control and data dependencies of the code, schedules the operations and allocates the registers and functional units. After that, it binds architectural components to three-address code expressions representing the input algorithm code, and applies optimizations to avoid redundancies in the design. The methodology does not apply code optimizations, but accepts the code as it is. The optimizations are applied at the architectural level resulting that different codes performing the same functionality have different custom architectures.

The exploration of custom processor structure is automated with respect to achieving a minimal number of execution cycles. The results show that the achieved execution cycle counts are within the same order of magnitude as those obtained by manual design, while the design time is significantly shortened. The drawback of the presented results is in excessive datapath components allocations as there were no explicit components constraints (except those related to the number of ports in data memory cores). Thus, the design is fully optimized in terms of execution cycles, but takes a considerable amount of implementation area.

ACKNOWLEDGMENT

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).

REFERENCES

[1] G. E. Moore, “Cramming More Components onto Integrated Circuits,” in Electronics, vol. 38, no. 8, pp. 114–117, April 1965.

[2] W. Ecker, W. Muller, R. Dömer, Hardware-dependent Software: Principles and Practice, Springer, 2009, pp. 1-6.

[3] D. D. Gajski, A. Gerstlauer, S. Abdi, G. Schirner, Embedded System Design, Springer, 2009, pp. 199-254.

[4] TIOBE Programming Community Index, http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

[5] How popular are various programming languages?, http://www.complang.tuwien.ac.at/anton/comp.lang-statistics

[6] C to HDL, http://en.wikipedia.org/wiki/C_to_HDL

[7] M. Reshadi, B. Gorjiara, D. D. Gajski, “NISC Technology and Preliminary Results,” Technical Report, University of California, Center for Embedded Computer Systems, Irvine, August 2005.

[8] D. Ivosevic, V. Sruk, “Evaluation of embedded processor based BDD implementation,” in Proceedings of the 33rd International Convention MIPRO, pp. 619-623, 2010.

[9] Riverside Optimizing Compiler for Configurable Computing (ROCCC) 2.0, http://roccc.cs.ucr.edu

[10] Xtensa Customizable Processors, http://www.tensilica.com/products/xtensa-customizable.htm

[11] S. Asano, T. Maruyama, Y. Yamaguchi, “Performance comparison of FPGA, GPU and CPU in image processing,” in Proceedings of International Conference on Field Programmable Logic and Applications, pp. 126-131, 2009.

[12] S. A. Khayam, “The Discrete Cosine Transform (DCT): Theory and Application,” Department of Electrical & Computer Engineering, Michigan State University, DCT Tutorial, WAVES-TR-ECE802.602, March 2003.

[13] G. Aggarwa, D. D. Gajski, "Exploring DCT Implementations," UC Irvine, Technical Report ICS-TR-98-10, March 1998..

[14] B. Gorjiara, D. D. Gajski, “Custom Processor Design Using NISC: A Case-Study on DCT algorithm,” in Workshop on Embedded Systems for Real-Time Multimedia, pp. 55-60, 2005.

[15] J. Trajkovic, D. D. Gajski, “Custom Processor Core Construction from C Code,” in Proceedings of Sixth IEEE Symposium on Application Specific Processors, Anaheim, California, June 2008.

[16] D. D. Gajski, N. D. Dutt, A. C-H Wu, S. Y-L Lin, “High-Level Synthesis: Introduction to Chip and System Design,” Springer, 1992.

[17] P. Coussy, D. D. Gajski, M. Meredith, A. Takach, “An Introduction to High-Level Synthesis,” in IEEE Design & Test of Computers, vol. 26, no. 4, pp. 8-17, July-August 2009.

[18] A. Orailoglu, D. D. Gajski, “Flow graph representation,” in Proceedings of the 23rd ACM/IEEE Design Automation Conference, pp. 503 - 509, 1986.

[19] SPARK: A Parallelizing Approach to the High-Level Synthesis of Digital Circuits, http://mesl.ucsd.edu/spark, University Of California, San Diego