Efficient C Level Hardware Design for Floating Point Biomedical DSP Applications*

Harry Sidiropoulos[†], Efthymia Kazakou[†], Christoforos Economakos[‡] and George Economakos[†], *Member, IEEE*

***Abstract*— Recent advances in embedded system design has increased their interference in different application domains, where software only solutions have prevailed. This new implementation platform require however quality of results in terms of speed, power and computational complexity, along with strict time-to-market schedules. Performance is sought by utilizing modern Field Programmable Gate Array (FPGA) devices, offering hundreds of GFLOPs with maximum power efficiency. Productivity is enforced with High-Level Synthesis (HLS) or Electronic System Level (ESL) or C-based hardware design methodologies, that offer an efficient abstraction level to boost-up early prototyping. However, just like the migration from schematics to Hardware Description Languages (HDLs) required specific coding styles for efficient hardware design, C-based hardware design also requires efficient coding guidelines to be followed. This paper presents a set of such coding guidelines, and evaluates their efficiency for FPGA based scientific, floating point arithmetic calculations. As found through extensive experimentation, the performance and area optimizations offered by efficient coding can improve the ones offered by HLS only, even more than 90%. So, while not every C program can be turned into hardware with the press of a button, efficient coded C programs can offer a profitable productivity boost.**

I. INTRODUCTION

The electronics design industry has traditionally been driven by two key factors: improve quality (in terms of performance, resource usage, power dissipation, etc.) and reduce time-to-market. The combined satisfaction of both factors can be achieved by modern design techniques like *High-Level Synthesis* (HLS), *Electronic System Level* (ESL) design or, in simpler terms, C-based hardware design. HLS, ESL and C-based hardware design, all more or less involve the automatic translation of untimed algorithmic descriptions into *Register-Transfer Level* (RTL) architectural descriptions, ready for implementation. As a research topic it started more than 30 years ago, and can be divided into three generations [6], with the third, starting in 2000 and lasting up to now, being more mature, starting from system level languages and mainly C/C++, offering a different design paradigm separated from RTL and *Hardware Description Languages* (HDLs) and, based on recent advances in *Field Programmable Gate Array* (FPGA) technology, highly improving quality of results.

Even though a lot of work was been presented in these three decades, the main misunderstanding still present is that C-based design tools can transform software into hardware in a press-and-go way, with minimal user interaction. This approach usually leads into low quality results, for which neither the technology nor the tools are to blame. The same thing happened during the migration from schematics to HDLs in hardware design, where specific, efficiency improving coding styles were introduced. Using the same approach, C-based hardware design also requires efficient coding guidelines to be followed, related to specific architectural optimizations.

This paper presents a set of coding guidelines for C-based hardware design and evaluates their efficiency for FPGA based scientific, floating point arithmetic calculations. With recent advances in FPGA technology, IEEE-754 based floating point applications are a growing trend. While the academic world is dealing with different architectures and optimizations [10], [5], demanding applications involving biomedical DSP algorithms [1], [3], [8] take advantage of them to offer quality of results. Comparisons between fixed and floating point implementations are shown in [7], [9].

The major contributions of this paper are the following. First, it presents a thorough set of C coding guidelines for efficient hardware design. These guidelines are applied in an iterative manner, using a custom script based environment, along with the architectural optimizations (loop pipelining and unrolling, memory organization, interface synthesis) offered by Calypto's Catapult ESL tool. Through extensive experimentation with floating point operator implementations (add/subtract, multiply, divide), it is shown that performance and area improvements offered by efficient coding can outperform the ones offered by architectural style selections only, even more than 90%. Second, it presents an efficient and reusable C level hardware implementation of single precision floating point operators, based on the SoftFloat open source project [4]. Finally, it supports the claim that while not every C program can be turned into hardware with the press of a button, efficient coded C programs can offer a profitable productivity boost.

II. PROPOSED METHODOLOGY

The design methodology proposed in this paper is a two step approach, based on the Catapult C-based synthesis environment. The first step applies algorithmic optimizations

*This research has been co-financed by the European Union (European Social Fund - ESF) and Greek national funds through the operational program "Education and Lifelong Learning" of the National Strategic Reference Framework (NSRF) - Research Funding Program: ARCHIMEDES III: Investing in knowledge society through the European Social Fund.

[†]H. Sidiropoulos, E. Kazakou and G. Economakos are with the School of Electrical and Computer Engineering, National Technical University of Athens, Heroon Polytechniou 9, GR-15780 Zografou, Athens, Greece geconom@microlab.ntua.gr

[‡]C. Economakos is with the Department of Automation, Technological Educational Institution of Sterea Ellada, GR34400 Psahna, Evia, Greece economakos@teihal.gr

978-1-4799-3163-7/13/$31.00 ©2013 IEEE

to the initially supplied C/C++ source design file. These optimizations are a set of coding guidelines, inserted or changed in the initial source with appropriate script files. Conditional compilation switches are used, so the whole process is rather interactive and not automated, leaving to the user the choice of either following or not following the proposed guidelines. All algorithmic optimizations are evaluated by performing a synthesis step in Catapult. The second step applies architectural optimizations, traditionally found in C-based design tools like loop unrolling, loop pipelining, memory organization and interface synthesis. Again, all optimizations are evaluated through synthesis. The overall methodology supports comparisons between the two types of optimizations (algorithmic and architectural), which is one of the key contributions of the paper. Details about these optimizations are given in the following subsections.

A. Algorithmic Optimizations

A set of algorithmic optimizations have been implemented in the proposed methodology, as conditionally inserted or changed code fragments (or even just informative comment notifications to the user). These fragments originate from the following rules:

1. Special bit accurate types provided by Catapult, are used, offering simulation performance and synthesis quality.

2. Input and output signals are inferred by the top level function parameters and they way they are used.

3. Pass-by-value function parameters require internal registers while pass-by-reference no.

4. Function output parameters are registered by default.

5. Loop boundaries should be fixed, for decidable execution time.

6. Conditionals should explicitly be denoted mutually exclusive, with complete **if** . . . **else** clauses, for decidable execution time. For example, the following two functions, when only two adders are available, give different implementations (schedules) when mutually exclusive paths are not explicitly defined (max_x and figure 1, two adders per control step), and when they are (max_n and figure 2, four adders in control step C1, while eventually only two will be used).

Listing 1. Non-explicitly/explicitly defined mutually exclusive code.

```
int max_x(int a[4], int b[4], int sela, int selb) {
  if (sela) return a[0]+a[1]+a[2]+a[3];
  if (selb) return b[0]+b[1]+b[2]+b[3]; }

int max_n(int a[4], int b[4], int sela, int selb) {
  int retval=0;
  if (sela) retval=a[0]+a[1]+a[2]+a[3];
  else if (selb) retval=b[0]+b[1]+b[2]+b[3];
  return retval; }
```

7. Parentheses can be used to force height reduction and common subexpression elimination.

8. The use of constant multipliers and shifts is much preferred than general purpose multipliers.

9. Arrays mapped to memories can be optimized for speed, by widening the memory I/O bitwidth (64 bit ports for 32 bit operands), allowing more than one parallel memory accesses.

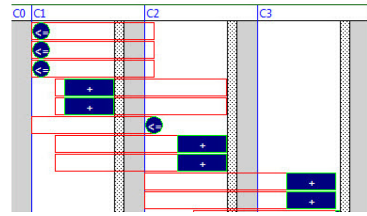

Fig. 1. Non-explicitly defined mutually exclusive schedule.

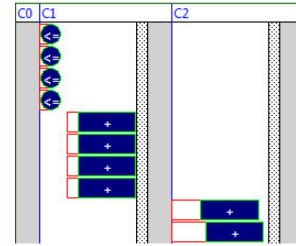

Fig. 2. Explicitly defined mutually exclusive schedule.

10. The size of memory mapped arrays can be extended to a power of 2, for simpler controller generation and performance improvements.

11. Global and static variables always generate registers but statics are preferred for code compactness.

B. Architectural Optimizations

Architectural optimization are those supported by the Catapult ESL tool and may be applied through GUI manipulations or script files (used to automate the HLS process). In brief, they are the following:

1. Loop pipelining, which is controlled by the initiation interval directive. This directive denotes that each loop iteration will wait a number of control steps before it starts. For a loop with no data dependencies, all iterations can be performed in parallel. For a loop with strong data dependencies, each iteration may be forced to start only after the previous has completely finished. Usually, feasible solutions lie in the middle. When one loop iteration partially overlaps another, Catapult generates a pipelined algorithm implementation.

2. Loop unrolling, which is the duplication of the loop body a number of times, denoted by the Catapult unrolling directive. Since each loop iteration takes at least 1 control step to finish, by duplicating the iteration we investigate the opportunity to put more operations within this limit and lower the repetitions.

3. Loop merging, which can combine loops with identical bounds. Normally, Catapult schedules consecutive loops found in source code one after the other, with no overlapping. If the loops however have identical bounds and data dependencies permit it, both loops can be executed in parallel, by merging their corresponding iterations.

4. Memory map threshold. Catapult can map data objects either in register files or in memories. Small data objects can be mapped in register files, with very fast access times but more complicated control logic while large data objects can be mapped in memories with slower access times but less complicated control logic.

5. Internal memory organization. Internal memories in Catapult have two properties that can affect performance, the

TABLE I
200 MHz OPERATORS.

Sol.	Lat.	Thr/put	LUTs	DFFs	DSPs
Adder/Subtractor					
no	95	100	2478	657	15
arch	45	30	3128	1233	26
	52.63%	70%	-26.23%	-87.67%	-73.33%
alg	15	20	1500	288	7
	84.21%	80%	39.47%	56.16%	53.33%
arch	15	5	1306	375	8
+alg	84.21%	95%	47.30%	42.92%	46.67%
	66.67%	83.33%	58.25%	69.59%	69.23%
Multiplier					
no	65	70	970	362	9
arch	30	20	1135	630	17
	53.85%	71.43%	-17.01%	-74.03%	-88.89%
alg	25	30	498	186	5
	61.54%	57.14%	48.66%	48.62%	44.44%
arch	20	5	837	587	12
+alg	69.23%	92.86%	13.71%	-62.15%	-33.33%
	33.33%	75%	26.26%	6.83%	29.41%
Divider					
no	205	210	3170	569	17
arch	335	325	5100	1328	20
	-63.41%	-54.76%	-60.88%	-133.39%	-17.65%
alg	65	70	994	275	3
	68.29%	66.67%	68.64%	51.67%	82.35%
arch	50	45	1179	435	7
+alg	75.61%	78.57%	62.81%	23.55%	58.82%
	85.07%	86.15%	76.88%	67.24%	65%

TABLE II
400 MHz OPERATORS.

Sol.	Lat.	Thr/put	LUTs	DFFs	DSPs
Adder/Subtractor					
no	102.5	105	2840	880	22
arch	37.5	30	4313	1723	30
	63.41%	71.43%	-51.87%	-95.80%	-36.36%
alg	15	17.5	1102	317	6
	85.37%	83.33%	61.20%	63.98%	72.73%
arch	15	2.5	1382	990	8
+alg	85.37%	97.62%	51.34%	-12.50%	63.64%
	60%	91.67%	67.96%	42.54%	73.33%
Multiplier					
no	45	47.5	1362	527	10
arch	35	30	2037	854	3
	22.22%	36.84%	-49.56%	-62.05%	70%
alg	20	22.5	509	255	6
	55.56%	52.63%	62.63%	51.61%	40%
arch	20	2.5	842	982	7
+alg	55.56%	94.74%	38.18%	-86.34%	30%
	42.86%	91.67%	58.66%	-14.99%	-133.33%
Divider					
no	177.5	180	3338	667	22
arch	325	312.5	5589	1452	22
	-83.10%	-73.61%	-67.44%	-117.69%	0%
alg	52.5	55	984	208	6
	70.42%	69.44%	70.52%	68.82%	72.73%
arch	40	40	1265	436	6
+alg	77.46%	77.78%	62.10%	34.63%	72.73%
	87.69%	87.20%	77.37%	69.97%	72.73%

number of available I/O ports and a number of independent blocks that they can be split. Both properties, increase the number of parallel memory accesses in a single control step.

Architectural optimizations can be iteratively applied and tested before final decisions are made, with the use of appropriate script files.

III. EXPERIMENTAL RESULTS

A. Operator Implementation

Experimental result with the previously presented methodology are given in tables I and II. They report implementation details for three single precision (32 bit) floating point operators, an adder/subtractor, a multiplier and a divider. For all implementations, the largest Virtex-6 Xilinx FPGA device has been used, the 6VLX760. Moreover, two operation frequencies have been selected, 200 MHz (table I) and 400 MHz (table II). For all implementations, Catapult was used to obtain preliminary results through algorithmic and architectural optimizations and then Precision Synthesis and Xilinx ISE to get low-level, accurate results.

All implementations start from the corresponding C level open source software implementation found in the SoftFloat library [4] and each table presents hardware implementation details for four solutions in three sections, one for each operator. The first (solution no) is exactly the SoftFloat code, without any optimization. The second (solution arch) is the result of architectural optimizations applied only. The third (solution alg) is the result of algorithmic optimizations applied only. Finally, the fourth (solution arch+alg) is the result of combined architectural and algorithmic optimizations applied. For each solution, two performance metrics are reported, latency and throughput in ns, as well as three FPGA area metrics, the number of *Look-Up Table* generators (LUTs), *D-type Flip Flops* (DFFs) and special purpose DSP blocks. In each operator section, rows 1, 2, 4 and 6 show absolute values, while rows 3, 5 and 7 show improvement (or overhead) percentages of the corresponding solution with respect to row 1 (solution no). Finally, row 8 (which is the most interesting) shows improvement (or overhead) percentages of the final solution (solution arch+alg), with respect to row 2 (solution arch). In essence, this is the comparison between HLS with and without specific C level coding styles (algorithmic optimizations).

As it can be seen, both tables offer performance improvements in all solutions compared to solution no, except the divider operator where an unbounded **while** loop limits the effectiveness of HLS transformations. This comes along with area overheads, when architectural optimizations are used alone (solution arch). While this is an expected result (optimizations offer faster hardware requiring more resources), it does not happen when algorithmic optimizations are used alone (solution alg). On the contrary, following appropriate coding guidelines has been proved to offer both performance and area improvements in all cases. Furthermore, when architectural and algorithmic optimizations are combined (solution arch+alg), top performance improvement is achieved, reaching more than 90% in some cases, with area improvements or overheads in different operators. Overheads are

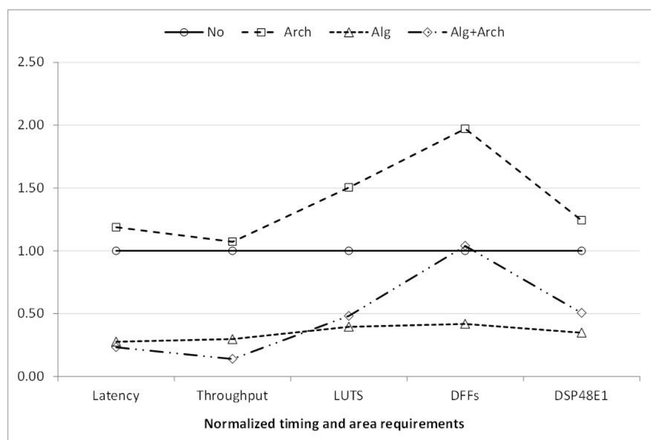

Fig. 3. Normalized result graph.

however less than the arch solution in all cases, except from the DFFs in the 400 MHz multiplier. The most interesting comparison is given in the final row of each section, where solutions arch (HLS) and arch+alg (HLS+coding guidelines) are compared. In all cases except the 400 MHz multiplier both performance and area improvements are reported, reaching again as high as 90% in some cases.

Finally, figure 3, gives a graphical view of all the results found in tables I and II. It shows normalized average values for each solution (with no having the value 1) and all performance and area metrics. As it can be seen, when algorithmic optimizations are involved (two lower lines), average improvements are always found. In fact, the alg solution looks like a baseline drift in the whole plot. The arch solution is by far the most demanding by average in terms of area and close to the no solution in terms of performance (due to the inefficient handling of the unbound **while** loop in the divider operator). Overall, this figure gives a clear image of the advantages of correct C level coding for C-based hardware design.

B. Application Support

Computed Tomography (CT) scanning uses special x-ray equipment to generate multiple views of the inside of the body. These multiple x-ray views are reconstructed into cross-sectional images of the body, using very computationally intensive algorithms. At the core of any CT scan image reconstruction is an algorithm called *Filtered Backprojection* (FBP). Backprojection is nothing more than adding each filtered x-ray image data sets contribution into each pixel of the final image reconstruction. Each x-ray view data set consists of hundreds of floating point numbers, and there are hundreds of these data sets. In order to test the proposed hardware floating point operators of the previous subsection in a demanding application, the single core backprojection algorithm found in [2] for a 4096x4096 image size, has been implemented for the Xilinx Virtex-6 6VLX760 FPGA running at 200 MHz and using the Catapult ESL tool.

The results found, even preliminary (no low-level implementation has been performed and non-optimized trigonometric function implementations have been selected), show performance improvements similar to the operator implementations. Specifically, the no solution has a 1530 ns latency, 1625 ns throughput and 21856 sec total execution time (for the whole 4096x4096 image), the arch solution 410 ns, 205ns and 2757 sec, the alg solution 105 ns, 70ns and 941 sec and the arch+alg 85 ns, 45 ns and 605 sec (area metrics are not reported because the huge memory requirements did not allow the design to fit in the FPGA device, prohibiting also low-level implementation). So, with very fast and efficient integration opportunities (operator and application code is written in C), the proposed coding methodology can give very promising results and an overall 36x performance boost (comparing no to the arch+alg solution).

IV. CONCLUSIONS

This paper has given experimental results to support the claim that C based hardware design should not be considered as a press-and-go procedure that transforms software into hardware. While modern ESL tools offer the power to transform most C/C++ programs into hardware, it is efficiently coded programs that can offer the profitable productivity boost that this technology is promising. Applying specific coding guidelines to C level software implementations together with HLS transformations, quality of results comparable with hand written RTL have been presented, thus proving the maturity of the underlining technology.

REFERENCES

[1] J. Chandran, A. Stojcevski, A. Zayegh, and T. Nguyen. Implementation of a colorimetric algorithm for portable blood gas analysis. In *22nd International Conference on Microelectronics*, pages 411–414. IEEE, 2010.

[2] W. Chapman, S. Ranka, S. Sahni, M. Schmalz, L. Moore, U. Majumdar, and Elton B. Multi- and many-core technologies: Architectures, programming, algorithms, and applications. chapter Backprojection on Multicore and GPU Architectures. Chapman-Hall/CRC Press, 2012.

[3] W. C. Fangl, C. C. Choul, T. H. Hungl, K. C. Linl, A. H. Li, Y. C. Chang, B. K. Hwang, and Y. W. Shau. An efficient and accurate empirical mode decomposition of the technical design and methods for biological sound. In *Biomedical Circuits and Systems Conference*, pages 320–323. IEEE, 2012.

[4] J. Hauser. Softfloat, a free, high-quality software implementation of the IEC/IEEE standard for binary floating-point arithmetic. `http://www.jhauser.us/arithmetic/SoftFloat.html`.

[5] K. S. Hemmert and K. D. Underwood. Fast, efficient floating-point adders and multipliers for FPGAs. *ACM Transactions on Reconfigurable Technology and Systems*, 3(3):11:1–11:30, 2010.

[6] G. Martin and G. Smith. High-level synthesis: Past, present, and future. *IEEE Design and Test of Computers*, 26(4):18–25, 2009.

[7] B. W. Robinson, D. Hernandez-Garduno, and M. Saquib. Fixed and floating point analysis of linear predictors for physiological hand tremor in microsurgery. In *International Conference on Acoustics Speech and Signal Processing*, pages 578–581. IEEE, 2010.

[8] F. A. Samman and P. Surapong. SPECTRON: Streaming processor specific for adaptronic and biomeditronic applications. In *International Conference on Biomedical Engineering*. IEEE, 2012.

[9] D. Xiao, L. Wenjun, D. Hui, and W. Guangzhi. Hardware acceleration for motion tracking system used in image-guided surgery. In *3rd International Conference on Biomedical Engineering and Informatics*, pages 1498–1502. IEEE, 2010.

[10] L. Zhuo and V. K. Prasanna. Scalable and modular algorithms for floating-point matrix multiplication on reconfigurable computing systems. *IEEE Transactions on Parallel and Distributed Systems*, 18(4):433–448, 2007.