Analysis and Design of a Specialized Pipeline for Numerical Algorithms Implementation

Viorel CARBUNE
Technical University of Moldova
sirius.c-032@mail.ru

Sergiu ZAPOROJAN
Technical University of Moldova
zaporojan_s@yahoo.com

Abstract. **The numerical algorithms often bring a lot of elementary calculations which involve the fixed-point multiply, add and subtract operations. Many of these numerical and signal processing algorithms require repetitive use of multiply and accumulate operation. The contribution of the paper is to present the analysis and design of an FPGA-based specialized pipeline, in order to develop flexible (co)processors for applications in the field of numerical analysis and digital signal processing. Essentially, the paper focuses on some details of the analysis and design of the proposed pipeline structure. In particular, provided analysis considers the bus traffic utilization in the system. The obtained results are plotted for some specific parameters. The simulation results are also drawn. For this reason, the Altera Quartus software was used.**

Index Terms — **DSP, FPGA, multiply and accumulate operation, numerical algorithm, pipelining.**

I. INTRODUCTION

In scientific computation and signal processing (DSP), high-performance is strictly important. Essentially, these computations can be classified into compute-bound and input/output-bound computations. In a compute-bound calculation the number of arithmetic operations is much larger that the number of input and output elements (e.g. matrix multiplication). The computational tasks involved by algorithms in numerical analysis and DSP are typical of the compute-bound class. In addition, many algorithms in numerical computing and digital signal processing have a high regularity [1, 2, 6]. Hence, numerical methods with these characteristics are quite suitable for a specialized treatment.

To address the performance barrier of scientific computation and DSP, a standard approach in the past has been to increase the operating frequency of the processor. Other well known approaches to improve the performance include the use of additional processors, the use of specialized programmable processors or through the use of FPGA-based processor architectures. Adding additional devices to a system can be costly, especially under the requirements for system reliability.

On the other hand, modern FPGA circuits, with their ability to integrate multiple (co)processors in a single device, can provide advanced solutions to accelerate the performance. Another key advantage of modern FPGAs is the ability to adapt and quickly respond to changing application requirements. As a direct result of the above capabilities, FPGAs can be used to develop highly performance architectures for numerical analysis and DSP applications.

A new category of very high-performance programmable logic devices has been developed to address the un-met needs of system designers. The MathStar Field Programmable Object Array (FPOA) is an example of this category, offering many very useful capabilities [7]. Because of its high-performance, the FPOA is useful in a wide range of applications, including those in the areas of machine vision, medical imaging and image processing. These applications are built around extremely fast specialized building blocks.

The goal of this paper is to present the analysis and design of an FPGA-based specialized pipeline, in order to develop flexible (co)processors for numerical applications. First of all, background section focuses on the features and basics in the field. Then, basic sections are presented. At first, here we discuss some details of the analysis and design of the pipeline. Mainly, our analysis considers the bus traffic utilization in the system. Finally, we present simulation results and conclude the work in the last section.

II. BACKGROUND

The numerical algorithms often bring a lot of elementary calculations. Most of these elementary calculations involve the fixed-point multiply, add and subtract operations. Essentially, the differences between conventional and special-purpose processors involve optimization for specific arithmetic operations and data handling. Such processors are optimized to efficiently execute optimized operations which allow the efficient implementation of numerical processing algorithms. The input signals can be audio, image-based or simply numerical.

Many of these specialized numerical and DSP algorithms require repetitive use of the following operation group:

$$A = B \times C + D \qquad (1)$$

This operation group is clearly a multiply and an addition also known as a multiply and accumulate (MAC). This operation is so common that DSP processors have been optimized to implement one or more MAC operations during each processor instruction cycle. Data handling has also been given significant design attention. Extra buses have been added to processors to allow them to more efficiently handle internal and external data transfers. Pipelines and additional data paths and registers have also been added to speed up arithmetic operations and data transfers [3, 4].

Fore example, the fast Fourier transform (FFT) is one

DSP building block that frequently requires high speed. The FFT can be factored in a variety of different ways. Each way results in a different algorithm. The most common algorithm is the Cooley-Tukey one, which recursively factors each N point transform into a pair of N/2 point transforms combined by a "butterfly" operation until the reduced transforms are each a single sample long. Each butterfly operation consists of a complex multiply by a "twiddle factor" (that is, a phase rotation of the input) followed by a two-point FFT, which is just a complex sum and difference [1, 6]. The regularity of the algorithm and data sequencing is ideal for a high-performance implementation.

Basically, special-purpose processors fall into two main categories based on the way they represent numerical values and implement numerical operations internally. These two main formats are well known: fixed point and floating point. The differences between fixed and floating point processors are significant enough that they require very different internal implementation, instruction sets and approaches for algorithm implementation. Fixed point processors represent and manipulate numbers as integers. Floating point processors primarily represent numbers in floating point format, although they can also support integer representation and calculations [5, 8].

Developing an understanding of which applications are appropriate for floating point processors is very important. The inherently large dynamic range available in floating point designs means that dynamic range limitations can be practically ignored in a practical design. Floating point processors can implement both floating point and integer operations, making them more flexible. In the same time, floating point processors tend to be more expensive because they implement more complexity and have wider buses (32 or 64 bits). On the other hand, floating point processors tend to be more high level language friendly. Floating point capability is appropriate in systems where gain coefficients are changing with time, or the coefficients have large dynamic ranges. Thus, relative ease of development and schedule advantage are being traded off against higher cost and hardware complexity when considering floating point design implementations [3].

Conventional floating-point implementation treats each add or multiply operation as a stand-alone floating-point operation requiring normalized inputs and outputs. When these basic operations are assembled into more complicated operators, the intermediate normalize and de-normalize operations are often unnecessary and represent a considerable amount of wasted hardware. With a hybrid approach [1], it is possible to take larger pieces of the algorithm and treat them as fixed-point blocks that operate on the mantissas of the input data, renormalizing after several algorithm steps rather than after each elemental operation. The input to the fixed-point operator has to be de-normalized so that it shares a common exponent, and the fixed-point operator must have enough extra bits to allow for any combination of inputs without overflow.

Engineers targeting numerical computing to FPGA circuits have traditionally using fixed-point arithmetic, mainly because of the high cost associated with implementing floating-point arithmetic. As mentioned above, that cost comes in the form of increased circuit complexity and often degraded maximum clock frequency. Certain applications demand the dynamic range offered by floating-point hardware but require speeds and circuit sizes usually associated with fixed-point hardware.

To speed up the numerical arithmetic computation, different pipelining techniques are being used. Most of modern's arithmetic pipelines are designed to perform fixed functions. These arithmetic units perform fixed-point and floating-point operations separately [4].

The remainder of this paper focuses on some details of the analysis and design of the FPGA-based specialized pipeline. Next section presents a flexible pipeline structure capable to meet numerical computing requirements. Some details of its analysis are discussed.

III. THE PIPELINED STRUCTURE ANALYSIS

Depending on the function to be implemented, different pipeline stages in an arithmetic unit require different hardware logic. Since all arithmetic operations can be implemented with the basic add and shifting operations, the core arithmetic of stages require some form of hardware to add and to shift. The arithmetic or logical shifts can be easily implemented with shift registers. High-speed addition requires either the use of a carry-propagation adder which adds two numbers and produces an arithmetic sum, or the use of a carry-save adder which adds three input numbers and produces one sum output and a carry output [4].

The proposed structure (fig.1) consists of pipeline stages, three blocks of memory, an address counter CT, three FIFO buffers, and a selector unit. The latter is introduced to provide repetitive use of operands on line W. To implement the pipeline stages of the fixed-point multiply, add and subtract unit, we use the techniques proposed and described in [9].

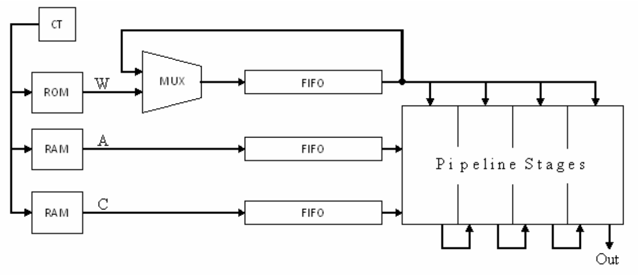

Fig.1. The pipeline structure

In general, the proposed structure is capable to execute calculations acording to the equation:

$$R_{mp\times nq} = W_{m\times n} \otimes A_{p\times q} \pm C_{mp\times nq}, \qquad (2)$$

where $\otimes$ represents the Kronecker product; W, A and C are fixed-point inputs, while R is the output of the pipeline. It is obvious from equation (2), that this structure has the ability not only to execute multiply and add/subtract operations upon the scalar operands (e.g. MAC operation), but is also able to perform matrix calculations. It should be mentioned, that the Kronecker product, denoted by $\otimes$, is an operation on two matrices of arbitrary size resulting in a block matrix. It gives the matrix of the tensor product with respect to a standard choice of basis. The Kronecker product should not be confused with the usual matrix multiplication which is an entirely different operation. So, the above structure is flexible enough and can be successfully used to implement various numerical algorithms.

In order to provide the analysis of the above specialized structure, we will denote the clock period by t. Then, we can write: $t_{pipe} = T_{CT} = T_{RAM} = t$. Next, the four stage arithmetic pipeline will be considered.

Let now provide the analysis of the bus traffic utilization when a calculation is running on our structure. Suppose $m \times n$ is the size of matrix on input W, while $p \times q$ indicates the size of matrix on input A. Then, the number of operands needed to run a calculation on the pipeline is given by relation

$$N_{OP} = (m \times n) \bullet k, \qquad (3)$$

where, k represents the number of iteration loops. Obviously, if $k = (p \times q)$, the equation (3) is written as

$$N_{OP} = (m \times n) \bullet (p \times q). \qquad (4)$$

The total number of clock periods needed to execute the calculation can be written as

$$N_{tot} = \frac{T_{tot}}{t}, \qquad (5)$$

where T_{tot} denotes the running time (full simulation time).

Evidently, to provide a complete bus traffic analysis both pipeline and operation latencies should be considered. The latency of pipeline is given by equation:

$$L_{pipe} = N_{st} \times t_{pipe}, \qquad (6)$$

where N_{st} denotes the number of stages in the pipeline.

On the other hand, the full latency of the calculation will be written in the form:

$$L_{operation} = T_{CT} + T_{RAM} + n \times T_{FIFO} + L_{pipe}, \qquad (7)$$

where n represents the number of columns in the matrix (e.g. the number of operands in the FIFO buffer).

Taking into account the above notations, we can write the equation to express the bus busy time:

$$B_{busy} = T_{CT} + T_{RAM} + k \times n \qquad (8)$$

or

$$B_{busy} = 2 \times t + k \times n, \qquad (9)$$

and the equation to express the bus free time:

$$B_{free} = k \times N_{OP} - N_{OP} + N_{st} \times t_{pipe}. \qquad (10)$$

The calculation running time T_{tot} can be now written as

$$T_{tot} = B_{busy} + L_{pipe} + k \times N_{OP} - N_{OP}. \qquad (11)$$

The above equations were used to construct the diagrams of bus traffic using versus the number of iteration loops (fig.2). It is obvious, that the bus free time is rising when the number of loops also rises. Depending on the size of n-parameter, the reducing of the bus traffic is different for different values of n. As can be seen from fig.2, the reducing of the bus traffic significantly speeds up for k = 3 and n = 16. Figure 3 illustrates the same results but in another form. It can be seen, that the bus free time is twice bus busy time, for k = 3 and n = 16.

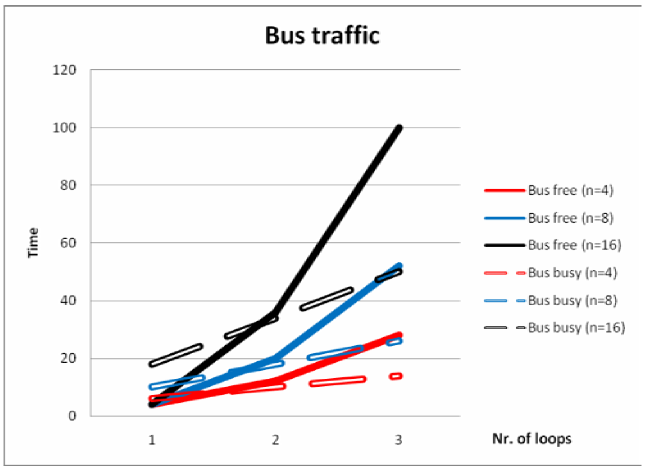

Fig.2. Bus traffic versus number of loops

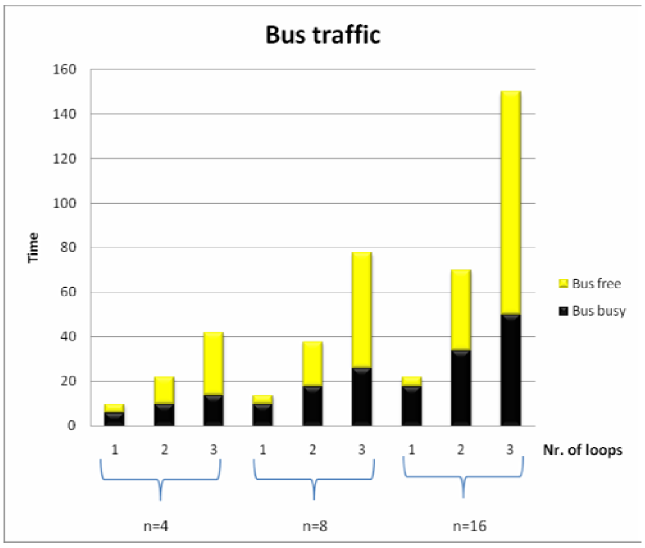

Fig.3. Bus traffic histograms

We fitted the design into a single Cyclone II family FPGA from Altera. The design (fig.4) was created by using Altera Quartus software.

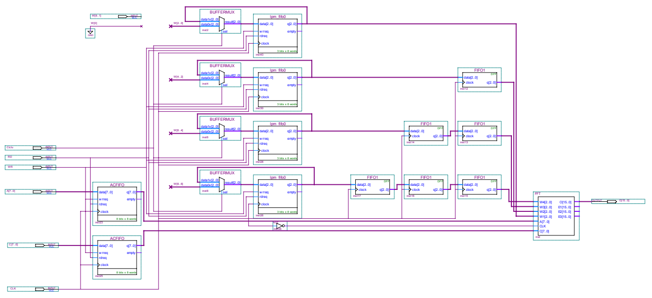

Fig.4. The schematic design

Figure 5 represents the total logic elements used while the FIFO buffer length changes from 4 to 16 words.

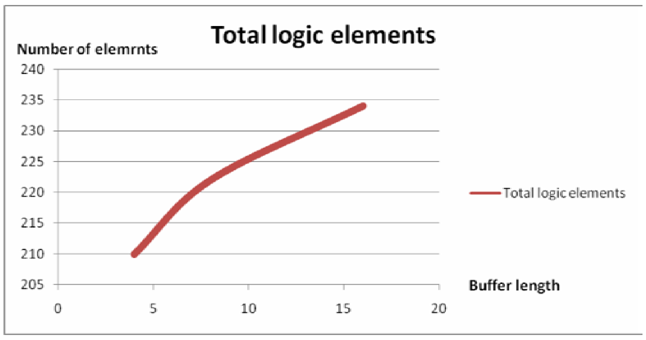

Fig.5. Hardware cost versus FIFO length

IV. SIMULATION RESULTS

To verify the above theoretical results, a lot of simulations were performed. Figures 6-10 illustrate the simulation diagrams for some cases. Figures 6 and 7 consider the calculation according equation (2) under following conditions: $m = 2;\ n = 2;\ p = 2;\ q = 3$. The matrices (12) and (13) have been used as inputs.

$$W_{2\times 2} = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}, A_{2\times 3} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \quad (12)$$

$$C_{4\times 6} = \begin{pmatrix} 1 & 2 & 5 & 6 & 9 & 10 \\ 3 & 4 & 7 & 8 & 11 & 12 \\ 13 & 14 & 17 & 18 & 21 & 22 \\ 15 & 16 & 19 & 20 & 23 & 24 \end{pmatrix} \quad (13)$$

$$R_{4\times 6} = \begin{pmatrix} 2 & 4 & 7 & 10 & 12 & 16 \\ 6 & 8 & 13 & 16 & 20 & 24 \\ 17 & 22 & 22 & 28 & 27 & 34 \\ 27 & 32 & 24 & 40 & 41 & 48 \end{pmatrix} \quad (14)$$

The output resulting matrix (14) was calculated in two ways: first by loading entire matrix W into the FIFO buffer (fig.6), and, then, by only loading a given line of that matrix at a given time (fig.7). These cases are respectively associated to equations (4), and (3).

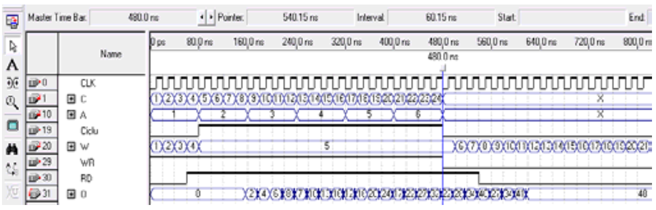
Fig.6. Loading entire matrix W

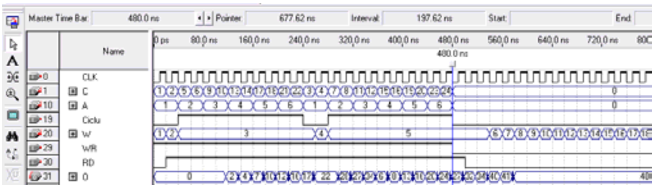
Fig.7. Loading the matrix W line by line

In fig.8 no loops are used ($k = 1$) and the FIFO on line W is only used. This mode can be used to calculate FFT. The simulation result of the multiply and add operation is given in fig.9. Both the looping and FIFO on the line W have been used. Finally, fig.10 presents a test for multiply operation (the loops have been used).

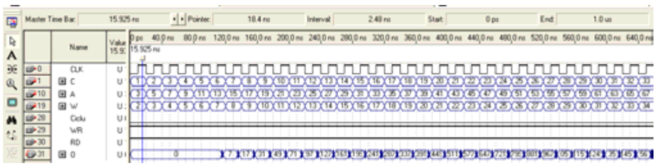
Fig.8. Calculation with no loop

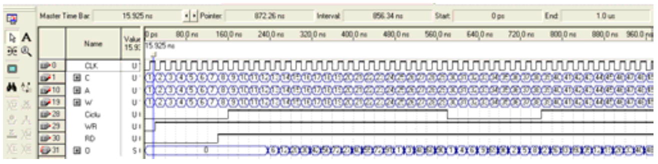
Fig.9. Multiply-Add with loop

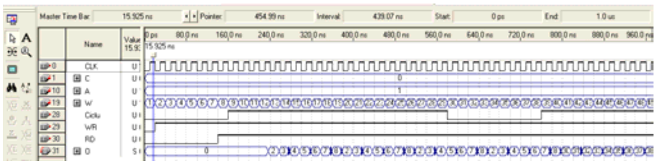
Fig.10. Multiply with loop

In all these cases the processor design was running at 50 MHz. The simulation results confirmed the theoretical results presented in the previous section.

REFERENCES

[1] R. Andraka. How to build ultra-fast floating-point FFTs in FPGAs, Retrieved April 30, 2007 from http://www.dspdesignline.com
[2] O. Brudaru, G. M. Megson, D. Galea. Systolic Algorithms in Numerical Analysis. Editura Academiei Romane, 1996.
[3] R. Cofer, B. Harding. Fixed-Point DSP and Algorithm Implementation, Retrieved October 25, 2006 from http://www.dspdesignline.com
[4] K. Hwang. Advanced computer architecture, McGraw-Hill, 1993.
[5] S. M. Mueller, W. J. Paul. Computer architecture: complexity and correctness. Springer, 2000.
[6] L. Rabiner, B. Gold. The theory and applications of digital signal processing), 1978 (In Russia).
[7] S. Riley. How to use Field-Programmable Object Arrays in image processing, Retrieved June 27, 2007 from http://www.pldesignline.com
[8] W. Stallings. Computer organization and architecture: designing for performance, 4-th edition. Prentice Hall, 1996.
[9] S. Zaporojan, V. Moraru, V. Gîscă. The Special Purpose Pipeline Arithmetic Units. Buletinul stiintific al Universitatii "Politehnica" din Timisoara. Seria Automatica si Calculatoare. Special Issue Dedicated to third int. conf. CONTI'98, Timisoara, Romania, Oct.29-30, 1998. V.43 (57), N.4, pp.238-245.