Comparison of Pipelined Floating Point Unit with Unpipelined Floating Point Unit

M Venkateswarao, J Triveni, K Bhargavee Latha, K Naga Deepika, KR Pavan
Department of Electronics and Computer Engineering, K L University, Andhra Pradesh, INDIA
mekavrao@gmail.com, trivenijatla05@gmail.com, kondraguntabhargavi@gmail.com,
deepika.katepalli@gmail.com, pavankr42@gmail.com

Abstract

Floating-point numbers are broadly received in numerous applications due their element representation abilities. Floating-point representation has the capacity hold its determination and exactness contrasted with altered point representations. Any Digital Signal Processing (DSP) calculations utilization floating-point math, which obliges a huge number of figuring's every second to be performed. For such stringent necessities, outline of quick, exact and effective circuits is the objective of each VLSI creator. This paper displays a correlation of pipelined floating-point snake dissention with IEEE 754 organization with an unpipelined viper additionally protests with IEEE 754 arrangement. It depicts the IEEE floating-point standard 754. A pipelined floating point unit in light of IEEE 754 configuration is produced and the outline is contrasted and that of an unpipelined floating point unit and an investigation is defeated speed, range, and force contemplations. It builds the rate as well as is vitality productive. Every one of these changes is at the expense of slight increment in the chip region. The basic methodology and approach used for VHDL (Very Large Scale Integration Hardware Descriptive Language) implementation of the floating-point unit are also described. Detailed synthesis report operated upon Xilinx ISE 11 software and Modelsim is given.

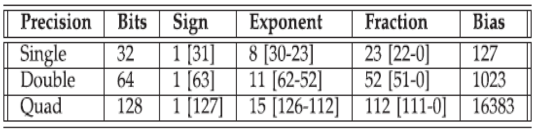
Keywords: *Floating point unit, IEEE 754 standard format, Pipelining, VHDL, Xilinx ISE, Modelsim*

1. Introduction

In advanced digital signal processors (DSP) and modern microprocessors, micro chips floating-point unit is an imperative component. The architectures grew so far for floating point unit is in view of arrangement of huge operations: shift, swap add, normalize and round. Because of these operations, the aggregate methodology backs off. Floating-point unit ought to be quick to match with the increasing clock rates demanded by profound sub-micron technologies, likewise they must be small for being utilized as a part of parallel processing systems. Since, in the customary unit, all the stages were performed with single clock cycle, yet the frequency of this clock was confined because of circuit constraints. Subsequently, at whatever point, the operation of an extensive number of values was performed, the conventional floating-point unit proved to be inefficient. This latency could be overcome if the idea of pipelining is introduced in the simple adder. The floating- point unit had been subdivided into four stages, which were pipelined based on the proper timing sequences. The clock frequency, which could be utilized for these stages, could be higher when contrasted with the clock frequency utilized for customary floating-point adder. Additionally, while the two inputs are being processed and passed on to subsequent stages, new inputs enter the introductory stage and the cycle proceeds. These outcomes in overall quicker operation. In this paper, the floating-point algorithm is clearly explained and floating point unit implementation using Very Large Scale Integration Hardware Descriptive Language (VHDL) is described. It further deals with the core theory of pipelining and the enhanced potential of the floating-point unit. Simple floating-point unit and pipelined unit have been compared in terms of speed of operation and area on chip. Table 1 show the bit ranges for single (32-bit) and double (64-bit) precision floating-point values

Received December 20, 2014; Revised February 1, 2015; Accepted February 15, 2015

Table 1. Bit range of single, double and quadruple precision FPU



Precision	Bits	Sign	Exponent	Fraction	Bias
Single	32	1 [31]	8 [30-23]	23 [22-0]	127
Double	64	1 [63]	11 [62-52]	52 [51-0]	1023
Quad	128	1 [127]	15 [126-112]	112 [111-0]	16383

2. Floating Point Representation

2.1 Single Precision Floating Point Representation

Sign bit decides the sign of the number, which is the sign of the significand also. Exponent is either a 8 bit signed integer from −128 to 127 (2's Complement) or a 8 bit unsigned integer from 0 to 255 which is the acknowledged one-sided shape in IEEE 754 binary32 definition. In the event that the unsigned integer configuration is utilized, the exponent quality utilized as a part of the number-crunching is the exponent moved by a predisposition for the IEEE 754 binary32 case, an exponent estimation of 127 speaks to the real zero

The genuine significand incorporates 23 fraction bits to one side of the binary point and a certain driving bit (to one side of the binary point) with worth 1 unless the exponent is put away with all zeros. Hence just 23 fraction bits of the significand show up in the memory organize yet the aggregate precision is 24 bits (equal to log10(224) ≈ 7.225 decimal digits)
The bits are laid out as follows:

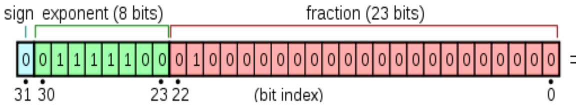


Figure 1. Single Precision Floating Point Representation

In IEEE-754 format, the significant bit always takes on an implied ‘1’ for the most significant digit assuming that the value represented is normalized.

2.2 Double Precision Floating Point Representation

Double-precision binary floating-point is a generally utilized arrangement on PCs, because of its more extensive territory over single-precision floating point, disregarding its execution and transfer speed expense. Similarly as with single-precision floating-point design, it needs precision on integer numbers when contrasted and an integer configuration of the same size. It is ordinarily referred to just as double. The IEEE 754 standard indicates a binary64 as having
Sign bit: 1 bit
Exponent width: 11 bits
Significand precision: 53 bits (52 explicitly stored)

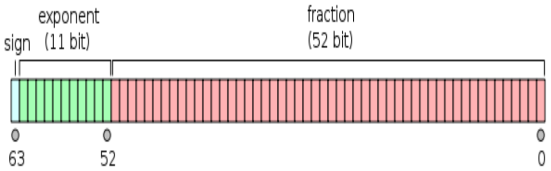


Figure 2. Double Precision Floating Point Representation

2.3 Quadruple Precision Floating Point Representation

This 128 bit quadruple precision is planned not just for applications obliging results in higher than double precision, [1] additionally, as an essential capacity, to permit the processing of double precision comes about all the more dependably and precisely by minimizing flood and round-off blunders in moderate figurings and scratch variables: as William Kahan, essential engineer of the first IEEE-754 floating point standard noted, "Until further notice the 10-byte Extended organization is a middle of the road bargain between the estimation of additional exact math and the cost of actualizing it to run quick; soon two more bytes of precision will get to be passable, and at last a 16-byte group.

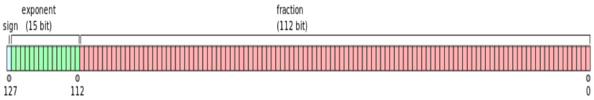


Figure 3. Quadruple Precision Floating Point Representation

3. Floating Point Arithmetic

The IEEE-754 standard is the most generally utilized floating point representation for genuine numbers on PCs. Numerous codes permit or oblige that some or all number-crunching be done utilizing IEEE-754 arrangements and operations. This area gives an extremely short outline of the standard, to help in better comprehension the proposed system. Far reaching data on floating point number juggling can be found in .floating point representation basically speaks to genuine numbers in exploratory documentation. Experimental documentation communicates numbers as a base and an example. Case in point, 123.45 could be spoken to as 1.2345*10^2 or 12.345 * 10^1. IEEE 754 floating point numbers have three parts: the sign, the exponent, and the mantissa. The mantissa is made out of the portion and a verifiable driving digit (i.e., 1). The example base (i.e., 2) is likewise certain and require not be put away. Consequently, IEEE-754 gliding point numbers are spoken to as (-1)^sign *1.fraction * 2^exponent. Table 2 demonstrates the format (the quantity of bits and the bit ranges for every field) for the most ordinarily utilized precisions. quadruple precision is seldom actualized in equipment and is typically performed by programming traps. The sign bit is 0 for positive numbers and 1 for negative numbers. The exponent field needs to speak to both positive and negative exponents. To do this, an inclination is added to the real exponent to get the put away exponent. For single accuracy floating-point numbers, this worth is 127. In this manner, an exponent of zero implies that 127 is put away in the exponent field. A put away estimation of 200 shows an exponent of (200- 127), or 73. The mantissa speaks to the accuracy bits of the number. To amplify the amount of representable numbers, floating-point numbers are put away in standardized structure. To expand the amount of representable numbers, floating-point numbers are put away in standardized structure. This fundamentally puts the radix point after the first nonzero digit. For instance, 123.45 in floating point would be (-1)^0* 1.9289062 *2 ^6 in single exactness. Thusly, the put away representation would be 42F6E666 in hexadecimal representation.

Finally, exponent field values of all 0s and all 1s are held by IEEE to denote special values in the floating point plan. Zero is a special value denoted with an exponent field of zero and a fraction field of zero. While -0 and +0 are particular values because of the extra sign bit, they both compare as equal. The values -infinity and +infinity are denoted with an exponent of all 1s and a fraction of all 0s. The sign bit recognizes negative infinity and positive infinity. Having the capacity to denote infinity as a particular value is helpful because it allows operations to proceed past flood situations. Operations with limitless values are very much characterized in the IEEE-754 standard. Finally, the value not a number (NaN) is utilized to speak to a value that does not speak to a real number. NaN is spoken to by a bit pattern with an exponent of all 1s and a nonzero fraction

There are two categories of NaN: quiet NaN (QNaN, most significant fraction bit set, propagates freely through most arithmetic operations) and signaling NaN (SNaN, most significant fraction bit clear, signals an exception when used in operations). Semantically, QNaN denotes indeterminate operations, while SNaN denotes invalid operations.

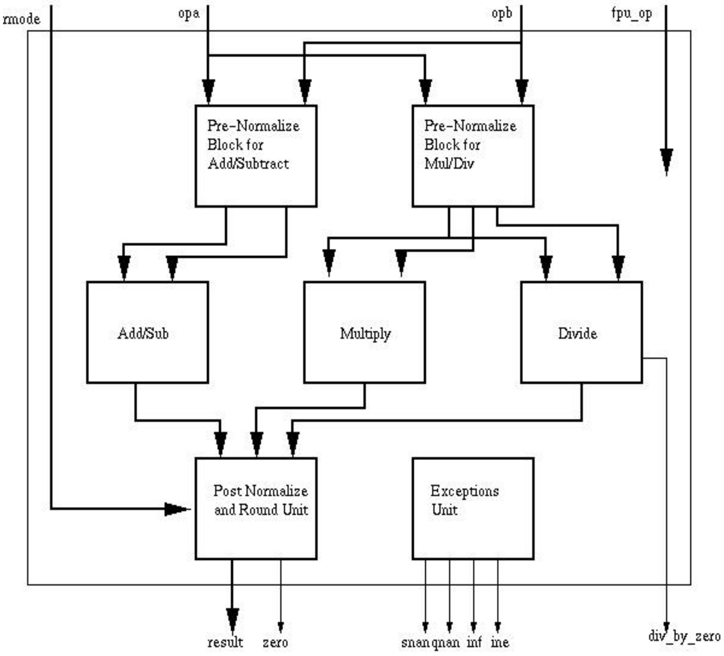

rmode
opa
opb
fpu_op
Pre-Normalize Block for Add/Subtract
Pre-Normalize Block for Mul/Div
Add/Sub
Multiply
Divide
Post Normalize and Round Unit
Exceptions Unit
result
zero
snan
qnan
inf
ine
div_by_zero

Figure 4. Floating Point Arithmetic

3.1 Calculating the Exponent

In this area, we talk about the exponent estimation for each of the three sorts of FPU capacities, to be specific number-crunching operations, changes, and different operations. We take note of that the exponent is computed freely of the part operation; consequently, the outcome is not correct on the grounds that conceivable portion standardization may influence the last estimation of the exponent

3.2 Arithmetic Operations

The main classification is math operations, for example, addition, subtractions, multiplication, and divisions. We remind that the IEEE-754 representation of standardized floating-point operands is (-1)^s *1.f*2^e, where s is the sign, f is the portion, and e the exponent. Subsequently, exponent
count in duplication and division is straightforward

((-1)^s1*1.f1*2^e1)*((-1))^s2*1.f2*2^e2)
=(-1)^s1+s2 *1.f1 *1.f2 *2^e1+e2 (1)

for multiplication and

((-1)^s1*1.f1*2^e1)/((-1))^s2*1.f2*2^e2)
=(-1)^s1+s2 *1.f1 /1.f2 *2^e1-e2 (2)

for division. In this way, we just need to include (for augmentation) or subtract (for division) the exponents, operations which can be performed by the same equipment structure. On the off chance that the division floods and needs to be standardized, the exponent needs to be balanced in like manner. Henceforth, for number juggling operations, we can just anticipate the exponent of standardized results with a +or-1 precision. Therefore, if a mistaken result varies from the right result by 1, slip covering will happen. Then again, our guess is that in the vicinity of a control rationale blunder, the information way is defiled widely; henceforth, the likelihood of such concealing is low.

3.3 Conversions

Another regular operation performed in FPUs is transformation from/to number/floating-point representations. The exponent of the outcomes can be precisely ascertained by fittingly balancing the information operand. For floating point accuracy transformations (single to twofold and the other way around), the exponent needs to be counterbalanced by _896, in light of the fact that the genuine exponent is es _ 127 in single exactness and ed _ 1;023 in twofold accuracy. Hence, for single to twofold change, the exponent is ed = (es-127)+1023 and for double to singles es=(ed-1023)+127

4. Pipelined Floating Point Arithmetic

Pipelining, a system for attaining to speedier clock rates while relinquishing dormancy, offers a financial approach to acknowledge worldly parallelism in computerized frameworks. To accomplish pipelining, info process must be subdivided into a grouping of subtasks, each of which can be executed by specific equipment organize that works simultaneously with different stages in the pipeline. The snake outline pipelines the steps, looking at, swaping, shifting, addition and standardization to accomplish the summation each clock cycle. Every pipeline stage performs operations free of others. Data information to the snake persistently streams in. As is extraordinary, legitimate pipelining expands the throughput of floating-point adders . With a specific end goal to attain to a fitting pipelined combined floating-point add–subtract unit, the latencies of the segments in the proposed outline are explored. Every part is actualized in Verilog HDL and combined with the Nangate 45-nm innovation standard-cell library. Every pipeline stage is executed each cycle so that the biggest idleness decides the throughput of the configuration. Figure 5 demonstrates the information stream, the dormancy of every part, and the critical path

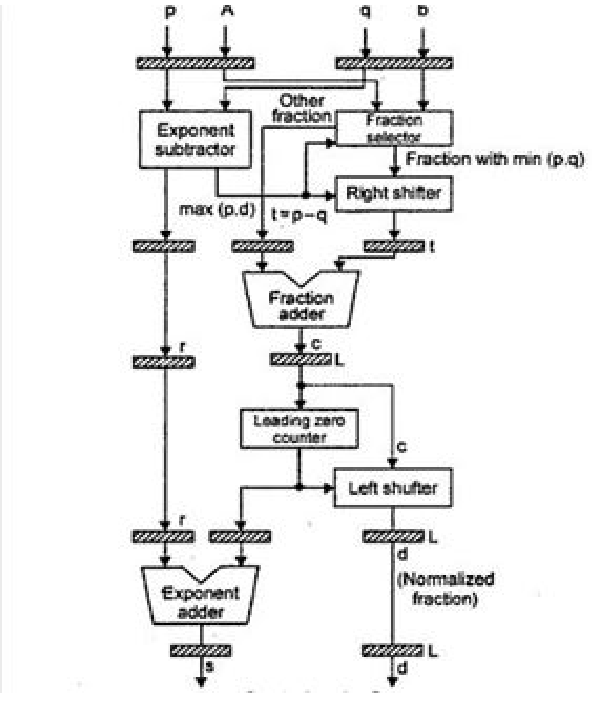


Figure 5. Pipelined Floating Point Arithmetic

4.1 Operations

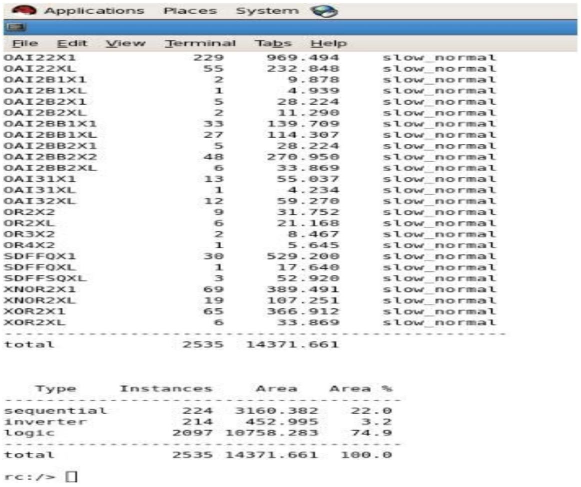
Cutting edge FPUs generally execute more operations, for example, total worth, refutation, and correlation. In every one of these operations, the exponent is exceptionally easy to compute. Nullification/supreme worth operations influence just the sign (i.e., the exponent is the same). Correlation operation last estimation of the exponent. results are usage particular, as the yield result is the examination result and not a floating-point number. For instance, SPARC ISA characterizes the exponent field of the yield as 0, and the examination result is put away in the banners field.

Table 2. Outlines the exponent operation for regular FPU operations

Operation	Result Exponent	Fraction Normal.	Sign
Addition	$max(e_1, e_2)$	Yes	$sign(max(e_1, e_2))$
Subtraction	$max(e_1, e_2)^1$	Yes	$sign(max(e_1, e_2))$
Multiplication	$e_1 + e_2$	Yes	$sign(e_1) \oplus sign(e_2)$
Division	$e_1 - e_2$	Yes	$sign(e_1) \oplus sign(e_2)$
Single to Double	$e_1 + 896$	No	$sign(e_1)$
Double to Single	$e_1 - 896$	No	$sign(e_1)$
Negation	e_1	No	$-sign(e_1)$
Absolute Value	e_1	No	$+$

5. RESULTS

5.1 Power and area



```
OAI22X1                229     969.494     slow_normal
OAI22XL                 55     232.848     slow_normal
OAI2B1X1                 2       9.878     slow_normal
OAI2B1XL                 1       4.939     slow_normal
OAI2B2X1                 5      28.224     slow_normal
OAI2B2XL                 2      11.290     slow_normal
OAI2BB1X1               33     139.709     slow_normal
OAI2BB1XL               27     114.307     slow_normal
OAI2BB2X1                5      28.224     slow_normal
OAI2BB2X2               48     270.950     slow_normal
OAI2BB2XL                6      33.869     slow_normal
OAI31X1                 13      55.037     slow_normal
OAI31XL                  1       4.234     slow_normal
OAI32XL                 12      59.270     slow_normal
OR2X2                    9      31.752     slow_normal
OR2XL                    6      21.168     slow_normal
OR3X2                    2       8.467     slow_normal
OR4X2                    1       5.645     slow_normal
SDFFQX1                 30     529.200     slow_normal
SDFFQXL                  1      17.640     slow_normal
SDFFSQXL                 3      52.920     slow_normal
XNOR2X1                 69     389.491     slow_normal
XNOR2XL                 19     107.251     slow_normal
XOR2X1                  65     366.912     slow_normal
XOR2XL                   6      33.869     slow_normal
---------------------------------------------------------
total                 2535   14371.661


   Type      Instances     Area     Area %
-----------------------------------------
sequential         224   3160.382    22.0
inverter           214    452.995     3.2
logic             2097  10758.283    74.9
-----------------------------------------
total             2535  14371.661   100.0

rc:/> []
```

Figure 6. Power consumption of pipelined floating point unit

5.2 RTL schematic

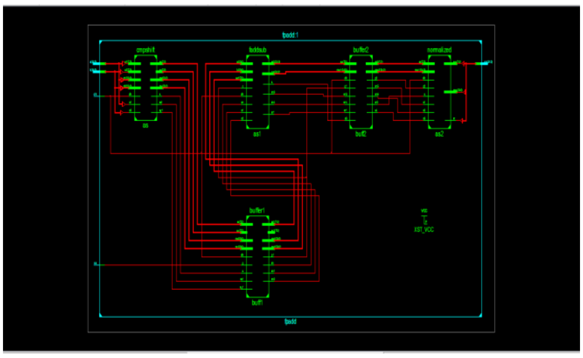


Figure 7. RTL schematic for pipelined FPU

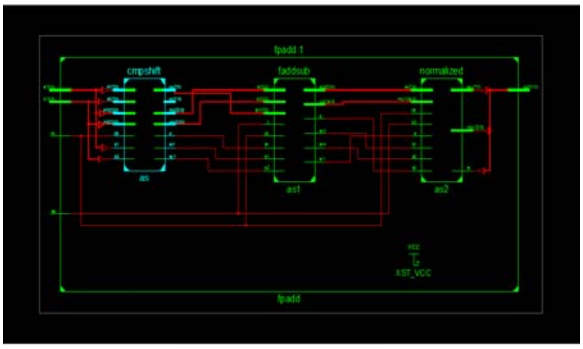


Figure 8. RTL schematic for pipelined FPU

5.3 output

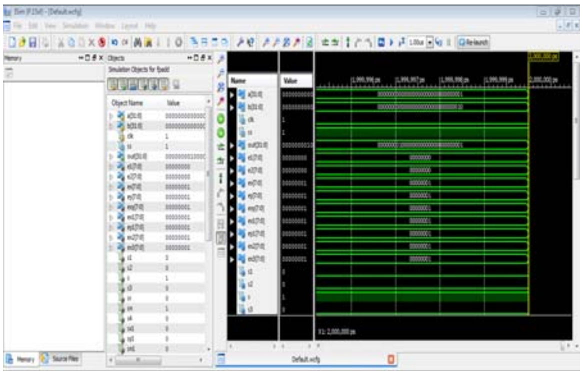


Figure 9. Output for pipelined FPU

6. Conclusion

Architectures for design and implementation of floating point unit is presented. The floating point unit has many applications .This unit consists of number of interconnected floating point adders, subtracters, multipliers, and wordblock. This paper presents improved architectures which apply pipelining to the floating point and compares the area, latency, throughput, and power consumption with the traditional floating point unit. Here we are implementing pipelining to increase the throughput of the floating point unit .As it uses pipelining stages they were well balanced and throughput increases. In the pipelining technique we are using two buffer stages so that we can give many number of inputs when compared to traditional fused floating point unit .In the buffer itself the inputs were waited for one clock cycle .By implementing the pipelining , 40% of power is consumed.

References

[1] IEEE Standard for Floating-Point Arithmetic, ANSI/IEEE Standard 754-2008, New York: IEEE, Inc., Aug. 29, 2008.

[2] RK Montoye, E Hokenek, and SL Runyon. “Design of the IBM RISC system/6000 floating-point execution unit”. *IBM J. Res. Develop*. 2000; 34: 59–70.

[3] E Hokenek, RK Montoye, and PW Cook. “Second-generation RISC floating point with multiply-add fused”. *IEEE J. Solid-State Circuits*. 2001; 25(5): 1207–1213.

[4] T Lang and JD Bruguera. “Floating-point fused multiply-add with reduced latency”. *IEEE Trans. Comput*. 2004; 53(8): 988–1003.

[5] CH Ho, CW Yu, PHW Leong, W Luk, and SJE Wilton. “Floating-point FPGA: Architecture and modeling”. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst*. 2009; 17(2): 1709–1718.

[6] YJ Chong and S Parameswaran. "*Flexible multi-mode embedded floating-point unit for field programmable gate arrays*". In Proc. FPGA. 2009: 171–180.

[7] AM Smith, GA Constantinides, and PYK Cheung. "*Fused-arithmetic unit generation for reconfigurable devices using common subgraph extraction*". In Proc. ICFPT. 2007: 105–112.

[8] CW Yu, AM Smith, W Luk, PHW Leong, and SJE Wilton. "*Optimizing coarse-grained units in floating point hybrid FPGA*". In Proc. ICFPT. 2008: 57–64.

[9] G Govindu, L Zhuo, S Choi, and V Prasanna. "*Analysis of high-performance floating-point arithmetic on FPGAs*". In Proc. Parallel Distrib. Process. Symp. 2004: 149–156.

[10] M Langhammer. "*Floating point datapath synthesis for FPGAs*". In Proc. FPL. 2008: 355–360.

[11] F de Dinechin, C Klein, and B Pasca. "*Generating high-performance custom floating-point pipelines*". In Proc. FPL. 2009: 59–64.

[12] V Betz, J Rose, and A Marquardt. Architecture and CAD for DeepSubmicron FPGAs. Norwell, MA: Kluwer. 2002

[13] PM Seidel and G Even. "Delay-optimized implementation of IEEE floating-point addition". *IEEE Trans. Computers*. 2004; 53(2): 97–113.

[14] D Tan, CE Lemonds, and MJ Schulte. "Low-Power Multiple-Precision Iterative Floating-Point Multiplier with SIMD Support". *IEEE Trans. Computers*. 2009: 58(2): 175-187.

[15] T Lang and JD Bruguera. "Floating-Point Multiply-Add-Fused with Reduced Latency". *IEEE Trans. Computers*. 2004; 53(8): 988-1003.

[16] V Zyuban, D Brrok, V Srinivasan, M Gschwind, P Bose, PN Strenski, and PG Emma. "Integrated analysis of power and performance for pipelined microprocessor". *IEEE Trans. Comput*. 2004; 53(8): 1004–1016.