Design of Very Deep Pipelined Multipliers for FPGAs

Alex Panato, Sandro Silva, Flávio Wagner, Marcelo Johann, Ricardo Reis, Sergio Bampi
Universidade Federal do Rio Grande do Sul - Instituto de Informática
Av Bento Gonçalves, 9500, Bloco IV, Porto Alegre, RS, Brazil
e-mail: <panato,svsilva,flavio,johann,reis,bampi >@inf.ufrgs.br

Abstract

This work investigates the use of very deep pipelines for implementing circuits in FPGAs, where each pipeline stage is limited to a single FPGA logic element (LE). The architecture and VHDL design of a parameterized integer array multiplier is presented and also an IEEE 754 compliant 32-bit floating-point multiplier. We show how to write VHDL cells that implement such approach, and how the array multiplier architecture was adapted. Synthesis and simulation were performed for Altera Apex20KE devices, although the VHDL code should be portable to other devices. For this family, a 16 bit integer multiplier achieves a frequency of 266MHz, while the floating point unit reaches 235MHz, performing 235 MFLOPS in an FPGA. Additional cells are inserted to synchronize data, what imposes significant area penalties. This and other considerations to apply the technique in real designs are also addressed.

1. Introduction

Pipelines are widely used to improve the performance of digital circuits, since they provide a simple way of implementing parallelism from streams of sequential operations. As more stages are inserted in the pipeline, each stage becomes shorter, and ideally presents a smaller delay. So, the resulting circuit will exhibit bigger latency but higher sustained performance when the pipeline is fully utilized.

Theoretically, we can push the pipeline depth to a level of using a single gate between two registers. But usually, there is a compromise between performance improvements obtained with increased pipeline depth and the penalties imposed by the additional memory elements inserted in between the stages.

In [1] it is presented an 8-bit, full custom, integer multiplier using pipeline stages of a single half adder. [2] uses the same methodology to implement a two's complement multiplier. Besides the high throughput achieved, the techniques need very complex and manual work, since they employ full custom design, and work only to specific technologies and bit widths, not being accessible to regular ASIC designs.

In this work we investigate a methodology to design the deepest pipelined circuits in FPGAs, starting from VHDL. FPGA devices have some specific characteristics that allow the designer to implement a "gate level" pipeline with optimal performance, the only remark being that the word gate here means any 4-input function with a single output. Longer stages will present twice the delay of logic elements and will use an outside connection. Shorter stages do not take advantage of the fact that the FPGA cell can implement any function with the same delay. The idea already appears in an Altera Application Brief [3], but we did not find descriptions and results of a methodology or implementation anywhere else.

Despite the fact that FPGA architectures differ from vendor to vendor, they still present a set of basic common features that allow building gate level pipelines in VHDL. By doing so, it is possible to reuse and map the design to many different devices, and reuse it at the press of a button, in contrast to the full custom approach

As a case study, we developed an architecture for integer multiplication that exploits the deepest pipelines and then we build a floating point multiplication unit that is able to perform 235 MFLOPS in an Altera *Apex20KE* device. The integer architecture is parameterized to any number of bits, what increases its applicability. Yet the floating-point unit is restricted to only single precision, 32-bit, as presented in this paper, but can be easily extended to larger widths.

The rest of the paper is organized as follows. Section 2 explains how the technique is employed, and presents some information about the Altera APEX FPGA architecture. Section 3 describes the design of a parameterized array integer multiplier, along with its performance, which is compared to the default multipliers offered in the Altera library. In section 4 a complete IEEE 754 compliant floating pointer multiplier is presented, which not only achieves higher absolute frequency, but also better performance/area tradeoff. Finally, section 5 presents some discussion and our concluding remarks.

2. Deep pipelines in FPGA

An FPGA device is generally an array of configurable basic blocks called logic cells (LCs) or logic elements (LE)s. The second term is preferred and used throughout

Proceedings of the Design, Automation and Test in Europe Conference and Exhibition Designers' Forum (DATE'04)
1530-1591/04 $20.00 © 2004 **IEEE**

this paper to avoid misinterpretation. Abstracting implementation details, one can think of an LE as being composed of a look-up table (LUT), a register cell (latch, flip-flop), and a multiplexer, as it is shown in Fig. 1. The LUT can implement any truth table up to a given number of inputs, 4 in this case, although there are other simple gates outside the LUT that let the LE implement some functions with a larger number of inputs. The multiplexer selects the output from the LUT or the register as the output of the LE.

The pipeline depth of a circuit implemented in an FPGA can be pushed to the level of using the register cell in every single LE that is necessary. We may still try to put as much logic as we can inside a single LE, basically in the LUT part. But every time a combinational circuit does not fit into one LE, an additional stage is introduced. This will guarantee that there is no path longer than a single LE between any two storage elements, and is the shortest possible path between them. Therefore, our main goal is to design circuits that have this property, and to check out the performance limits that can be achieved in these devices.

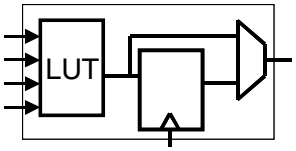

Figure 1. A simplified FPGA Logic Element

In many cases, a synthesis process with a VHDL description as input can produce gate net-lists that are not exactly what the designer wanted. But to implement pipelines at the level of logic elements, this situation must not occur, and the designer must have full control of the resulting implementation. Hopefully, using the register cells is easier that one might think. It is sufficient to include a clause that depends on the clock signal to make the output assignment of small partial functions, just as it is done in normal situations to describe a memory element. These partial functions will be contained in basic entities that may be instantiated to build the circuit under consideration.

Fig. 2 shows an example of a half adder description where the outputs are registered, and can be used as a stage of the deep pipeline. The synthesis generates two logic elements for this entity, one for the sum output S, and the other for the carry output COUT. Such an entity can be instantiated anywhere in a bigger design, and the result of its synthesis will still be the same pair of LEs.

This explains how to describe basic functions in which the design must be decomposed. The architecture of the circuit must also be adapted to allow this decomposition, since not all possible logic functions fit into a single LE. The rule when defining basic entities is that for every output, there should be at most four inputs that may affect its result, because LUTs have 4 inputs.

```
12 Architecture arch_1 of mcell1 is
13 signal soma,carry: std_logic;
14 begin
15     soma <= PP xor CIN;
16     carry <= PP and CIN;
17 process begin
18     wait until CLK = '1';
19         S <= soma;
20         COUT <= carry;
21 end process;
22 end arch_1;
```

Figure 2. VHDL of a basic block

There is another point that must be observed and significantly affects the design of the circuit. All the paths from the inputs to the outputs must pass through the same number of LEs. This is necessary to synchronize data in the pipeline. Whenever a path from an input to an output is shorter than the largest one, additional delay elements must be inserted to make the data flow at the same (logic) speed. These delay cells can be declared just as the other entities were. As it might be expected, the insertion of LEs whose sole purpose is to produce delays greatly affects circuit area, increasing device usage, and possibly limiting the application of such approach.

In our implementations, basic entities are grouped together using structural VHDL to form bigger building blocks. This hierarchy allows us to adequate the circuit structure to the FPGA architecture and to perform some placement optimizations that are described later on.

In the Apex20K family of devices, each set of 10 LEs is grouped in a structure called Logic Array Block (LAB), which in turn is grouped in sets of 10 or 16 to form the MegaLab structure. Communication between LEs in the same LAB is extremely fast, with minimal delays caused by interconnects. Connections between LEs in different LABs inside the same MegaLab have bigger delays, but are still very fast. Delays between LEs start to become critical in the interconnections when they are placed in separate MegaLabs. There is still, however, a set of "fast interconnects" between neighbor MegaLabs that can be used to keep the signals with minimal delays. But they are limited in the sense that are restricted only to neighbor MegaLabs and also because there are only a few of these fast lines. Given that a MegaLab is not square, the system will run out of horizontal connections first.

3. Design of an Integer Multiplier

In order to test the gate level pipeline technique just explained, an integer array multiplier was first designed. The fastest types of multipliers are the parallel ones, as Wallace [5] and Dadda [6] architectures. However, these architectures do not have the same regular structure already present in the Array multiplier [7]. As Fig. 3a shows, each line of logic cells computes basically one partial product, and could be a separate pipeline stage. A comparison among this approach and the parallel

architectures for an ASIC are found in [8]. We chose the Array architecture as the starting point. By inspection one can see that delays are also propagated horizontally. Therefore, if partial products were used as pipeline stages, each stage would have to wait for the propagation of carry signals and will then have the delay of many LEs.

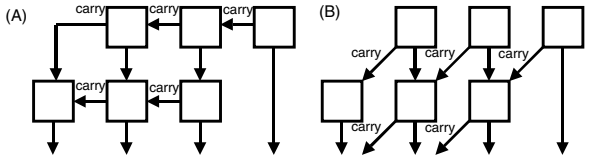

Figure 3. Classic and adapted carry propagation

So, in order to implement the gate level pipeline, there were two options. The first one was to consider the circuit as running diagonally from top-right to bottom-left, and the other was to adapt the carry propagation to be taken into account only at the next stage, as Fig. 3b shows. We chose the second approach, as we expected it to minimize the amount of additional delay elements to be inserted.

3.1 Multiplier architecture

Fig. 4 shows an example of a 4-bit integer multiplier where it is possible to observe the elaborated architecture. The next two sections explain the basic blocks and the intermediate level structure, respectively.

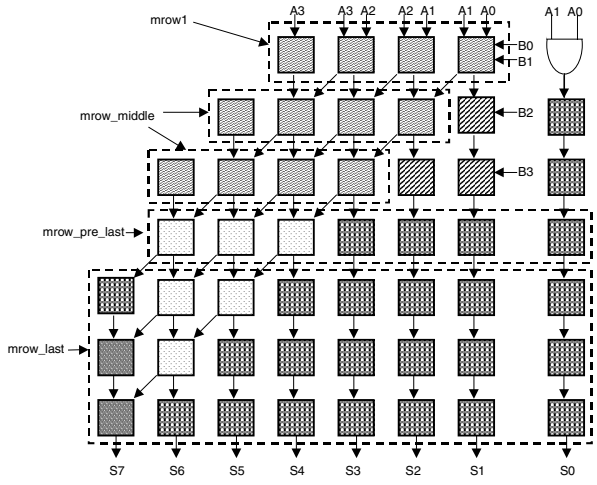

Figure 4. 4-bit array multiplier

3.2 Basic building blocks

Six types of basic blocks are needed to implement the integer multiplier (see Fig. 5), and they are:

a) A half adder that adds results and carry, called mcell1;
b) A multiplication block that computes the sum of two single bit multiplications, called mcell2a;
c) A multiplication block that computes a single bit multiplication and runs the result into a full adder, called mcell2;
d) A block for propagation only, called mcell3;
e) A half adder without carry, called mcell4;
f) A double delay block, to propagate input B and results, called mcell5;

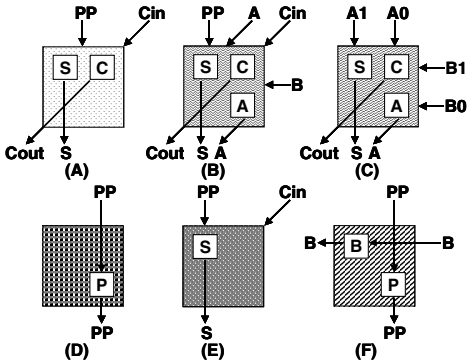

Figure 5. Basic building blocks

3.3 Intermediate level structure

Instances of the basic blocks are used to assemble intermediate level, regular blocks, that perform specific tasks in the multiplier (see Fig. 4). The four intermediate level structures are:

mrow1: Corresponds to the first stage of the pipeline and is composed of *n* mcell2a cells. Two bit multiplications are possible at this stage since it does not have a previous one, and bit multiplications are implemented with AND gates. The output of this structure is a vector of *n+1* bits for results and a vector of *n-1* bits for carry out.

mrow_middle: Corresponds to the next *n-2* stages. Each stage uses *n* blocks of mcell2 and some blocks of mcell5 for propagation of previous results.

mrow_pre_last: The propagation of inputs A and B are no longer needed. So, the n^{th} stage uses *n-1* instances of mcell1 for carry adjust and *n* instances of mcell3 for propagating previous result.

mrow_last: Implements the *n-1* last stages of the pipeline, performing only carry adjust in the most significant bits. Each line uses one instance of mcell4, except the first, who uses a mcell3 instead, and instances of mcell1 and mcell3, starting with *n-2* instances of mcell1 and *n+1* instances of mcell3. In the following lines, the number of mcell1 instances is decreased by 1, and the number of mcell3 instances is increased by 1.

There are also adjacent structures for mrow1 and mrow_middle described in the top entity to compute the least significant bit of the first stages. The first one uses a simple AND gate, and the next ones synchronize B inputs and propagate the result of the least significant bit.

3.4 Latency and Logic Elements prediction

We described the architecture of the multiplier in a parameterized way, so that it can be instantiated for any required number of bits. Since the implementation is highly regular, both latency and circuit size can be predicted. The latency will always be *(2*n-1)* cycles.

The number of LEs used in each basic block is well known, and corresponds to the number of small squares shown in Fig. 5. So, it is possible to estimate the total size of the resulting circuit in number of LEs by the following equation, for a multiplier of *n* bits:

$$\#LEs = 5 \cdot n^2 - 3 \cdot n - 2 + \sum_{i=1}^{n-2} 3 \cdot i$$

Table 1 presents the latency and circuit size prediction for commonly used data widths. The 24 and 54 bit widths are used in the floating point IEEE 754 standard, which will be discussed in section 4.

Table 1. Latency and size prediction.

#bits	Latency	#LEs
4	7	75
8	15	357
16	31	1545
24	47	3565
32	63	6417
54	107	18550
64	127	25915

3.5 Implementation and Simulation Results

We tested the resulting performance of the integer multiplier synthesized for a range of bit widths from 4 to 64. Table 2 shows in the second column the operating frequency achieved in each circuit. The first thing to note is that the performance obtained is very high for this kind of device. In fact, we investigated why the 4 and 8 bit circuits presented the same performance, and found out that there is a limitation in the device due to the clock distribution. This leads us to two conclusions. The first is that these versions of 4 and 8 bits could possibly operate in higher frequencies provided that their paths are not the limiting factor. And the second is that our results are very close to the limiting aspects of the FPGA technology, and will be hardly outperformed. But the performance still drops a little as the bit widths become wider. This is due to the delay of interconnects, which increases with the increase in circuit size. The third column shows the operating frequency of multipliers generated by the Altera´s MegaWizard tool. We can see in the next column the improvement that we get over this standard solution (up to 3 times). Of course, it is very important to note that our latency is really giant compared to other solutions, and therefore the technique should not be used as a common solution. However, in many applications, we do have the scenario where uninterrupted multiplication sequences are needed, such as in filters, other DSP operations, in such a way that a long latency will be accepted.

Table 2. Operating frequencies.

#bits	UFRGS	Altera	ratio
4	290	279	1,04
8	290	200	1,45
16	266	137	1,94
24	217	120	1,80
32	218	107	2,04
54	165	43,3	3,84
64	141	50,7	2,78

However, the main drawback of this approach is the circuit size. Table 3 compares the device usage of our solution against the one resulting from the Altera multipliers. It is possible to see that our penalty in area is bigger that our gain in performance for most of the circuits. Although deep pipeline still wins in the 54 bit operations, circuit size may limit severely the application of such technique. It might be better to replicate standard components, which not only operate in parallel but also have lower latency.

Table 3. Comparing circuit sizes.

#bits	UFRGS	Altera	ratio
4	75	34	2,21
8	357	160	2,23
16	1545	560	2,76
24	3565	1256	2,84
32	6417	2160	2,97
54	18550	6188	3,00
64	25915	8610	3,01

3.6 Placement optimizations

Although the simulation reveals very high clock rates, they may still be enhanced if we can reduce the longer interconnect paths that start to appear in larger circuits. One option is to develop a specific placement method for this architecture. But from the designer´s perspective, we are left with the option of manually placing the synthesized cells.

In order to investigate its impact on performance, we manually adjusted the placement of a 32-bit multiplier using the Logic Lock tool available in Altera Quartus II, version 2.2. With this tool it is possible to specify regions where VHDL entities must be placed.

For best performance, blocks with strong interaction shall be close, preferably in the same MegaLab. We grouped the design in each pipeline stage.

These groups must be placed according to the data flow, starting from mrow1, following the mrow_middle instances, then the mrow_pre_last and finally to the mrow_last instances. Fig. 6 shows an example of such a placement for the first rows. These first groups include also the path traversed by the least significant bits.

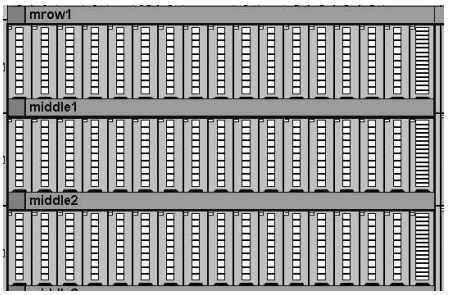

Figure 6. Region assignment with Logic Lock.

We have manually placed the blocks at the same column of MegaLabs in the FPGA, just changing to a neighbor column when the first one is filled up. This is better that placing them line by line, since there are more fast lines in the vertical direction, as it was described before. In fact, one of the most sensible points was at the turn point where we change the column. Whenever this turn is placed, the performance drops strongly because we run out of horizontal fast lines and the connections start to use slower ones.

The solution to this problem was to slightly alter the original placement of each group in a MegaLab, from the case depicted in Fig. 7a to the one shown in Fig. 7b. In this case, each group we initially planed to place in a single MegaLab is split in two parts, and placed at vertically neighbor MegaLabs. By doing so, we increase the neighborhood of the last group in a column with respect to the first group of the next column, and they have access to twice as many fast interconnect lines as they had before.

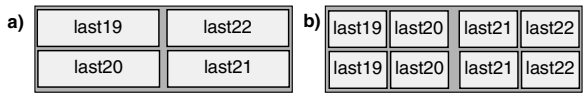

Figure 7. Placement options between columns.

After the manual placement, we measured the performance of the 32-bit multiplier, and it increased from 218 to 246 MHz, resulting in a gain of 14,2%. It is still possible to find other placement options that will increase the performance further, and this data indicates that additional research may be conducted with this goal, such as the development of specific placement tools for circuits like this.

Although the placement optimizations just described are restricted to the chosen FPGA architecture and device, the overall methodology employed is applicable to other devices from vendors like Xilinx and Altera, without any changes in the VHDL code.

4 IEEE 754 Floating Point Multiplier

Here we present an implementation of a single precision floating point multiplier following the IEEE 754 standard [4]. This implementation is important to demonstrate that the very deep pipelines may result in circuits that outperform standard solutions by larger margins. In this standard, the first bit represents the mantissa's signal, the next 8 bits the exponent and the other 23 bits the mantissa, but without its implicit value of 1 at the integer part. For more details, the reader may refer to the standard´s document [4]. The architecture developed is presented in Figure 8.

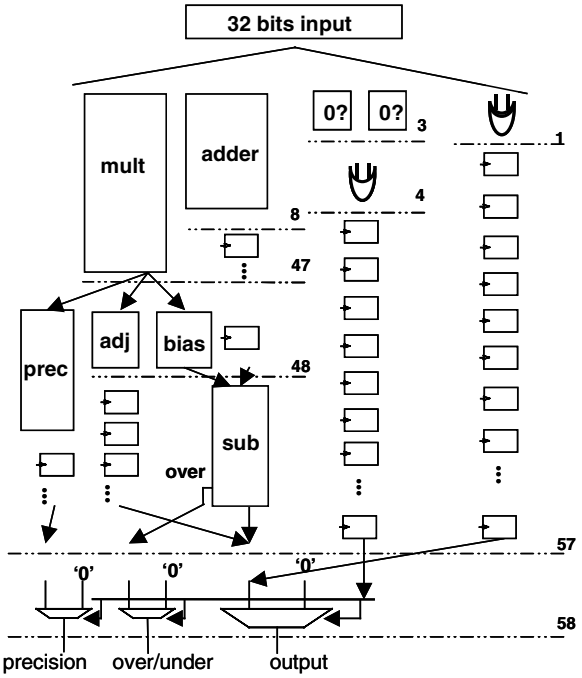

Figure 8. IEEE 754 floating point multiplier.

There is a **xor** gate (at top right) that determines the signal bit of the result, which is propagated through the other 56 stages (1 to 57). In the next column from the right we can observe comparators that check whether one of the operands is zero or not (called '0?'), which require 4 clock cycles, and whose result is propagated for 53 stages (4 to 57). This stage is necessary to set the output to zero if at least one of the inputs was zero.

The exponent of the result is computed using an adder and a subtractor described in the same way used to develop the multiplier, and will not be detailed in this paper. The adder adds the exponent of both inputs while the subtractor removes the extra bias introduced in this operation, also indicating if there was underflow or overflow in the multiplication. The multiplication uses a 24-bit integer multiplier described in section 3, and requires 47 clock cycles to finish.

Since the standard defines that the most significant bit of both operands is one, the result will have at least one number 1 in the two most significant bits (1X or 01). So, if the most significant bit is 1, it is necessary to normalize both mantissa and exponent. The first normalization is performed by just changing the lines connected to a multiplexer (in the 'adj' structure). To normalize the exponent, another multiplexer is introduced that selects two possible values to correct the bias (in the 'bias' structure). If the operation needs normalization, the value assigned is 128, and otherwise 127.

To indicate the precision, a 3-cycle structure is introduced after multiplication. It just compares the rejected values of mantissa to zero. If all values are zero, the precision bit is assigned. The least structure is a block of multiplexers. If at least one input is zero, all values are zero. Otherwise, the computed result is used.

Table 4 shows the results of synthesis and simulation of this single precision unit of multiplication, compared to the multiplication unit generated by the Altera´s MegaWizard tool.

Table 4. Floating point multiplication results.

Team	LCs	xAltera	Freq	xAltera	Pipeline
UFRGS	4434	2,68	235	3,73	58
Altera	1655	--	62,9	--	5

It is possible to see that in this case, our proposed methodology and architecture produces a result that greatly outperforms the standard solutions. Furthermore, while our multiplier is 3.7 times faster than the Altera multiplier, it is only 2,68 times bigger, and we have an absolute gain in the performance/cost relation.

It is also important to note that the floating-point multiplier has almost the same performance of the integer-multiplier alone (recall table 2). This is expected, since the delay is proportional to the delay of each stage in the pipeline, and the technique investigated keeps the stages limited to single logic elements in the FPGA.

5. Conclusions

This work investigated a methodology to design very deeply pipelined circuits in FPGAs, starting from VHDL. A specific architecture for integer and floating-point multiplication was developed that demonstrates the applicability of the methodology. The results greatly outperform other solutions in terms of clock rates, and in absolute performance/cost ratio in the case of the floating-point operation. The 16-bit integer multiplier achieves a frequency of 266MHz, while the floating-point unit reaches 235MHz, performing 235 MFLOPS in an FPGA.

The proposed method presents an intrinsically larger latency, but there are applications in which latency is not a limiting factor, but high throughput is desired. In this approach, it is possible to keep similar operating frequencies for larger circuits, since delays depend only on paths that go from one LE to another, e.g. of a single step of logic and connection.

However, when scaling up the circuits, as in the data widths, the additional delay elements may cause the resulting size to become too large, affecting the performance, as longer interconnections are used, at least when considering only generic vendor tools for placement.

In smaller circuits, the performance limit is due to the clock skew of an FPGA, that limit the maximum frequency used in the device. This is a strong indication that we are very close to the performance limits of the device, and generating circuits that operate at this limit.

A possible limitation of the approach is the nature of 2D propagation of signals in some kinds of circuits, as it happens in the multiplier. Carry propagation in the integer multiplier needs more stages in both dimensions, and causes more cells to be inserted. Yet, the floating-point multiplier does not increase the complexity and delay compared to the integer unit, as its datapath runs only in one direction.

We are currently working in a variation of the array multiplier architecture that could reduce the area due to register-only cells (now 40% of device usage). However, the application of this technique only for the design of multipliers might not be too much appealing, even because there are dedicated multipliers in modern FPGAs that can exhibit better performance. Our experiments can be seen as proofs of concept, and the methodology should be extended to other kinds of circuits.

6. References

[1] NOLL, Tobias G., SCHMITT-LANDSIEDEL, Doris, KLAR Heinrich & ENDERS Gerhard. A Pipelined 330-MHz Multiplier. *IEEE J. Solid-State Circuits*, vol sc-21, no 3, June 1986.

[2] SOMASEKHAR, Dinesh & VISVANATHAN, V. A 230-MHz Half-Bit Level Pipelined Multiplier Using True Single-Phase Clocking. *IEEE Trans on VLSI Systems*, vol 1, no 4, December 1993.

[3] ALTERA. Pipelined Multipliers in FLEX 8000 Devices. Application Brief 134. May 1994, version 1.

[4] IEEE Standard for Binary Floating-Point Arithmetic in ANSI/IEEE Std 754-1985. New York, USA, 1985.

[5] WALLACE, C. S. A Suggestion for a Fast Multiplier. *IEEE Transactions Electronic Computer*. EC-13:14-17, 1964.

[6] DADDA, L. Some Schemes for Parallel Multipliers. *Alta Freq*. 34:349-356. 1965.

[7] BROWN, Stephen D. Fundamentals of Digital Logic with VHDL designs. Boston: McGraw-Hill, 2000.

[8] STILES, Bryan W. & SWARTLANDER, Earl E. Jr. Pipelined Parallel Multiplier Implementation. *Conference Record of the Twenty-Seventh Asilomar Conference on Signals, Systems and Computers, 1993*. November 1993.