# Efficient Implementation of Arithmetic Calculation Circuits

Lazaros Batsilas
Department of Informatics and Telecommunications Engineering,
University of Western Macedonia,
Kozani 50100, Greece
st0181@icte.uowm.gr

Minas Dasygenis
Department of Informatics and Telecommunications Engineering,
University of Western Macedonia,
Kozani 50100, Greece
mdasyg@ieee.org

***Abstract*—In recent years considerable attention has been devoted in the development of reconfigurable architectures for different classes of problems. One specific example is the fama (flexible-adder-multiplier-adder) cell which was proposed to provide efficient arithmetic computation in DSP applications. Although this architecture has been shown to outperform alternative implementations in specific examples, trying to exploit it manually in novel problems requires considerable effort. This work describes an algorithm and a tool which use fama architecture cells as building blocks and combine them to perform complex computations. The tool takes a mathematical expression string as input and generates a VHDL description of the circuit evaluating this expression.**

## I. Introduction

Reconfigurable architectures have proven successful in modern applications both in terms of performance as well as cost or power consumption [1]. Recently a reconfigurable architecture was proposed, designed to enable the execution of combinations of arithmetic operations in an efficient manner in terms of time and space [2]. This effort is particularly relevant in the domain of digital signal processing in which optimization of mathematical computations is a decisive factor in determining system performance, since heavy computation modules with light-weight control structures are involved.

The proposed architecture presents a compromise between two different paradigms [3]. On the one hand, special purpose designs result in efficient implementations, but require considerable amounts of skilled work and time. On the other hand, general purpose hardware components are flexible providing solutions that are fast and straightforward to implement but are not optimized to meet the needs of a specific application. The approach described here utilizes reconfigurable circuits that have been carefully optimized to efficiently calculate a set of arithmetic expressions, while being flexible enough to be combined together into circuits performing arbitrary, complex operations.

Increased performance of this methodology in the domain of DSP has been demonstrated in real system examples [4]. However, widespread adoption of this scheme is contingent on the development of appropriate tools, which would automate design, relieving developers from devoting unnecessary time and effort. To this end, this work describes a tool[1] which was developed to generate optimized VHDL code, simply by specifying a mathematical expression using a high-level description. Such an expression may exhibit arbitrary complexity with respect to the number and grouping of operations.

This paper is organized as follows: In the next section the fama cell which is the building block out of which the total circuit is built is described. In section 3 the processing steps are outlined and explained. Section 4 presents the results of measurements for each stage of processing, while in the last section some possible directions for future work are presented.

## II. Description Of fama Cell

Central to this work is the flexible adder-multiplication-adder cell introduced in [2]. This is a combinatorial circuit which is able to perform efficiently basic arithmetic operations frequently encountered in DSP applications. Cells are configurable. By setting configuration signals appropriately, different datapaths within the cell may be selected, resulting in different arithmetic manipulations of the cell inputs.

Fama cells perform operations efficiently by exploiting the Carry-Save representation [5-6]. Signals in this representation are comprised of two separate parts, the “carry” and the “save” part. Both of these are encoded in standard two-complement binary format, while the overall value of such a signal is the sum of these parts. The fact that there are two different parts for each signal is exploited in the implementation of the adders used. Specifically, summation proceeds without carry propagation delay, since carries can be saved as a separate signal to be added to the result at a later stage. Fama cell operations support calculations in Carry-Save format without the need for conversion to standard form and produce a result that is also in the Carry-Save format. Thus, individual cells can be connected into a network of cells performing more complicated calculations. In many cases, a standard adder is necessary only as the last step of the computation in order to produce the final result in standard binary form.

1 The tool is available available upon request for any research team interested in working with FAMA cells

There are four inputs and one output in each cell. Inputs X*, Y* and K* are in Carry-Save form while, input A is in two-complement binary form. This signal is one of the two inputs fed to the multiplier. The output Z* is also in carry-save format.

As the name suggests individual fama cells comprise three sequential operation circuits: An adder, feeding a multiplier which is in turn fed into a second adder. Adders are implemented using the 4:2 C-S design, while the multiplier uses a modified Booth scheme [7]. Both adders can be configured to perform subtractions by setting the appropriate configuration signal which selects the inverted input to be fed to the adder. In addition, there are two more configuration signals which serve as select signals into two 2-to-1 multiplexers which determine datapath within the cell. Thus, based on the configuration signals, different operations can be performed. These include the operations listed below. More simple operations such as addition of two C-S inputs can also be performed by using appropriate fixed values for some of the signals (e.g. setting them to 0).

$A\,(X^* \pm Y^*) \pm K^*$

$(X^* \pm Y^*) \pm A\,K^*$

$(X^* \pm Y^*) \pm K^*$

$A\,(X^* \pm Y^*)$

$A\,K^*$

## III. Description Of Our Algorithm

This paper describes a tool which which takes as input a string containing a mathematical expression and generates VHDL code that executes this expression. The input must conform to the usual conventions adopted by common programming languages like C, C++ or Java. It may contain additions, subtractions, multiplications and combinations of these. Multiplications take precedence over additions and subtractions, while it is possible to define grouping of operations by parentheses, which may be used and nested at any level. It is also possible to define string variable names as inputs. Again, standard C conventions are adopted.

The overall process can be divided in two stages. First the input string is processed to generate a graph in memory whose nodes are either input nodes or fama cells configured and connected appropriately. Second a netlist of components is compiled based on this graph as a JSON file which serves as input to the VHDL generation module synthesizing the final code.

The first stage follows the steps depicted in figure 1. Parsing is performed in a multi-pass fashion with each step performing a different function. The first step is the conversion of the input string into a standard data flow graph which is comprised by two different types of nodes, input nodes and operation nodes. Operation nodes types include addition, subtraction and multiplication nodes. Each of these accepts two inputs and produces one output, the result of the corresponding operation.

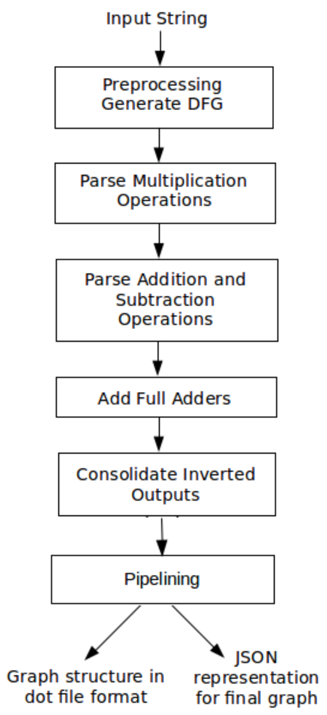
Input String
Preprocessing Generate DFG
Parse Multiplication Operations
Parse Addition and Subtraction Operations
Add Full Adders
Consolidate Inverted Outputs
Pipelining
Graph structure in dot file format
JSON representation for final graph

Fig. 1. Basic stages in generating final graph.

Next, this initial graph is processed in a bottom-up manner, identifying multiplication nodes and replacing them by fama cells. During these substitutions, multiplication nodes are first grouped with neighboring nodes resulting in a network of nodes that can be directly mapped to one of the fama cell operations listed above. This is done with the goal of maximizing the number of nodes replaced. The new fama nodes created are then wired into the graph, while old nodes are discarded.

Since there is only one multiplier in every cell, the number of cells used cannot be less than the total number of multiplications in the given input. If the input string does not contain long chains of additions or subtractions it is possible for all addition and subtraction nodes to be grouped together with multiplication nodes and to be removed from the graph. In this case no processing is necessary. However, if this is not the case and addition/subtraction nodes still exist, a second bottom-up pass is performed in order to incorporate these nodes into cells.

The restriction imposed by the fact that one of the inputs to the multiplier is not in C-S format needs to be addressed. Since cell outputs are in C-S format, an output cannot be directly connected to the A input of a second cell. As a result, when it is necessary to multiply two different cell outputs, one of them needs to be converted to standard form by summing the Carry and the Save parts. An alternative is to use the distributive property of multiplication and use different multipliers for the C and S part of the overall signal. Since addition introduces considerable delay, this issue is basically a trade-off between the number of cells and the latency of the circuit.

A second restriction emerges because of the asymmetry of the fama cell with respect to the multiplier when it is fed to the final addition/subtraction unit. As inversion of the multiplier output contributes to the maximum combinatorial delay, no inverter is included in the design. This, however, means that this signal may serve as the minuend and not the subtrahend of the final subtraction. Thus, in certain cases it is only possible to compute the inverse of the desired expression in a single step. In many times this negative sign can be absorbed in subsequent cells by changing additions to subtractions and/or reversing the order of inputs in downstream cells. In the remaining cases, inverters are used.

Upon completion of the analysis of the input string and generation of a network of components, this network is encoded into a text file in JSON format which contains a netlist of components used. This is then used to generate the VHDL description of the circuit. In this description the bitwidth of signals is parametrized. A graphic representation of inputs, components and the connections between them is also generated which provides a more accessible, visual representation of the graph.

The VHDL code generated contains a pipelined implementation of the design. The basic time scale in the circuit is determined by the combinatorial delay of a fama cell. When other components are used (for example adders that convert a signal from Carry-Save to binary format or inverters) delay flip-flops are added based on the delay of these components and the difference with the delay of a fama cell.

## IV. Algorithm Complexity and Performance

To measure performance of the algorithm, time measurements were taken for inputs that contain varying number of terms. Input strings of different sizes were generated randomly. Additions and multiplications were generated with equal probabilities. Additionally, parentheses were included. Opening parenthesis were added with a probability of 0.2 in all appropriate positions, while positions of the corresponding closing parentheses were also selected at random.

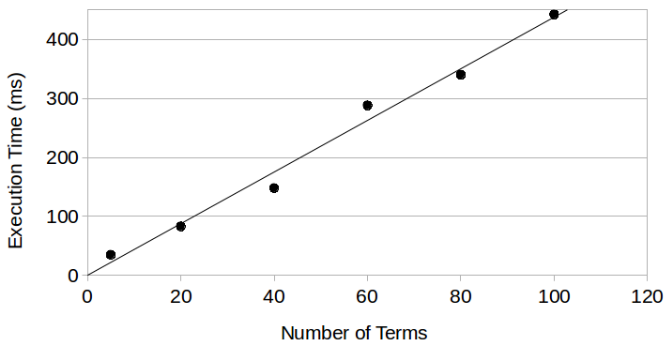

Fig. 2. Execution time for different sizes of input

Figure 2 shows the results of time measurements with respect to input string size. It is clear that the dependence of execution time on input size is linear. This is not surprising since both DFG generation and graph processing steps process one term or one node at a time, while it is also true that the number of cells in the final design is expected to be linear with input size.

Figure 3 presents a more detailed view of the different stages of the algorithm. Specifically, it shows how the overall processing time is allocated to the first six basic steps of the algorithm. As can be seen from the figure, processing of the input string and generation of the DFG, requires substantial computational power (about 45% of the total computation time). The final stage of creating a pipelined design through the addition of DFFs also requires considerable time (about 15%).

With respect to the main computation, which involves the consecutive steps of consolidating first all multiplication nodes and then consolidating the remaining addition/subtraction nodes, the fraction of time required is about 40%. It is interesting to note that most of this time is devoted for the multiplication-parsing step. This is a reflection of the statement made earlier that in many cases additions/subtractions can be merged with neighboring multiplication nodes, leaving few or no addition/subtraction nodes to be parsed in a subsequent pass. This is expected to hold in all cases where no long chains of consecutive addition/subtraction nodes exist.

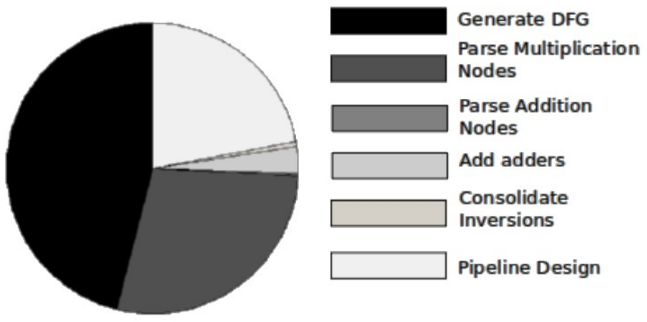

Fig. 3. Individual time requirements for different stages of the algorithm

## V. Future work

One of the important features of the components described is that they are reconfigurable. This not only permits their combination in different ways to produce complex networks, it also opens up the possibility of reuse. Thus, when area or other constraints limit the maximum number of cells that can be used, complex calculations can still be performed by creating the necessary connections between cells and providing time-dependent data and control signals. Two types of control signals are necessary. The first of these would determine the dataflow within the overall circuit, selecting connections between outputs and inputs. The second class would operate on a finer grain level configuring individual cells. We are working on this problem and studying ways to deal with dependencies, ensure intermediate result persistence when needed and optimize hardware design.

At the same time, we would like to extrapolate our methodology to a higher level of abstraction. Specifically we would like to design a tool for which the gamut of available cell operations is parametrized. In this setting, the user would determine what kind of operations a cell is capable of completing, as well as the precedence of each operation. For each operation the original DFG created would then be scanned for parts that can be mapped directly to the given operation. This would be a valuable tool facilitating widespread application both of existing architectures as well as architectures which are likely to be proposed in the future.

## References

[1] S. Stojanovic, D. Bojic, M. Bojovic, M. Valero and V. Milutinovic "An overview of selected hybrid and reconfigurable architectures" in IEEE International Conference on Industrial Technology (ICIT), pp. 444-449, 2012.

[2] M. Galanis, G. Theodoridis, S. Trgoudas and C. Goutis "A high-performance datapath for synthesising DSP kernels. IEEE Transactions on computer-aided design of integrated circuits and systems, 25(6) pp. 1154-1163, 2006

[3] M. Wolf, In High Performance Embedded Computing, 3rd Ed., Elsevier 2014.

[4] S. Xydis, I. Triantafyllou, G. Economakos, K. Pekmestzi, "Flexible datapath synthesis through arithmetically optimized operation chaining" in NASA/ESA Conference on Adaptive Hardware and Systems pp.407-414 .

[5] N. Weste and K. Eshraghian, "Principles of CMOS VLSI Design – A Systems Perspective". Reading, MA: Addison-Wesley, 1985.

[6] A. Verma, P. Ienne. "Improved use of the Carry-Save representation for the synthesis of complex arithmetic circuits" in Proceedings of IEEE/ACM ICCAD, pp. 791-798, 2004.

[7] P. E. Madrid, B. Millar and E. E. Swartzlander, Jr, "Modified Booth's algorithm for high radix fixed point multiplication", IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Vol. 1, No. 2 June 1993.