A 1.5 Ghz VLIW DSP CPU with integrated floating point and fixed point instructions in 40 nm CMOS

Timothy Anderson, Duc Bui, Shriram Moharil, Soujanya Narnur, Mujibur Rahman, Anthony Lell, Eric Biscondi, Ashish Shrivastava, Peter Dent, Mingjian Yan, Hasan Mahmood
Texas Instruments, Dallas, Texas, USA
Communication Infrastructure – DSP Systems

***Abstract*— A next generation VLIW DSP Central Processing Unit (CPU) which has an integrated fixed point and floating point Instruction Set Architecture (ISA) is presented. It is designed to meet a 1.5 GHz core clock frequency in a 40nm process with aggressive area and power goals. In this paper, the benchmarking process and benefits of newly defined instructions such as complex matrix multiply is explained. Also, the CPU datapath is described in detail, highlighting several novel micro-architecture features. Finally, our design methodology as well as verification methodology to ensure functional correctness utilizing formal equivalent verification is described.**

Keywords- CPU, DSP, Floating Point, Fixed Point, Multiply, Complex Matrix Multiply, Additions, SIMD, Formal Verification

I. INTRODUCTION

As wireless communications and video applications become dominant driving forces, it is clear that latest Digital Signal Processors need to not only perform fixed point arithmetic but also floating point. Along with more required data bandwidth, more computations per cycle are also needed. In this paper, we describe the benchmarking process to define new ISA enhancements through micro-architecture details and methodologies used in designing the latest VLIW DSP CPU from Texas Instruments, the TMS320C66x. This new CPU has improved vector processing capabilities with the addition of SIMD instructions that operate on 128-bit vectors of 16-bit or 32-bit quantities, achieving 32 multiply operations per cycle. General processing power is also improved by shortening the latencies of key floating point operations like multiply and addition. Also, more specific instructions are added to target complex linear algebra and video processing. The TMS320C66x CPU provides up to 5 times improvement over the previous generation, TMS320C64x CPU, on key wireless benchmarks such as FFT or MMSE [3].

II. BENCHMARKING & ISA ENHANCEMENTS

Many benchmarks critical to the targeted communication signal processing were identified to be significant bottleneck on the fixed-point-only TMS320C64x architecture. For instance, MIMO MMSE [3] solution requires inverting a matrix and multiplying the result with another matrix. It was found that to maintain the desired precision and overall Bit Error Rate (BER), an appropriate scaling at various stages of

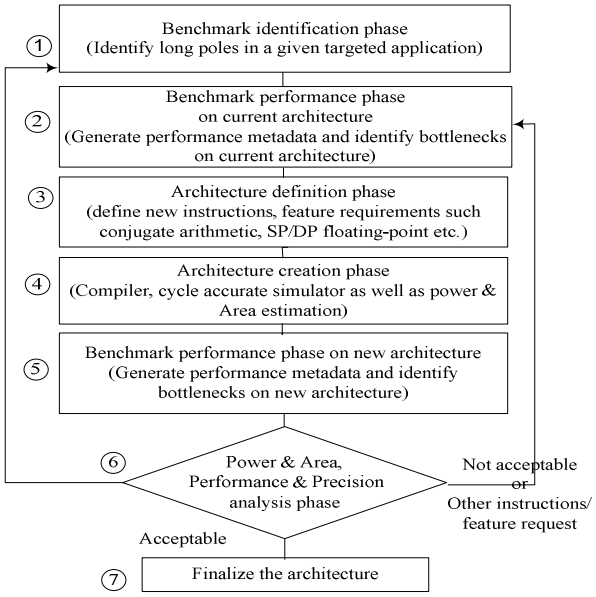

Figure 1. High level benchmarking methodology

the algorithm is required to avoid overflow/underflow. Without scaling, there was up to 0.5dB degradation in BER performance. On the older TMS320C64x architecture, different flavors of scaling were supported on the fixed point functional units. Therefore, the MMSE implementation is heavily bounded by fixed point functional units. The inverse and scaling part of a symbol based 2x4 MMSE was then benchmarked using an older Texas Instruments floating-point architecture, the TMS320C67x. Single precision IEEE floating-point data [4] does not require such scaling since it has 24-bit precision and higher dynamic range. The performance data collected for this mixed fixed-point and floating-point symbol based MMSE benchmarks shows 3.2 times improvement over the TMS320C64x design. This motivates us to enhance our next generation CPU, the TMS320C66x, to have integrated fixed point and floating point architecture. Also, recent wireless standards like LTE and WiMAX require the implementation of advanced receivers which heavily use complex linear algebra. In this context, a large number of complex matrix operations, including matrix multiplies, have to be computed. Therefore, more parallel multiply operations on the multiplier unit is needed, as well as more SIMD instructions utilizing those

1063-6889/11 $26.00 © 2011 IEEE
DOI 10.1109/ARITH.2011.20

multipliers, along with complex conjugation supports, and SIMD instructions to do format conversion between 16-bit/32-bit integers and floating point data type. In particular, we added a new Complex Matrix Multiply (CMATMPY) instruction, which can perform all the required operations for the multiplication of one complex vector of size [1x2] by a complex matrix of size [2x2]. CMATMPY uses two source operands and produces a 128-bit data field that represents two packed complex numbers. Each produced complex number is represented with two 32-bit signed values (real and imaginary). The combination of multiple CMATMPY instructions can be used to compute larger complex matrix multiplication. Various flavors of the CMATMPY instruction enable the conjugation of one operand as well as the selection of the output precision. With these new complex instructions added and complex floating-point support, a 5.5 time improvement over the baseline TMS320C64x implementation was seen for mixed fixed-point and floating-point symbol based 2x4 MMSE benchmark. Similar exercises were performed for other critical benchmarks, and on doing trade-off between different criteria such as improvement factor, power, performance, area and time to market, the architecture for next generation VLIW DSP CPU was finalized.

III. DATAPATH OVERVIEW

This new TMS320C66x CPU supports more than 320 instructions, with approximately 80 instructions newly added over the previous generations, to accelerate general DSP applications as well as targeted instructions to address broadband communications, wireless communications, and video applications. It is capable of executing eight instructions per cycle. Many of these are SIMD instructions, which operate on multiple data elements that have been packed together into a wide vector. The CPU datapath is divided into 2 identical half-datapaths with four functional units in each (Fig. 2):

- Add & Logic unit (.L): Comprised of two modules. One is the floating point module which performs floating point add and conversion instructions. The other is the fixed point module which performs fixed point add, along with all logical operation instructions.
- Add & Shift unit (.S): similar to the .L unit but also supports all shifts, including left shift with saturation, bit-field extracts.
- Multiply unit (.M): perform all fixed and float multiply instructions using a common multiply array.
- Data Address unit (.D): performs 32-bit fixed point add instruction and all memory load and store instructions.

Each half-datapath also includes one 32 entry, 32-bit register file (RF). Each RF has 6 write ports and 10 read ports, which enable simultaneous reads/writes of source/result operands from all functional units.

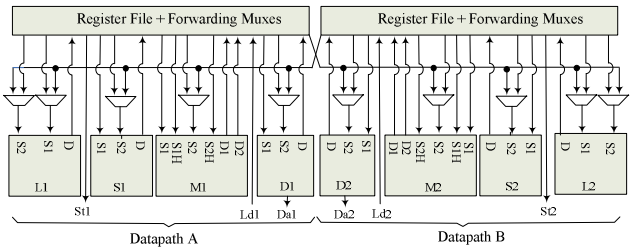

Figure 2. Datapath Overview

Furthermore, the register files are divided into odd and even banks of registers to reduce the porting requirements when using 64-bit operands. The division of the RF into odd and even registers effectively yields a 64-bit read/write capability for each port. In addition, a "cross-path" bus enables each half-datapath to read a single operand stored in the opposite half-datapath's RF. This cross-path bus has been enhanced relative to the prior generations, from 32-bits wide to 64-bits wide to support more data transfer bandwidth between the two half-datapaths.

IV. FUNCTIONAL UNIT DETAILS

A. M-unit combined multiplier

High performance DSP requires the flexibility to operate on many different formats of data. Data may be integer fixed point data – signed or unsigned, real or complex, 32-bit, 16-bit or 8-bit, or it may be floating point data – single-precision real, single-precision complex, or double precision. This integrated multiplier unit performs all these operations using a single set of multiplier arrays. (Fig. 3)

1) General operation

The multiplier array takes in two operands, src1 and src2. The src1 and src2 buses are 128-bits wide to support SIMD operations such as multiplying four respective 32-bit words in each 128-bit source, returning four 32-bit results. Not all operations require 128-bit source operands – many take in 64-bit operands, and a few only take 32-bit operands. In any case, most instructions will send the corresponding word lane of src1 and src2 to each respective "multiply cluster". The notable exceptions to the rule are:

- Double precision floating point multiply
- Complex 32-bit multiply (both 32-bit integer, and 32-bit single precision floating point)
- Matrix multiply – vector * matrix

In the above three cases, the lower 64-bits of src2 are duplicated and sent to clusters 2 & 3, and similarly for src1 except for the matrix multiply case (since the matrix is a 128-bit operand on src1). Each source is sent through masking logic before being sent to the multiplier clusters. There are three different types of masking that occur. The first type is actuated when a double precision multiply operation is taking place. Recall that the IEEE double precision floating point format [4] has a 53-bit mantissa, where 52-bits come from the source operand, and the 53rd bit is an implicit 1. The masking logic will zero out bits 63 through 53 of the src1 and src2 operands before sending to

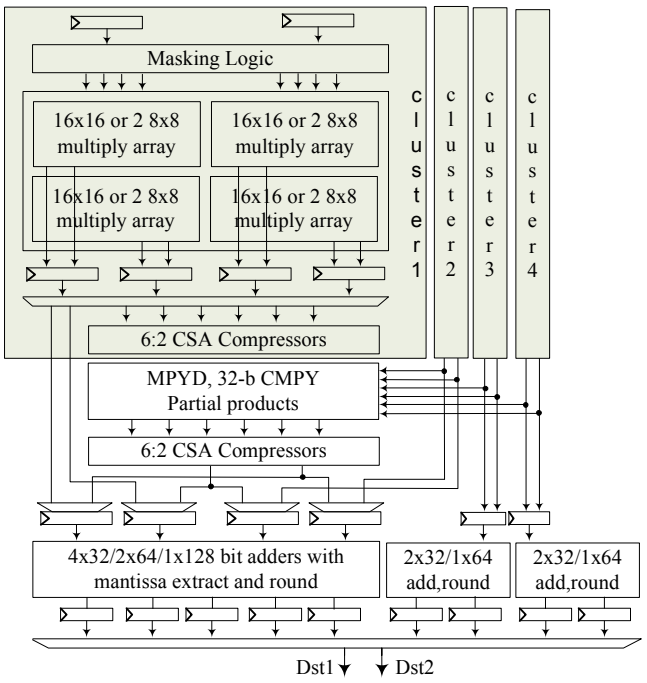

Figure 3. Combined multiply unit

the multiply clusters, and will set bit 52 of the incoming operands. The second type of masking is actuated when single precision multiplies are to be calculated. In this case, bits 31 through 24 of each word in the incoming source operands are set to zero, and bit 23 is set to a 1 before sending to the multiply clusters. The final type of masking occurs for the 2-way SIMD dot product instructions. In this case, inputs to the cross multipliers (LxH and HxL) are zeroed out.

2) Multiply Cluster Operation

Each multiply cluster receives two 32-bit sources, src1 and src2. Each cluster contains four 16x16 multipliers, which we'll label MPY0, MPY1, MPY2, and MPY3. MPY0 multiplies src1[15:0] by src2[15:0], MPY1 multiplies src1[31:16] by src2[31:16], MPY2 multiplies src1[15:0] by src2[31:16], and MPY3 multiplies src1[31:16] by src2[15:0]. For the sake of brevity, we'll refer to the result of MPY0 as LxL, MPY1 as HxH, MPY2 as LxH, and MPY3 as HxL. Note that each multiplier array output is not the complete result of "a * b," but rather is the final two compressed products from the Wallace tree. In other words, each multiplier outputs two 32-bit words that when added together will produce the actual product. We'll call these results LxL_a, LxL_b, HxH_a, HxH_b, etc. The multiplexors in front of the 6:2 Carry-Save-Adders (CSA) are used to provide different combination of the outputs of the multipliers to the 6:2 CSA. In addition, the 6:2 CSA is configurable such that the carries from bit position 31 do not propagate to bit position 32. The six inputs to the 6:2 CSA will be referred to as PP0, PP1… PP5. Depends on the operation, they are configured as below:

Product	Operation	Product	Operation
PP0	$HxH_a << 32 \mid LxL_a$	PP3	$HxH_b << 32 \mid LxL_b$
PP1	$LxH_a << 16$	PP4	$LxH_b << 16$
PP2	$HxL_a << 16$	PP5	$HxL_b << 16$

- 32x32 multiplication:

Note that there is no space for rounding bits when calculating the 32x32 result. However, some fixed point multiply instructions also requires rounding. This is achieved by inserting a rounding bit inside one of the 16x16 multipliers in the appropriate bit location, and off the critical path so that there is no timing impact.

- 16-bit real, 16-bit imaginary complex number multiplication

Product	Operation	Product	Operation
PP0	$HxL_a << 32 \mid \sim LxL_a$	PP3	$HxL_b << 32 \mid \sim LxL_b$
PP1	$LxH_a << 32 \mid HxH_a$	PP4	$LxH_b << 32 \mid HxH_a$
PP2	<rounding vector >	PP5	0x00000000

Note that imaginary numbers are represented as a 32-bit number, with the upper 16-bits as the real component, and the lower 16-bits as the imaginary component. Therefore, the real portion of the result will be HxH – LxL, and the imaginary portion of the result will be HxL + LxH. The above muxing will produce the complex results out of the 2x32 bit adders following the 6:2 CSA. In addition, the 6:2 CSA in this case will be injected a "+2" carry input into unused locations in the LSBs. This carry injection will not affect the critical path. The +2 carry input arises because we need to have a full 2's complement on both LxL_a and LxL_b.

- 16-bit real, 16-bit imaginary complex number multiplication, one operand conjugated

Product	Operation	Product	Operation
PP0	$\sim HxL_a << 32 \mid LxL_a$	PP3	$\sim HxL_b << 32 \mid LxL_b$
PP1	$LxH_a << 32 \mid HxH_a$	PP4	$LxH_b << 32 \mid HxH_b$
PP2	< rounding vector >	PP5	0x00000000

This case is similar to the ordinary complex multiplication except that one of the input operands has had its imaginary portion negated. This changes the equations to

Result_real = HxH + LxL

Result_imaginary = LxH – HxL

In this case, the extra +2 must be added from bit position 32. We accomplish this by setting bit 33 in the rounding vector for this case. All other cases use the 32-bit datapath, and may disable the 32-bit carry in the 6:2 CSA (for example when we want to get HxH and LxL as separate results). Double precision multiplication is achieved through performing a 64-bit x 64-bit multiplication. The 64x64 multiplication is built up from the results from each of the multiply clusters the same way that we build a 32x32 result from the four

16x16 multipliers inside the multiply clusters. In addition, the 32-bit real, 32-bit imaginary results are handled the same way as the 32-bit cases, just on wider data. This can be seen in the diagram with the final 6:2 CSA in front of a 128-bit adder. In addition, there is a floating point pipeline which is run in parallel to the main datapath, which handles the exponent calculations. The final 2:1 adder may end up incrementing the final exponent. The final result multiplexing will join the exponent, sign, and mantissa when required. This solution optimizes area between floating point and integer multiplication operations – previous solutions had separate fixed and floating point multipliers. In addition, the technique of keeping all the partial products in redundant form for all intermediate additions saved logic delay, enabling 4 cycle multiplier operation for all operations. Previously, it would take 10 cycles for Double Precision multiplication due to extra cycles needed to quadruple-pumping the data through a single 32x32 multiply array as well as the need to double-pumping the source operands.

B. L-unit and S-unit floating point adder module

The floating point add module resides inside the L and S units and can perform two Single Precision (SP) floating point additions or one Double Precision (DP) floating point addition per cycle, along with data conversions between fix and float formats, and table-lookup estimation functions for 1/x and 1/ √ x. In the prior TMS320C67x generation, a SP ADD takes 4 cycles to complete, while a DP ADD takes 7 cycles. The 3-extra cycles for DP ADD were needed for 64-bit data reads and writes since the previous generations RF only supported 32-bit read/write per port. Double-pumping of the 32-bit ports was used to obtain the 64-bit operands. With the RF enhanced to 64-bit read/write ports, we've eliminated the extra 3-cycle overhead over the legacy DP ADD instruction, which allowed for a 4-cycle latency floating point ADD pipeline. After further analysis, the floating point adder was redesigned using a faster dual-path implementation instead of the previously slower conventional single-path design [5], [6]. This helps the design to fit comfortably in 3-cycle pipeline, does not significantly increase area and still allows for some floor plan flexibility (Fig. 4). Both single precision and double precision additions now take only 3 cycles as compared with 4 and 7 cycles respectively as before. Since this new dual-path adder design fit into 3-cycle pipeline, some tradeoffs were made to reduce more area/power. For instance, instead of predicting the number of leading zeros for normalization, we used the simpler but slower method of detecting leading zeros, after the compound adder has computed the result. Our floating point adder may not be as fast as those optimized-for-delay designs [7], but the cost between speed and area/power consumption is balanced to meet our own objectives. The module also supports all format conversions between SP floating point, DP floating point and 32-bit integer. Furthermore, conversions between two packed SP values and two packed 16-bit integer values are also supported. These conversion functions were implemented

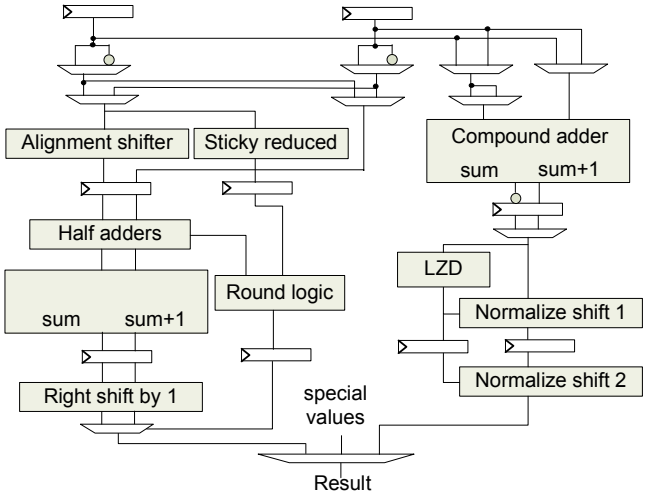

Figure 4. Floating Point dual path add pipeline

using the existing main blocks of the floating point add pipeline such as the alignment shifter, the adder and the LZD circuit. Therefore only a small extra area overhead was needed. Another place where we were able to save area as compared with previous design is to have the floating point compare instructions implemented by reusing the existing compare logic of the fix point module.

C. L-unit and S-unit fixed point module

1) Golden Cycle Path

The critical path of the L-unit is the "golden cycle" compared to which all other paths must be faster. The golden cycle path is the traditional 32-bit Saturation Add/Subtract computation where the carry out of bit 30^{th} is used to compute saturation condition to select either the normal sum result or a constant value. To implement subtract, we avoid the usual costly "plus 1" of 2's complement operation by using this identity:

$$Sum = a - b = a + NOT(b) + 1 = NOT(NOT(a) + b)$$

2) ABS Logic Block

Among other instructions, the L-unit implements the complex conjugate rotate (CROT), absolute-value (ABS) The general arithmetic performed by the CROT and ABS instructions involves negation of one operand. These instructions can be executed using the above "golden path" circuit, but requires forcing a constant value into one of the operand and therefore worsens the critical path. Therefore, negation is done using a separate and simplified circuit. Since negation only requires a carry-in of "1" to the LSB, only the "group-propagate" terms in a standard parallel-prefix carry tree are needed. The "group-generate" terms are not required, as there is no chance of generating a carry at any position other than the LSB. Thus the standard parallel-prefix carry-tree can be considerably optimized for implementing the negate operation. This optimized carry tree is therefore much faster and its area and power cost is greatly reduced. As a side benefit, this optimized carry tree

is also used to compute the logical-AND instruction required in the ISA.

3) Saturating Left Shift

.Besides normal 32-bit left/right shift, signed/unsigned extract and rotate, the .S unit also supports 40-bit left and right-shift and 32-bit left-shifts with saturation. The shifter module implements a special circuit for computing 32-bit left shifts with saturation in order to reduce the number of logic levels. Saturation occurs when values between two adjacent bits of the shift data are different, and the shift amount is large enough to translate that change into an overflow out of the entire data. The least significant 3 bits of the shift amount are used to calculate whether saturation can occur within an 8-bit block. The most significant 2 bits of the shift amount are then used to determine which of the 8-bit blocks is actually saturating. The overall saturate condition is then simply the logical-OR of the four modules which operate on each 8-bit field.

V. VERIFICATION METHODOLOGY

Datapath units like adders and multipliers have large number of input bits and is impractical to exhaustively exercise all input patterns with injected test vectors. Furthermore, several instructions require specialized corner-case handling. Writing focused test vectors to test each special condition is a difficult and time consuming task and there is still no guarantee that all cases are covered. Therefore, a flow has been developed to formally verify the design. Each instruction is modeled as a function using the C programming language. The C code is considered golden as it serves as the basis for benchmarking the design as well as running applications. The C code is then translated to behavioral RTL using a vendor tool. This auto-generated RTL is then compared to the implemented RTL, using a standard equivalence checking tool. The equivalence checking treats all registers as wires. This is done to simplify the checking methodology, as cycle accuracy can add a great deal of complexity to the equivalence checking process. Such simplification does not hurt the robustness of this methodology since it is only used for checking datapath computation results, with the assumption that all input operands staying static for the duration. The equivalence checking can also be constrained to verify the design on a per-instruction basis, in order to restrict the comparison to only valid opcode values. In parallel, normal functional tests are run on the implemented RTL to ensure cycle accuracy of the design. Thus, this formal verification methodology serves as an important add-on to the regular design verification flow using generated test vectors.

VI. DESIGN METHODOLOGY

The design methodology used has a number of unique features but builds upon basic, well-proven techniques. The circuit design style generally follows a standard-cell static-CMOS logic synthesis environment. This is extended with support for pass-transistor, hot-encoded muxes and hot-encoded mux-flops for timing and power optimizations. Initial synthesis was completed for each unit with its own floorplan to enable early timing fixing through out the design cycle. At the CPU level, a flat top-level floor plan was used. However, soft regions were used to put each unit to a specific placement, which matched to its unit-level floor plan to help guide the tools. To fine-grain pipeline balancing, clock skewing is used judiciously by either moving clock edges backward or forward. To minimize the amount of hold buffers involved and therefore power, we limit skewing to at most 1 clock buffer delay. Also, clock-gate cells and their corresponding registers are grouped together using relative placement to minimize clock skews and clock power. The library contains multi-threshold static CMOS cells with four different threshold voltages (Vt) for each cell: high Vt, extra-long-gate high Vt, normal Vt and extra-long-gate normal Vt. This enables static leakage power reduction for the non-critical paths. The most critical parts of the datapath such as the fixed point adder and register files are coded with direct cell instantiations and the tools are only allowed to size the cells up or down. This way, we can strictly control the synthesis result and ensure that the tools yield expected results. The rest of the datapath is coded with mostly structural RTL to help guide the tools achieve desired timing closure. Timing analysis was performed using static timing analysis with built-in signal integrity support to handle extra delay due to noise and capacitive coupling.

VII. CONCLUSION

We were able to improve the overall DSP performance up to 5 times with only 1.5x increased in area/power. Furthermore, the CPU critical paths were not affected by the extra logic that was added. We accomplished this through judicious design of our new datapath modules so that we can get the best performance with the least area increase. In the end, we were able to achieve a competitive balance between performance improvements and power saving by adding new, useful instructions along with novel design and methodology while minimize area and power.

REFERENCES

[1]. M.J. Flynn, “Very high-speed computer systems”, Proceedings of the IEEE, 54(12):1901--1909, December 1966

[2]. R. Lidl and H. Niederreiter, “Introduction to Finite Fields and Their Applications”. Cambridge: Cambridge University Press, 1986

[3]. Arogyaswami J. Paulraj, Dhananjay Gore, Rohit U. Nabar, and Helmut Bolcskei, “An Overview of MIMO Communications –A Key to Gigabit Wireless”, Proceeding of the IEEE, Vol, 92, No.2, Feb 2004

[4]. ANSI/IEEE Std 754-1985 "IEEE Standard for Binary Floating-Point Arithmetic", 1985

[5]. M.P. Farmwald, “On the design of high performance digital arithmetic units”, PhD thesis, Stanford University, 1981

[6]. Nhon T. Quach , Michael J. Flynn, “An improved algorithm for high-speed floating-point addition”, Stanford University, Stanford, CA, 1990

[7]. P.M Seidel, G. Even, “Delay-optimized Implementation of IEEE Floating Point Addition”, IEEE Transactions on Computers. Vol.53 issue2, February 2004