Unified Flow of Custom Processor Design and FPGA Implementation

Danko Ivošević, Vlado Sruk

Faculty of Electrical Engineering and Computing, University of Zagreb
Department of Electronics, Microelectronics, Intelligent and Computer Systems
Unska 3, Zagreb, Croatia
danko.ivosevic@fer.hr
vlado.sruk@fer.hr

Abstract— **The automation of custom hardware design often focuses on hardware optimizations for smaller portions of code that dominate the design execution. The same presumption can be stated for custom processor design. The data path of the processor can be well optimized for particular blocks of code that are formed during control flow extraction. However, larger source codes can have tens of blocks that result from Control Flow Graph (CFG). We implemented a global semi-automated flow that hierarchically forms the set of blocks which contributions are modeled into processor architecture. Resulting processor model is translated to RTL description and implemented inside FPGA logic.**

Keywords: Custom Processor Design, No-Instruction-Set Computer, High-Level Synthesis, Data Path Design, FPGA Implementation

I. Introduction

Embedded System Level (ESL) design is an emerging methodology for custom digital system design with focus on higher level design specification, [1]. 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, Fig. 1.

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

This work is motivated by this challenges and deals with 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) tools. Such tools usually produce hardware description in RTL code that is ready for logic synthesis targeting FPGA device. Although gate-level synthesis designs usually have higher performance, HLS strengths are shorter design time and availability for more users. The research of HLS methodology lived its zenith in academic societies in past decades. During the time, a number of commercial tools appeared in the market, [3,4].

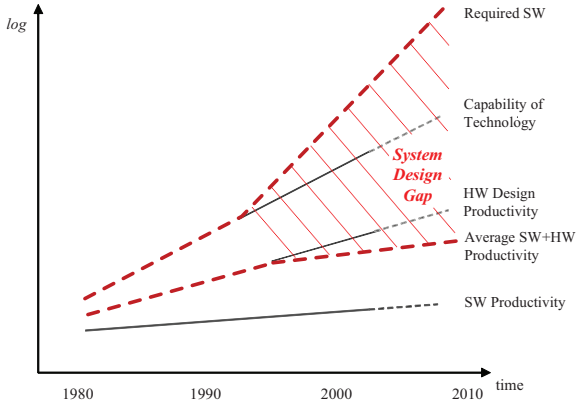

Fig. 1. HW and SW productivity gap (source: [2])

In this work, we turned to processor architecture model as custom product for C code specification. The model of architecture is designed according to No-Instruction-Set Computer (NISC) concept, [5]. In this concept, the system designer can arbitrarily choose data path components having in mind the C application code features. On the contrary, in this paper the task and methodology of automated data path design is presented. The resulting architecture that contains such automatically designed data path is in final implemented in FPGA device. The methodology assumes no compiler-style optimizations, but fully customizes the architecture. Thus, the emphasis is on optimizations that are applied on data path 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 is broken into two stages: processor as the execution engine, and its mapping to the implementation platform.

In the following subsections, overview of high-level synthesis and processor based design are described as the key concepts of this paper. The complete custom processor design flow is elaborated in Section II. In Section III and Section IV the important features and results of several test cases are presented. Section V gives final conclusions on the presented work.

A. Traditional High-Level Synthesis

High-level synthesis is referred in the literature as C synthesis, or algorithmic synthesis. It is an automated process of hardware design from algorithmic specification. This process analyzes the input code and produces RTL schedule conducted by the architectural constraints. The logic synthesis process finally translates the RTL description into *bitstream*

978-1-4673-2232-4/13/$31.00 ©2013 IEEE

for target implementation. Lifting the specification level to algorithmic representation means better control over design, optimizations and verification at RTL level. The popularity and flexibility of C programming language and design flows integration capabilities were decisive for its usage in most of high-level synthesis flows.

Traditional HLS flow has a number of common steps, [6]:

- Lexical processing – usually translates the source code to a form of intermediate representation. It has many similarities to higher level languages compilers. Some of them are used in actual design flows for translation to standardized intermediate representations.
- Algorithm optimizations – also assumes some of the higher level compiler functionalities in a sense of code optimizations and parallel execution expression.
- Control and data dependencies analysis – deals with operation inputs and outputs recognition and data dependencies expression. Operations are analyzed in three-address code notation that is generated without any timing dependencies.
- Technology library processing - introduces implementation technology description in sense of functional, timing and allocation characteristics.
- Resource allocation – determines implementation sets of functional units from components library.
- Operation scheduling – based on data dependencies and functional unit latencies produces finite state machines. Principles of scheduling can be time- or resource-constrained.
- Functional units and register bindings – dedicates operations to functional units and operand to registers.
- Output processing – generates RTL code that implements finite state machine ready for logic synthesis.

B. Processor Based Design

There are several embedded processors that are engaged in FPGA design flows. Xilinx offers easy-to-use soft-core MicroBlaze processor within its Embedded Development Kit, [8]. It is general-purpose 32-bit processor with instruction set architecture similar to RISC-based DLX architecture, [9]. Customizable aspects of this core are: cache size, pipeline depth and memory management. Some operations, such as multiplications, divisions and floating point operations can be implemented in hardware, and performance tuning is possible through standard gcc compiler optimizations. Besides that, there are other soft and hard embedded processor cores such as Xilinx PowerPC, Altera Nios or Altium TSK 51/52 and 3000A within their well-known integrated development suites.

Concepts of Application-Specific Instruction Processor (ASIP) with higher grade of customization with instruction set extensions, such as Xtensa, [10]. In a sense of data path customization there is NISC toolset. The concept of this toolset allows full customization of data path and program words that make up the control unit. There is no predefined instruction set as it is formed according to data path contents. The architecture is translated into synthesizable Verilog code targeted for FPGA implementation. Data path is fully customized by user or fully determined by algorithmic specification. In this paper we describe implementation of automatic data path customization for C algorithm specification. It is motivated by NISC where the control logic, that reads the custom instruction memory and issues the appropriate signals to the data path, is fixed. Related to this, the previous work for custom co-processor design focuses on iterative algorithm for resource usage optimization, [11]. Similar concept is Transport Triggered Architecture (TTA) custom processor which is scalable with respect to instruction-level parallelism and focused on customizable interconnect network, [12].

II. Design Flow

The flow of FPGA implementation from C code with processor architecture model in-between is supported by several software tools. In this section, the overview of the flow and important aspects; code profiling, implementation platform description and data path design, are presented.

A. Overview

Design flow is characterized by several processes, and their inputs and outputs. Globally, there are three major processing steps in the design flow, Fig. 2:

- C code preprocessing. Code is analyzed by its procedures and basic blocks formed by the control flows of the procedures. Resulting notation is Control and Data Flow Graph (CDFG). SPARK tool is used for initial conversion of C code to CDFG.
- Architecture build. As the most complex process of the flow it is implemented within standalone *ArkBuilder* tool. Three separate processes can be recognized. The first is scheduling of CDFG three-address code statements followed by operand and operations usage analysis. Such analysis produces combinations of operands and operations encapsulated within registers and functional units. Thus, the simplified (or provisional) data paths consisting only of register files, functional units and their connections are formed for every basic block. The last step is the design of final data path. The design is based on integrating all basic blocks data path contributions. The result is data path completed with data memory, multiplexers as arbitral components, and connections interfacing the control unit.
- Simulation/Implementation. Separate 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 relying on generated IP cores is used for verification purposes.

The central point of the flow is the algorithm for final data path design. It forms the complete architecture that logically corresponds to CDFG 'per basic block' schedules. The data path contributions of all basic blocks are thus integrated into the final data path. The significance of basic blocks are

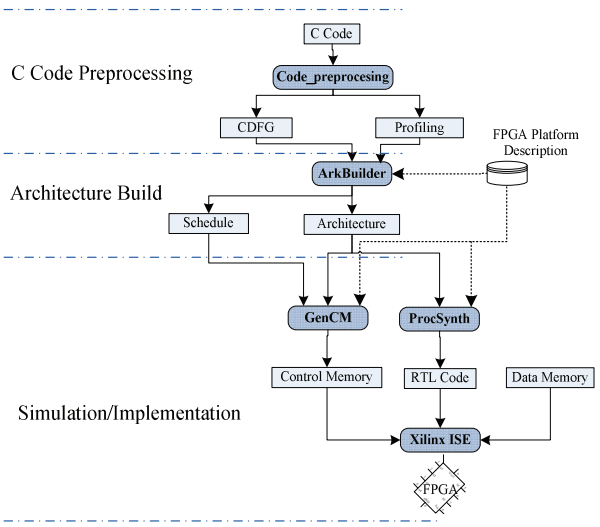

Fig. 2. Design synthesis flow.

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 data path design is provided with following input information:

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

Here introduced aspects of data path design are described in following subsections using short example to clarify the theoretic information.

B. Two-Level Profiling

The code profiling is performed at two levels of abstraction:

- Procedure level
- Basic block level

However, the profiling is basic block based as the 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 data path 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 as output produce finite state machine with data (FSMD). The analysis of statement operands and operation is performed by each cycle and their non-overlapping usages are noted to allocate registers and functional units, respectively. In such way, the binding of statements to registers and functional unit is implicit.

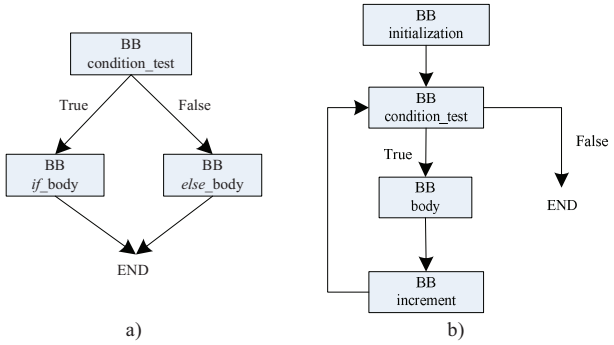

Fig. 3. Control flow constructs of: a) conditional branch, and b) loop

```
proc1    // calls = 5
{
   // no branches or loops
}

proc2    // calls = 4
{
   for (i = 0; i < 10; i++)
   {
      // loop body
   }
}

proc3    // calls = 3
{
   for (i = 0; i < 20; i++)
   {
      if (test_value < cond_value)
      {
         // entered in 40% of cases
      }
      else
      {
         // entered in 60% of cases
      }
   }
}
```

Fig. 4. Simplified code with branches and loop

Basic control flow constructs are conditional branch and loop, Fig. 3. Conditional branch consists of three, and loop consists of four control flow basic blocks. If code structure in Fig. 4 is transformed to CDFG, the appropriate profiling would be as in Table I. The code structure is flattened to basic block level of abstraction where every basic block gets unique identifier. The special purposed code annotations help in generating profiling info for key basic blocks; those representing bodies of conditional branches and loops. It helps the user to identify other basic blocks that result from those constructs. The user fills the correct profiling info for loop condition test and increment as for these the info is not generated automatically.

TABLE I
CODE PROFILING AT TWO LEVELS

Procedure name (#calls)	Procedure part (#iterations)	Basic Block	Total # of iterations
proc1 (#calls = 5)	procedure body (#iterations = 1)	BB0	5×1 = 5
proc2 (#calls = 4)	loop initialization (#iterations = 1)	BB1	4×1 = 4
	loop condition test (#iterations = 10)	BB2	4×10 = 40
	loop body (#iterations = 10)	BB3	4×10 = 40
	loop increment (#iterations = 10)	BB4	4×10 = 40
proc3 (#calls = 3)	loop initialization (#iterations = 1)	BB5	3×1 = 3
	loop condition test (#iterations = 20)	BB6	3×20 = 60
	branch condition test (#iterations = 20)	BB7	3×20 = 60
	branch *if* body (#iterations = 0.4×20 = 8)	BB8	3×8 = 24
	branch *else* body (#iterations = 0.6×20 = 12)	BB9	3×12 = 36
	loop increment (#iterations = 1)	BB10	3×20 = 60

C. Platform Description

The description of FPGA platform is provided as set of libraries for all component types:

- Functional units – adder, subtractor, multiplier, divider, shifter, logic functions
- Controller logic
- Memory elements – register, register file, data memory
- Other components – multiplexers.

The components are described in aspects of their operation, inputs, outputs, control signals, latencies and occupation of FPGA logic (slices, DSPs).

D. Basic Block Analysis and Data Path Design Methodology

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 data paths, are performed for every basic block of the CDFG. The last, final data path design unifies all basic blocks data paths into unique one and interfaces it with the 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. 5 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 Table II. Usage analysis notes operand reads and operation activity per state, as in Table III and Table IV.

If one register per operand and one functional unit per operation would be assumed, there would be much redundancies in the data path. Therefore, minimization of

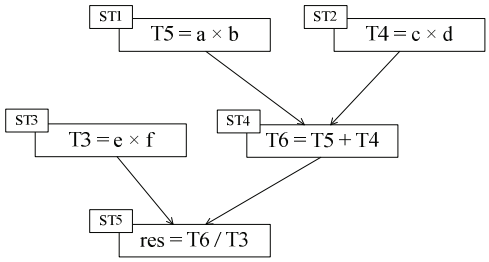

Fig. 5. Data flow graph for expression in (1)

TABLE II
SCHEDULE FOR EXPRESSION (1)

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

TABLE III
OPERAND USAGE ANALYSIS

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

TABLE IV
OPERATION USAGE ANALYSIS

Operation ↓ State →	S1	S2	S3
ADD	0	1	0
MUL	2	1	0
DIV	0	0	1

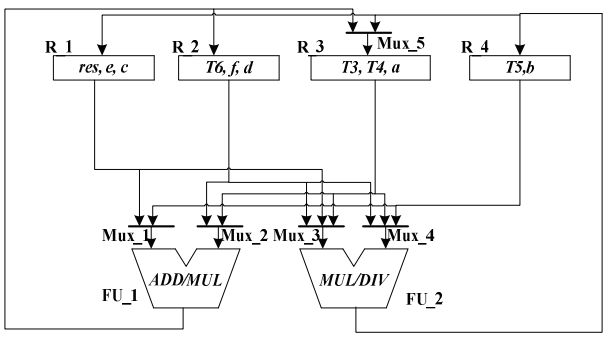

Fig. 6. Final data path for expression in (1).

registers and functional units allocations is applied using compatibility graphs, [15]. Having in mind operand lifetimes and operations usages per finite machine states, operands are appropriately merged to share a register, and combinations of operations are dedicated to functional units. Using this methodology, final data path for expression (1) consists of two functional units and four registers appropriately connected, Fig. 6.

E. Final Data Path Design Algorithm

Algorithm that designs the final data path integrates contributions of all basic blocks provisional data paths. It is also based on analysis of maximum needed operation instances per scheduled cycles for all basic blocks. The contributions of all basic blocks are integrated to the final data path 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 data path is exceeded, those types of functional units are not further instanced. The pseudo code describing the algorithm for data path design is following:

```
DP = Ø
FUmax_op = GET_MAX_INSTANCES(CDFG),
                          op ∈ {ADD, SUB, …, ASSIGN}
base_logic = {CTRL, DMEM_WRAP}
DP += base_logic
BB_sorted = SORT_BB(CDFG, desc)
for bb_current ∈ BB_sorted
    ACCOMP_dmem = ACCOMP_LOGIC(DMEM(bb_current))
     UPDATE_DP_RFS(bb_current)
    FU_sorted = SORT_FU (bb_current, desc)
    for fu_current ∈ FU_sorted
         Op= GET_OPERATION(fu_current)
       if FUinstances_op < FUmax_op
         ACCOMP_fucurrent = ACCOMP_LOGIC(fu_current)
          DP += fu_current + ACCOMP_fucurrent
       else
         UPDATE_DP(bb_current)
RF_UPDATE(DP)
UPDATE_CTRL_CONNS(DP, CTRL)
```

Data path (DP) is built from the scratch. As first, GET_MAX_INSTANCES() notes all operation types and analyzes their usage per basic blocks schedules. As result, maximum numbers of operations per scheduled state is noted ($FUmax_{op}$). Data path is initialized with base logic consisting of control logic (CTRL) and data memory wrapper logic (DMEM_WRAP). Basic blocks are sorted by significance (BB_{sorted}) and traversed in that order. Inside basic block data path, data memory and functional units accompanying logics are checked and integrated into final data path. Accompanying logic of a component is defined as all components connected with it. There are data memory ($ACCOMP_{dmem}$) and functional units ($ACCOMP_{fucurrent}$) accompanying logic consisting of registers connected to their inputs and outputs. When more than one input connection is brought to some input port, multiplexer is automatically instantiated in the final data path.

Functional units are sorted by number of three-address statements assigned to them. Before integration of new functional unit to the final data path, number of functional unit instances per operation type is checked against to previously calculated $FUmax_{op}$ value. If operation count exceeds this value, new functional units of same operation type are not further added to the final data path. In that case, statements assigned to functional unit that is currently processed are assigned to final data path functional unit of appropriate type. The final steps of the algorithm are data path updates denoted as RF_UPDATE() and UPDATE_CTRL_CONNS(). Function RF_UPDATE() assumes removing multiple appearances of the same operand identifiers what is common case when writing code in more than one procedure. Function UPDATE_CTRL_CONNS() interfaces the data path to control unit with connections for bringing constant value to data path and status signals to control unit.

F. Summary

The approach of our processor architecture design compared to traditional high-level synthesis shows several differences, Table V. There are no algorithm optimizations applied, but optimization is undertaken at architectural level during basic blocks data paths designs. Scheduling phase starts immediately after data dependencies analysis and is base for operation and operand usage analyses. Those analyses are base for the joint task of resource allocation and binding. Platform file is used during scheduling phase as it provides information on components latencies and during final data path design as it provides components occupation data.

III. TEST CASES

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

- Discrete cosine transform (DCT) on 8×8 matrices in two versions: original (with three nested loops) and unrolled, [16].
- 32-point DCT used in MP3 decoder.
- SHA-1 encryption algorithm, [17,18].

Table VI shows characteristics of test cases CDFGs: numbers of basic blocks, statements, operands and different operation types.

TABLE V
RELATION OF CUSTOM PROCESSOR DESIGN FLOW TO HIGH-LEVEL SYNTHESIS

High-Level Synthesis Steps	Our Approach		
	Yes/ No	No. in Order	Comment
Lexical Analysis	Yes	1.	Code preprocessing: transformation to CDFG
Algorithm Optimizations	No	-	Optimizations are performed only at architectural level
Control/Data Dependencies Analysis	Yes	2.	Only data dependencies are analyzed
Technology Library Processing	Yes	3.	Platform file that describes mapping to FPGA
Resource Allocation	Yes	5.	Joint process with 'Functional Units & Register Binding'
Operation Scheduling	Yes	4.	Three-address statements within control flow basic block
Functional Units & Register Binding	Yes	5.	Joint process with 'Resource Allocation'
Output: RTL Code	Yes	6.	Synthesizable Verilog code

TABLE VI
TEST CASES CDFG CHARACTERISTICS

Test Cases	# Basic Blocks	#State-ments	#Ope-rands	#Opera-tions
DCT 8×8	15	28	29	6
Unrolled DCT 8×8	5	161	191	8
32-point DCT	1	791	890	6
SHA-1	31	158	150	12

The first two, DCT and Unrolled DCT represent the same code written in different styles and elaborated in [16]. 32-point DCT 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. SHA-1 has the most complex control flow and the most diverse operations: additions, subtractions, logical AND, OR, XOR and NOT, assignment, comparisons, left and right shifts, and memory reads and writes.

The analysis of maximum needed operation instances is shown in Table VII. During final data path design those values are tested as top values of operation instances allowed. As they express the top values of operation instances found across all basic blocks data paths those values can be considered as a kind of parallelizing potential measure.

TABLE VII
TOP OPERATION INSTANCES COUNTS

Test Cases	ADD	SUB	MUL	SHIFT	AND	OR	XOR	NOT	COMP	ASSIGN
DCT 8×8	1		1						1	1
Unrolled DCT 8×8	2		1		1	1			1	2
32-point DCT	3	2	1	1						4
SHA-1	1	1		1	2	2	1	1	1	3

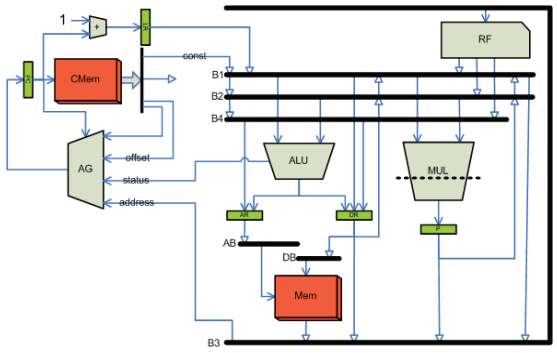

Fig. 7. NISC style MIPS architecture (source: [19])

IV. RESULTS

A. Comparison against other Custom Design Approaches

Custom architectures are built and their implementations are compared for respective test cases with three other implementation approaches:

- NISC style MIPS architecture as one of the referent architecture used inside NISC toolset, Fig. 7, [19].
- Xilinx MicroBlaze processor.
- Xilinx Vivado HLS tool, [20].

For all of them, the target platform was FPGA Virtex-5 XC5VSX50T device (package FF1136 speed grade -1), and performance was noted from tools reports, simulation or logic analyzer output waveforms. The referent Xilinx Design Suite used was version 12.3.

Our flow implements designs with performances range at the same level or better than other processor based designs, Table VIII. With capabilities for parallelizing execution and instantiation of more functional units it is better in cycle count measure than other processors. In achieved work frequencies it is below other processors. HLS tool is far ahead all implementations as it has fully custom control logic.

B. Data Path Designs Characteristics

Table IX presents data path characteristics of all custom built data paths. According to Table VI and Table VII expectations, SHA-1 has the most functional units instanced. There are seven functional unit in its data path, but, as one

TABLE VIII
COMPARISON OF CUSTOM DESIGNS IN CYCLE COUNTS

Test Cases	NMIPS (67–69 MHz)	Microblaze (100 MHz)	Vivado HLS (> 400 MHz)	ArkBuilder (57-61 MHz)
DCT 8×8	10382	51616	3929	10830
Unrolled DCT 8×8	3566	93248	832	3333
32-point DCT	788	28318	199	426
SHA-1	4327	67087	2726	3830

TABLE IX
CUSTOM DATA PATHS CHARACTERISTICS

Test Cases	Data path components			
	Functional units	Register Files	Multiplexers	Connections
DCT 8×8	3	14	13	80
Unrolled DCT 8×8	5	25	24	164
32-point DCT	5	29	32	343
SHA-1	7	30	37	264

functional unit can implement combinations of operations, the actual number of operations is higher (i.e. 12). Instantiation of functional units is controlled by checking the maximum allowed numbers per operation types. Therefore, the numbers of functional units instances are moderate for all test cases. On the other hand, the numbers of register files and registers is higher than expected as their instantiations are not controlled. These increase connections and multiplexers insertions which cause inaccuracy in resource occupation estimation.

V. CONCLUSION

In this paper, we elaborated the global flow of C code specification implementation to FPGA. The idea was in handling complex code examples and production of custom architectures for them. The methodology includes:

- code hierarchical division through procedures and their control flow blocks,
- control flow blocks profiling, producing provisional optimized data paths for all basic blocks,
- unifying basic blocks demands for components into final data path.

The results of FPGA implementation produced from such semi-automated flow were presented in aspects of execution clock cycles and work frequencies and compared against three other implementations.

Design time for final data path design with presented algorithm was within few seconds, except for 32-point DCT. 32-point DCT compatibility graphs analysis was much more costly because of huge number of statements.

Further work will include efforts on better control and optimization of all components; especially register files, and accurate resource occupation estimation. Also, the mechanisms of pipelining, on structural and functional unit level, are not still considered seriously. In all, there is a space for improvement in user impact during system design. More detailed analysis of functional units and registers utilization would allow testing of trade-off between cycle count performance and design minimization.

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.

REFERENCES

[1] A. Gerstlauer, C. Haubelt, A. D. Pimentel, T. P. Stefanov, D. D. Gajski, J. Teich, "Electronic System-Level Synthesis Methodologies," in *IEEE Transactions of Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 10, pp. 1517-1530, October 2009.

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

[3] G. Martin, G. Smith, "High-Level Synthesis: Past, Present, and Future," in *IEEE Design & Test of Computers*, vol. 26, no. 4, pp. 18-25, July-August 2009.

[4] (2013) Gary Smith EDA. [Online]. Available: http:// http://garysmitheda.com/

[5] 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.

[6] 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.

[7] 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.

[8] (2013) Platform Studio and the Embedded Development Kit (EDK). [Online]. Available: http://www.xilinx.com/tools/platform.htm

[9] (2013) MicroBlaze Soft Processor. [Online]. Available: http://www.xilinx.com/tools/microblaze.htm

[10] R. E. Gonzalez, "Xtensa: A Configurable and Extensible Processor," in *IEEE Micro*, vol. 20, no. 2, pp. 60-70, March-April 2000.

[11] 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.

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

[13] (2013) SPARK: A Parallelizing Approach to the High-Level Synthesis of Digital Circuits. [Online]. Available: http://mesl.ucsd.edu/spark

[14] O. Esko, P. Jääskelainen, P. Huerta, C. S. De la Lama, J. Takala, J. I. Martinez, "Customized Exposed Datapath Soft-Core Design Flow with Compiler Support," in *Proceedings of International Conference of Field Programmable Logic and Applications (FPL)*, Milano, Italy, August-September 2010.

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

[16] 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.

[17] (2013) Secure Hash Standards (SHS). [Online]. Available: http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf

[18] (2013) C to Verilog. [Online]. Available: http://c-to-verilog.com/howtos.html

[19] (2013) NISC Technology – Toolset online demo. [Online]. Available: http://www.ics.uci.edu/~nisc/demo/

[20] (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