Impact of Control Flow Blocks Granularity on Custom Processor Design Time

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

Abstract - One of the key issues for system level design topic is the design time. This paper describes custom processor design tool as part of C-to-hardware flow and analyses its design time. The flow starts with C code specification and ends with FPGA implementation. The way the C code is processed has impact on the flow execution time. The implemented C code processing results with Control Flow Graph (CFG), and large control flow code blocks severely prolong the overall design time. Between two possibilities for design time improvement, variations in their granularities are chosen over tool internal algorithm and data structures optimizations. For 32-point DCT test case the results show huge design time decrease at the expense on the design quality: implementation resource occupation and execution time.

I. INTRODUCTION

Embedded System Level (ESL) design is an emerging methodology for custom digital system design with focus on higher level design specification. With constant rise of capability of technology and the growth of software and hardware productivity gaps the needs for quality and reliable software support increase, [1].

Software solutions help in area of Electronic Design Automation (EDA) tools development and IP cores production that helps in closing of hardware productivity gap.

This work is motivated by this challenge and deals with the topic of custom processor architecture design from C code specification. The specification in C code and its derivatives, C++ and SystemC, is dominant in High-Level Synthesis (HLS) which usually produces hardware description in RTL code that is ready for logic synthesis targeting FPGA device. The research of this methodology had been the most intensive in academic society in past decades, and, during the time, a number of commercial tools appeared in the market, [2,3].

In this work, one of the production targets is a custom processor architecture model designed for C code specification. The model of architecture is designed according to No-Instruction-Set Computer (NISC) concept, [4]. In this concept, the system designer can arbitrarily choose datapath components having in mind the C application code features. On the contrary, in this paper the task and methodology of automated datapath design is presented. The resulting architecture that contains such automatically designed datapath is in final implemented in FPGA. The methodology assumes no compiler-style optimizations, but fully customizes the architecture. Thus, the emphasis is on optimizations that are applied on datapath level. In such way, the optimizations are closer to the implementation platform while the concept retains processor style of execution familiar to common user. The traditional HLS flow [5] is broken into two stages: processor as the execution engine, and its mapping to the implementation platform.

In previous works custom datapath design for such processor model is analyzed through manual datapath transformations in several iterations [6] and its automated construction [7]. Besides these, we elaborated manual and automatic datapath generation for specific BDD application with highly recursive nature [8] and DCT application code [9].

In the following sections, Section II and Section III, the overview and methodology of implemented design flow is presented. Section IV presents the results for several test cases, and Section V focuses on design time for 32-point DCT case which appeared to be time demanding. Section VI concludes on the presented methodology and results.

II. IMPLEMENTED DESIGN FLOW

Design flow is characterized by several processes, and their inputs and outputs. Globally, there are three major processing steps in the design flow, Fig. 1:

1. C code preprocessing.
   Code is analyzed by its procedures and basic blocks formed by procedures control flows. Resulting notation is Control and Data Flow Graph (CDFG), [10]. SPARK parallelizing compiler is used for initial transformation of C code to CDFG. Further it is altered by our tool to conform the later stages of design flow, [11].
2. Architecture build.
   As the most complex process of the flow it is implemented within our *ArkBuilder* tool. Three separate processes can be identified:
   a. Scheduling of CDFG three-address code statements.

b. Operand and operations usage analysis. Such analysis produces combinations of operands and operations grouped within registers and functional units. Thus, the simplified (or provisional) datapaths consisting only of register files, functional units and their connections are formed for every basic block.
c. Design of final datapath. The design is based on integrating all basic blocks datapath contributions. The result is datapath completed with data memory, multiplexers as arbitral components instanced at other components inputs, and connections interfacing the control unit.

3. Simulation/Implementation.
Specially designed tools *GenCM* and *ProcSynth* generate design RTL description and instruction memory initialization file for core generation. With inclusion of appropriate data memory core the design is synthesized and implemented in FPGA. The behavioral simulation is used for verification purposes. The synthesis, implementation and simulation steps are accomplished with *Xilinx ISE* toolset, [12].

The central point of the flow is the algorithm for final datapath design. It forms the complete architecture that logically corresponds to CDFG 'per basic block' schedules. The datapath contributions of all basic blocks are integrated into final datapath having in consideration basic blocks significance. The significance is defined by their execution cycles shares inside whole application run. Such shares are estimated by profiling of procedures and basic blocks, and basic blocks schedule lengths. Basically, the algorithm for final datapath design is provided with following input information:

- CDFG application description
- Application code profiling information
- Implementation platform description

Here introduced aspects of datapath design are described in following sections using short example to clarify the theoretic information.

III. Processor Design Methodology

The code profiling is performed at two levels of abstraction: procedure level and basic block level. At final, it is flattened at basic block level as number of procedure calls is multiplied with basic block iteration count to get the absolute basic block iteration count.

The code analysis is thus performed for each basic block independently. As the original code is transformed in basic block three-address code notation, its mapping to architecture datapath is straightforward. Functional units with two input ports and one output port correspond to the three-address code notation of basic block statements. Scheduling, allocation and binding tasks are performed on basic block three-address statements. Firstly, the scheduling of statements produce finite state machine with data (FSMD). The analysis of statement operands and operation is performed for every cycle and their non-overlapping usages are noted to allocate registers and

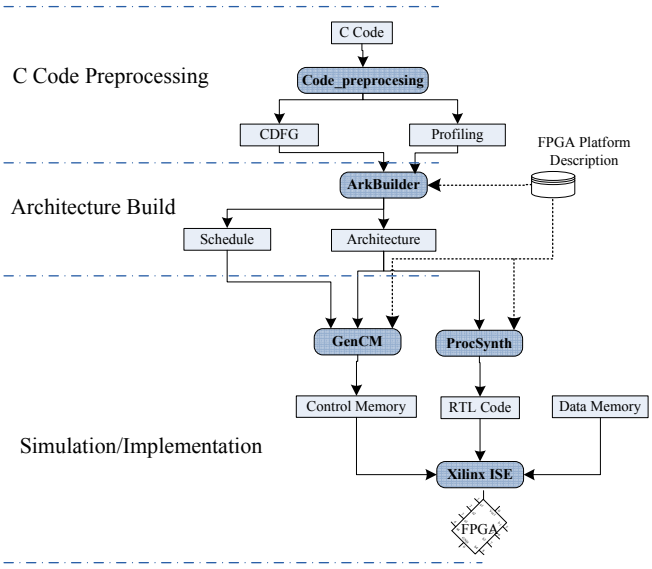


Fig. 1. Design synthesis flow.

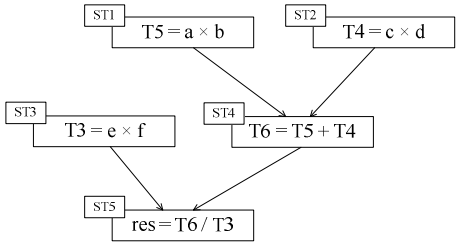


Fig. 2. Data flow graph for expression in (1)

functional units, respectively. In such way, the binding of statements to registers and functional unit is implicit.

A. Basic Block Analysis

As stated in Section II there are three steps in the methodology of architecture build. First two steps, scheduling and basic blocks analysis with forming of simplified datapaths, are performed for every basic block of the application CDFG. The last, final datapath design combines all basic blocks datapaths into unique one and interfaces it to control unit. Here, the methodology is going to be elaborated through simple line of code in (1) assumed to be only basic block content. It is a simple calculation consisting of three multiplications, an addition and a division.

$$g = ((a \times b) + (c \times d)) / (e \times f); \quad (1)$$

Fig. 2 shows data flow graph of (1) consisting of five three-address code statements: ST1 to ST5. Data dependencies are properly extracted to ensure the regularity of calculation. According to those dependencies, the scheduling phase produces finite state machine as in Fig. 3a. Usage analysis of operands and operators notes operand usages and operation activity per state, as in Fig. 3b and 3c.

If one register per operand and one functional unit per operation would be assumes, there would be much

State	Statements	Three-Address Code
S1	ST1	T5 = a × b
	ST2	T4 = c × d
S2	ST3	T3 = e × f
	ST4	T6 = T5 + T4
S3	ST5	res = T6 / T3

a)

Operation	S1	S2	S3
ADD	0	1	0
MUL	2	1	0
DIV	0	0	1

b)

Operand	S1	S2	S3
a	×		
b	×		
T5		×	
c	×		
d	×		
T4		×	
e		×	
f		×	
T3			×
T6			×

c)

Fig. 3. Scheduling (a), and operation (b) and operand usage analysis (c) for (1)

redundancies in the datapath. Therefore, minimization of registers and functional units allocations is applied using compatibility graphs, [13]. For instance, for operands that are not used in same states the priority edges are defined with *i/o* priority notes, Fig. 4a. Value *i* is the number of same operation types for which corresponding operands are input values, and *o* is the number of the same operation types for which they are outputs. For operand used in the same state the incompatibility edges are defined, Fig. 4b. By merging operands with priority edges between, final compatibility graph is derived having only incompatibility edges, Fig. 4c. Operands that are merged to the same node share a register. In the same manner, the combinations of operations are dedicated to functional units. Using this technique, final datapath for expression (1) consists of two functional units and four registers appropriately connected, Fig. 5.

B. Final Datapath Design Algorithm

Algorithm that designs the final datapath integrates contributions of all basic blocks provisional datapaths. The pseudo code that describes it, is following:

$DP = \emptyset$
$FUmax_{op}$ = GET_MAX_INSTANCES(CDFG),
$op \in$ {ADD, SUB, …, ASSIGN}
for $bb_{current} \in CDFG$
  FU_{sorted} = SORT_FU ($bb_{current}$, desc)
  for $fu_{current} \in FU_{sorted}$
    Op= GET_OPERATION($fu_{current}$)
    if $FUinstances_{op} < FUmax_{op}$
      $ACCOMP_{fucurrent}$ = ACCOMP_LOGIC($fu_{current}$)
      $DP \mathrel{+}= fu_{current} + ACCOMP_{fucurrent}$

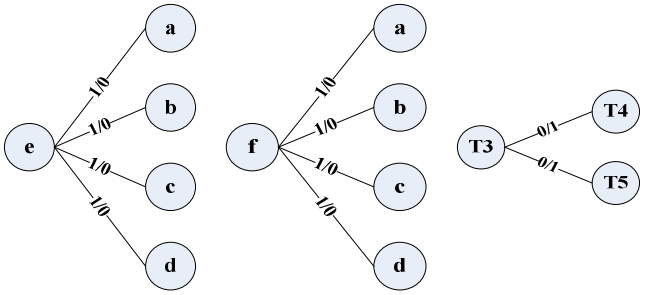
It is based on analysis of maximum needed operation instances for all basic blocks schedules. The contributions of all basic blocks are integrated to the final datapath through their functional units and all components that are connected to those functional units. When number of particular operation instances implemented within existing functional units in the final datapath is exceeded, those types of functional units are not further instanced.

As first, function GET_MAX_INSTANCES() analyses operation usages per basic blocks schedules. As result, maximum numbers of operations per scheduled state is noted - $FUmax_{op}$. Basic blocks are traversed and functional units are sorted by usage frequencies and integrated into final datapath along with their accompanying logic ($ACCOMP_{fucurrent}$); register files, multiplexers and connections. Data memory is instantiated for data arrays used in input C code. Before integration of new functional unit to the final datapath *DP*, number of functional unit instances per operation type is checked against calculated $FUmax_{op}$ value.

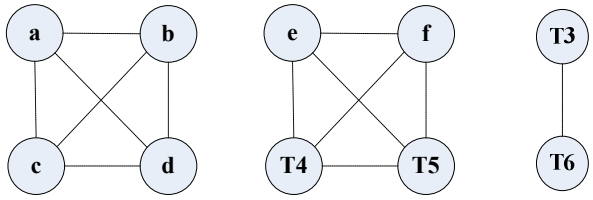
IV. PRELIMINARY RESULTS

Our approach of processor architecture modeling and implementation is tested on following C coded algorithms used inside NISC toolset, [14]:

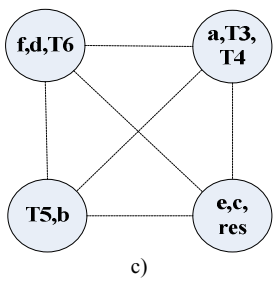
- Discrete Cosine Transform (DCT) on 8×8 matrices in two versions: original (with three nested loops) and unrolled, [6].
- 32-point DCT used in MP3 decoder.
- SHA-1 encryption algorithm, [15,16].



a)



b)



c)

Fig. 4. Initial priority edges (a), incompatibility edges (b) and final compatibility graph (c) for operands

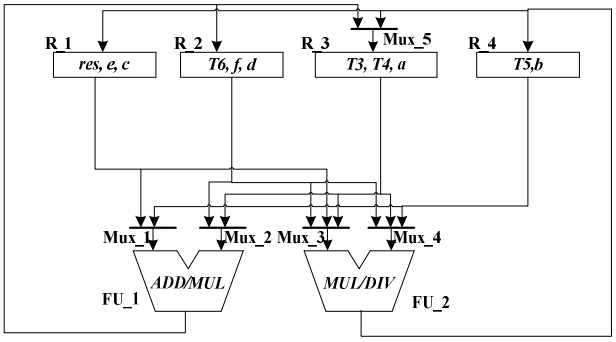


Fig. 5. Final datapath for expression in (1).

TABLE I. TEST CASES CDFG CHARACTERISTICS

Case	CDFG Characteristics			Design Time	Exec. Time / µs
	#blocks	*#statements*	*#operands*		
DCT 8×8	15	28	29	< 1 s	190,0
Unrolled DCT 8×8	5	161	191	1 s	54,9
32-point DCT	1	791	890	25,8 h	8,2
SHA-1	31	158	150	1 s	67,6

Table I shows characteristics of test cases CDFGs: numbers of basic blocks, statements and operands. The first two, DCT and Unrolled DCT represent the same code written in different styles and elaborated in [6].

Table I also shows processor design times and their FPGA implementation execution times. While other three cases have design times of one second or less, 32-point DCT has design time of 25,8 hours. The more detailed analysis shows that the most of it, more than 99%, is spent in stage of operands usage analysis and optimizations, i.e. in handling operand compatibility graphs. Therefore, in next section we focus on minimization of this case design time.

When execution times are compared to those of NMIPS RISC architecture model, the baseline in [6], the execution times are very similar (150,2 µs for DCT 8×8, 51,6 µs for Unrolled DCT 8×8, 11,5 µs for 32-point DCT and 64,3 µs for SHA-1). The implementations of the same applications on embedded Microblaze processor [17] are much slower (516,2 µs for DCT 8×8, 932,5 µs for Unrolled DCT 8×8, 280,4 µs for 32-point DCT and 670,9 µs for SHA-1), and much faster when designed with Vivado HLS tool (9,4 µs for DCT 8×8, 2,0 µs for Unrolled DCT 8×8, 0,5 µs for 32-point DCT and 6,5 µs for SHA-1), [18].

V. CASE STUDY: 32-POINT DCT

32-point DCT used here is optimized version of such transformation, but here it is the largest code without control flow dependencies. Inside only one basic block there are total of 80 additions, 119 subtractions, 80 multiplications and 49 shifts, [19, 20].

There is total of 209 lines of code (LoC) in C code specification. There are eight versions of code granulations undertaken to check the design time change. Starting with original input code (*cdfg0*), the code is granulated from the perspective of the implemented flow user. In *cdfg1* the code is split in two, in *cdfg2* in three parts, etc. The extreme situation is when every line of code is a separate block (*cdfg7*). Table II summarizes features of all CDFG versions in aspects of three-address statements, operands and scheduled cycles per block. More detailed view on granulation points in input C code is illustrated in Fig. 6 which depicts relations between granulation steps.

The comparison of designed datapath and implementation results is presented in Table III. The granulation of code to smaller portions significantly impacts the design time. After two steps of code granulation design time falls to below 20 minutes, and after following two it falls below a minute. The side-effect of such granulation sequence is architecture datapath and FPGA implementation resources occupations growths. This is caused by the fact that final datapath algorithm limits the functional units instantiation, but it is not the case for register files. Therefore, the FPGA resource occupations significantly rise and, in the same time, performance drops. The size problem escalates for *cdfg6* and *cdfg7* for which design could not be synthesize on Virtex-5 SX50T device that was targeted.

VI. CONCLUSION

This paper presents C-to-hardware design flow where processor architecture abstraction level is kept as important abstraction in design representation. The processor is modeled as No-Instruction-Set Computer with fully custom datapath. The datapath is customized according to scheduled three-address code, operand and operation usage analysis and their optimized binding to register files and functional units.

The preliminary results show successful FPGA implementation with performance in a range or better than processor based system design, but worse than high-level synthesis tool. The design time as the key strength in system level design appeared to be too high for test case without control flow dependencies and large number of lines. The processing of such code, 32-point DCT, took

TABLE II. 32-POINT DCT CDFGS FEATURES

Case	#blocks	#statements /block	#operands /block	#cycles /block
cdfg0	1	791	890	418
cdfg1	2	352, 459	445	177, 242
cdfg2	3	256-269	269-355	129-150
cdfg3	5	96-269	100-269	50-150
cdfg4	11	24-159	24-159	14-90
cdfg5	19	24-87	24-159	14-50
cdfg6	46	7-47	7–53	5-26
cdfg7	209	1-20	3–21	1-15

TABLE III. 32-POINT DCT DESIGNS CHARACTERISTICS

Case	Datapath Features		Design Time	FPGA Resource Occupation		Performance	
	#FUs	*#RFs*		*#Slices*	*#DSPs*	*#Cycles*	*Frequency / MHz*
cdfg0	5	29	25,8 h	1577	3	426	52,018
cdfg1	30	98	1,3 h	3353	3	433	46,517
cdfg2	29	117	17,2 min.	4107	3	439	45,486
cdfg3	26	107	6,3 min.	4780	3	454	41,075
cdfg4	14	106	41 s	5052	3	501	40,098
cdfg5	12	104	5 s	5681	3	559	35,088
cdfg6	17	147	2 s	N/A	N/A	N/A	N/A
cdfg7	4	149	5 s	N/A	N/A	N/A	N/A

hours of time. There were two options to shorten the design time: improvements in data structures and algorithms implemented inside tool or reorganization of input C code by breaking it into smaller portions. From the perspective of system designer that uses the implemented flow the latter, i.e. the specification alteration, is only possible impact on design time.

The changes in input code granulation significantly shorted the design time. As the code was granulated to smaller portions, the design time rapidly decreased until the level of only few seconds. The side-effect of such code granulation was in the growth of resulting datapaths and, consequently, the target FPGA device occupation. Also, the performance dropped twice.

Therefore, the future work in sense of shortening design time has following options: better algorithmic manipulation and usage of more appropriate data structures for storing compatibility graphs, and more intelligent C code preprocessing when large CDFGs are produced. Also, the consequential design size growth demands the control over register files datapath instantiations.

ACKNOWLEDGMENT

This work was supported by research grant No. 036-0362980-1929 from the Ministry of Science, Education and Sports of the Republic of Croatia. Special thanks we give to Center for Embedded Computer Systems (CECS) at University of Irvine California as the key support was based on prof. Daniel Gajski's team research and experience.

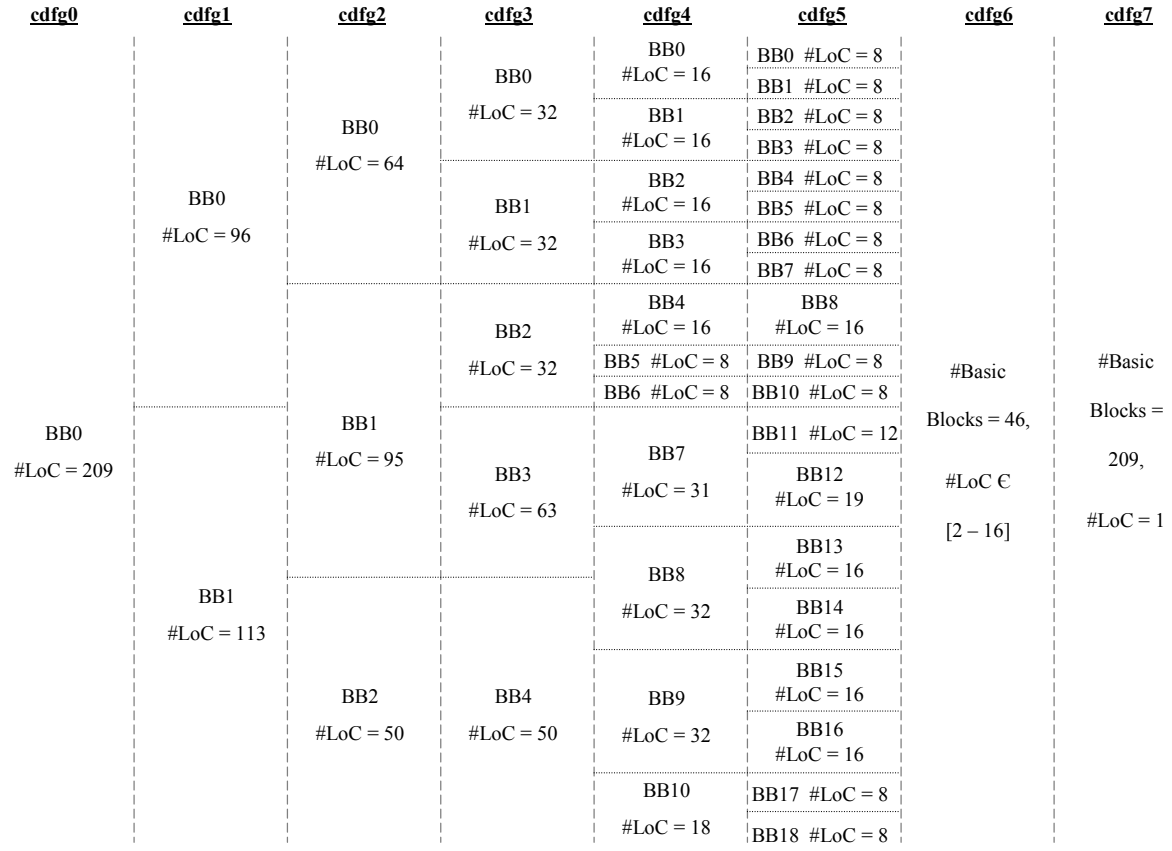


Fig. 6. Overview of 32-point DCT code granulation

REFERENCES

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

[2] (2013) C to HDL. [Online]. Available: http://en.wikipedia.org/wiki/C_to_HDL/

[3] (2013) Gary Smith EDA. [Online]. Available: http:// http://garysmitheda.com/

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[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] D. Ivosevic, V. Sruk, "Automated modeling of custom processors for DCT algorithm," in Proceedings of the 34th International Convention MIPRO, pp. 762-767, 2011.

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

[11] (2013) SPARK: A Parallelizing Approach to the High-Level Synthesis of Digital Circuits. [Online]. Available: http://mesl.ucsd.edu/spark

[12] (2013) ISE WebPACK Design Software. [Online]. Available: http://www.xilinx.com/products/design-tools/ise-design-suite/ise-webpack.htm

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

[14] (2013) NISC Technology – Toolset online demo. [Online]. Available: http://www.ics.uci.edu/~nisc/toolset/

[15] (2013) National Institute of Standards and Technology (NIST), "Secure Hash Standards (SHS)". [Online]. Available: http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf

[16] (2013) C to Verilog. [Online]. Available: http://c-to-verilog.com/howtos.html

[17] (2013) MicroBlaze Soft Processor. [Online]. Available: http://www.xilinx.com/tools/microblaze.htm

[18] (2013) C-based Design: High-Level Synthesis with Vivado HLS. [Online]. Available: http://www.xilinx.com/training/dsp/high-level-synthesis-with-vivado-hls.htm

[19] (2013) mbed: i2s_audio_madplayer. [Online]. Available: http://mbed.org/users/okini3939/code/i2s_audio_madplayer/docs/30b2cf4a8ee2/synth_8cpp_source.html

[20] (2013) libmad - MPEG audio decoder library. [Online]. Available: ftp://ftp.icsi.berkeley.edu/global/pub/speech/software/praatlib-0.3/src/mp3/mad_synth.c