FPU Generator for Design Space Exploration

Sameh Galal, Ofer Shacham, John S. Brunhaver II, Jing Pu, Artem Vassiliev, Mark Horowitz
Stanford University
Email: {sameh.galal,shacham,jbrunhav,jingpu,tema8,horowitz}@stanford.edu

***Abstract*—FPUs have been a topic of research for almost a century, leading to thousands of papers and books. Each advance focuses on the virtues of some specific new technique. This paper compares the energy efficiency of both throughput-optimized and latency-sensitive designs, each employing an array of optimization techniques, through a fair "apples to apples" methodology. This comparison required us to build many optimized FP units. We accomplished this by creating a highly parameterized FP generator, hierarchically encompassing lower-level generators for summation trees, Booth encoders, adders, etc. Having constructed this generator we quickly relearned a number of low-level issues that are critical and are often the most neglected by papers. By exploring cascade and fused multiply-add architectures across a variety of bit widths, summation trees, booth encoders, pipelining techniques, and pipe depths, we found that for most throughput based designs, a Booth-3 fused multiply-add architecture with a Wallace combining tree is optimal. For latency designs, we found that Booth-2 cascade multiply-add architectures are better. As we describe in the paper, Wallace is not always the optimal combining network due to wire delay and track count, and the precise way the CSA's are connected in the tree can make a larger difference than the type of tree used.**

I. Introduction

Since its introduction in 1914 [1] floating point arithmetic has been an important component in computer design. There are thousands of research articles describing clever ways to build math hardware leading to many books and conferences on this subject. Yet after many decades of research in this area, building a high-performance floating point unit is still a difficult task. This difficulty can be partially explained by the fact that the engineering challenges and manufacturing technologies have changed over time, which clearly changes the "optimal" set of circuits. In early designs, the number of devices in a chip were limited, so most attention was paid to packaging and fitting the function into the hardware, while today energy efficiency is the critical concern. Yet, changes in technology and design goals cannot be the complete story: CMOS has been the dominant technology for almost 30 years, and energy efficiency has been a key goal for over a decade.

In our view much of the problem arose (paradoxically) from the large amount of research done in this field. Commercial FPUs were highly optimized at circuit, logic, and architecture levels, and unless you were an expert at all of these levels, your design would not be competitive. Thus researchers could claim that their idea was a good one; it was slower than current designs for reasons that had nothing to do with their new change. On the other hand, it also could be the case that if you used a better base design, the differential that they saw would disappear.

To try to address this issue we leveraged a new design method and tool that supports designers creating a flexible generator for a circuit module, rather than simply creating the needed module [2], [3], [4]. These tools encourage designers to encode their knowledge about how a unit should be built into the generator. This makes the unit "smart" and able to reoptimize for the demands of a particular instance. Using this approach it is possible to create a multiplier generator from a Booth generator, and a generator that can sum partial products. This multiplier generator (rather than its output instance) can then be used as a module in a FPU generator. Section II describes the ideas behind building generators in more detail.

Having studied FPU trade-offs for a number of years, we decided to try to encode the knowledge that we learned into a generator, to allow us (or others) to create high-performance, functionally correct floating-point units easily. We also hoped this would allow us to better understand how various research proposals fare in creating a real FPU. After completing the FPU generator, we noticed that other designs occasionally beat our performance, which caused us to look for tricks that our system did not use. Once we found a trick, the modular nature of our generator allowed us to apply that optimization to all designs where it applied, since once a module was improved, every design that used that generator also improved.

Section III focuses on trade-offs in the design of a multiplier, and explains many of the subtle, but critical issues that affect multiplier performance. While these optimizations are well-known by experts in the field, by incorporating them into a generator, we were then able to do truer comparisons of system level trade-offs. Section V demonstrates this by looking at the trade-offs between different fused and cascade multiply-add implementations, pointing out which design choices lie on the energy optimal curve.

II. A FPU Generator

Recently a number of researchers have created languages for designers to create more flexible hardware units [3], [5]. The goal of these languages is to allow a designer to provide a procedural description of how the hardware should be instantiated, given a set of parameters that are fixed at the time the design is instantiated. It extends the capabilities of custom generators such as FPGA optimized FloPoCo [6] and library based data path generators [7] to a general infrastructure allowing designers to build their own hierarchical, extensible generators. The ability to use a full programming

1063-6889/13 $26.00 © 2013 IEEE
DOI 10.1109/ARITH.2013.27

language to ”compute” the needed hardware greatly extends SystemVerilog’s *parameters* and *generate blocks.*

A direct use case that demonstrates the usefulness of this type of procedural information is creating irregular structures such as Wallace trees. While the algorithm to create a Wallace tree is not complex, it is not regular enough to be encoded using the parametrization in SystemVerilog, which is why many people create scripts to build Wallace trees when they are needed. These new generator languages differ in that they hierarchically build more complex generators. That is, rather than using a script-based generator to create an instance of a Wallace tree to be used in a multiplier, they create a *multiplier generator* that calls a Wallace tree generator to create the combining network.

In building this hierarchical generator, the parameters that are explicitly exported at the top level become the architectural parameters of a module. In our FPU example, these are the bit-widths of the mantissa and exponent, the desired rounding modes, whether the unit should implement IEEE rounding/underflow/overflow, etc. But many of the parameters of the lower level blocks are not explicitly set by these system parameters. They then fall into two groups. Some of these parameters are constrained by either the parent’s parameters or design, and thus are effectively set by the parent (and/or its parameters). Others are not constrained in this current instance, and can be varied to optimize the design. Examples in a FPU include whether and which type of Booth encoding is used; the type of combining tree used in the multiplier; how the adder is connected to the multiplier; pipeline depth, etc. A complete generator needs to have a procedure for setting these parameters to optimize the design. We provide this function by creating an optimization wrapper scripts that generates and evaluates thousands of design variants. These designs’ results are used to approximate the energy/performance trade-offs of this space.

To create our FPU generator we used the Genesis2 compiler [3], [4]. Genesis2 uses Perl as a meta-language that extends SystemVerilog’s elaboration capabilities (maintaining the same hierarchical scopes), and it produces SystemVerilog-only instances as its output. Initially, this allowed us to use our existing SystemVerilog FPU design and then gradually extend its permutations. Additionally, since the language is an extension to SystemVerilog it allowed us to use existing ASIC methodology and flows (e.g. assertions, synthesis pragmas and verification primitives).

Generators also need to create validation collateral, since the user of the generator will not necessarily be an expert in that area. To create the needed test environment, we used IBM’s FPgen Test Suite, which is capable of generating high coverage tests for a variety of FP bit-widths and operations [8]. Our generator creates both the final Verilog and the information needed to configure the test generator for each instance. For example parameters like mantissa and exponent widths, IEEE rounding, and internal forwarding must be sent to the test program. Having this test infrastructure was essential for finding many subtle errors in our generator code.

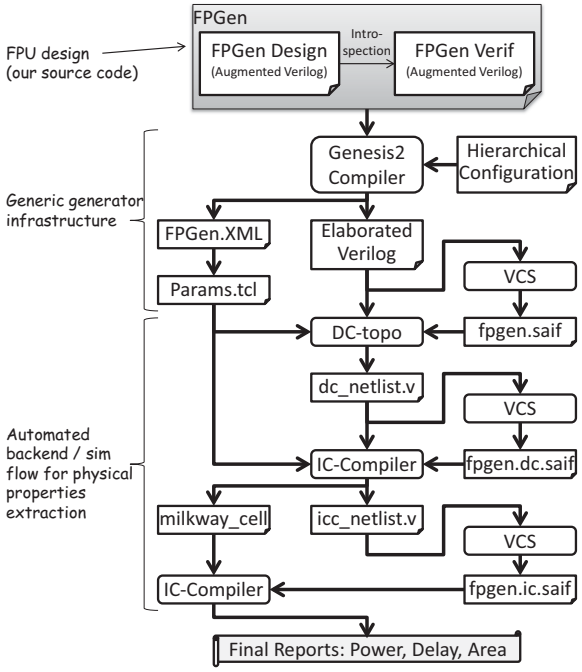


Fig. 1. FPGen design flow

Finally, for some of the experiments described below, we wanted to incorporate our knowledge of efficient *placement* for a multiplier tree. The multiplier tree accounts for a significant percentage of the FPU and is a regular structure whose construction is easily proceduralized. We saw that the quality of result could be improved by providing initial placement information to assist the placement algorithms. We achieved this by encoding the relative X and Y coordinates of individual cells in their instance names using Genesis2, with back end TCL regular expressions to extract this information after synthesis, and passing the resulting location information to the relative placement commands of the back-end tools.

Figure 1 depicts the generator flow block diagram. We used a standard Synopsys backend flow consisting of synthesis using Design Compiler Topographical Version F-2011.09-SP4 and IC Compiler Version F-2011.09-ICC-SP1. To maximize QoR and accuracy of measured power, we used Switching Activity Interchange Format (SAIF) which was extracted by simulation of the RTL and/or gate netlist using Synopsys VCS-MX. Target technology is TSMC 45nm.

III. Multiplication Architectures

To implement multiplication one needs to create hardware to generate the needed partial products (PPs) to be added and create the summation tree. Given the large amount of literature on both topics [9], we will not elaborate on the mathematical background; this section focuses only on implementation trade-offs.

1) Partial Products Generation

a) Unsigned group m encoding:

$$pp_i = y_{m(i+1)-1:mi} \cdot x \quad (1)$$

b) Modified Booth m encoding:

$$pp_i = (-2^m \cdot y_{m(i+1)-1} + y_{m(i+1)-1:mi} + y_{mi-1}) \cdot x \quad (2)$$

2) Partial Product Reduction

$$x \cdot y = \sum_{i=0}^{m-1} 2^{mi} pp_i \quad (3)$$

Fig. 2. Partial Product Generation and Parameterized Booth Encoding

A. Partial Product Generation & Booth Encoding

The simplest partial product generation method ANDs each multiplicand with the i^{th} bit from the multiplier to create n partial products. Each partial product PP_i is logically shifted by i positions to the left. The final result is the summation of these n vectors.

One can generate a factor of m fewer partial products by looking at groups of m bits instead of a single bit. A partial product PP_i is generated for every consecutive group of m multiplier bits ($y_{m(i+1)-1:mi}$) which can be any multiple of x between 0 and $(2^m - 1) \cdot x$. For example for $m = 2$, a partial product is selected from 0, x, $2x$ and $3x$. Generating the $2x$ multiple is easy by shifting x one bit to the left. However generating the $3x$ multiple is harder and adds to the multiplier delay.

Modified Booth algorithms address this issue by looking at overlapping groups of $m + 1$ muliplier bits ($y_{m(i+1)-1:mi-1}$) and using positive and negative multiples [10]. This means that there are fewer multiples to generate because the negative multiples are generated by inverting the positive multiples and adding one at the LSB position ($(-k) \cdot x = \overline{k \cdot x} + 1$). For example, with modified Booth 2, the PP's needed are only $-2x$, $-x$, 0, x and $2x$ multiples, all of which are easy to generate. Fig. 2 summarizes the Booth encoding algorithm. Table I compares different optimized implementations of Booth encoders that generate select signals, and Booth muxes that produce each bit in the partial product based on the select. This analysis emphasizes the tradeoff between the reduction of the number of Booth encoders and partial products, and the increased size and complexity of the Booth encoders and Booth muxes. As m grows, the muxes become bigger and slower, but the total area (and thus power) of the resulting multiplier tree goes down by n^2/m. The number of Booth encoders also goes down by n/m. To first order, the total area of the multiplier is composed of Booth muxes and CSA cells for summation, and is proportional to $\frac{n^2}{m} \cdot (1 + A(m))$ where $A(m)$ is the area of a Booth m mux normalized to the size of a CSA cell. For most n's modified Booth 3 minimizes the ratio $\frac{1+A(m)}{m}$ and as will be seen later is indeed the design of choice for low energy multipliers. Booth 4 is only marginally better but it doubles the number of wire tracks.

Our generator is capable of generating FPUs with any of the encoding options in Table I. Furthermore, several optimization have been incorporated to achieve the least delay and area. For example, in modified Booth 2, the inverted select signals are used so that the booth mux is only an XNOR gate and AOI gate as shown in Table I. The XNOR gate has been moved from being applied to the selected output to being applied to the input x to produce the signed x. This is advantageous because the $select\_x$ and $select\_2x$ signals arrive later than the sign S and the multiplicand x, and therefore this optimization balances the paths. This reduces the critical path of the Booth algorithm by 0.5 CSA gate delay without any additional area. Furthermore, $\overline{S \oplus x_j}$ is fed from the $j-1$ mux while $\overline{S \oplus x_{j+1}}$ is calculated in the j^{th} mux and fanned out to the $j+1$ mux. Furthermore the XNOR gate is implemented using an OAI gate which is smaller and faster ($\overline{S \oplus x_{j+1}} = \overline{S}x_{j+1} + S\overline{x_{j+1}}$). For modified Booth 3, which requires $0, \pm1, \pm2, \pm3, \pm4$ multiples, the ±3 multiples present a challenge because they cannot be generated using simple shift or inversion and are therefore called "hard" multiples. We further incorporated into our generator optimized 3-multiple hardware as described by Ruiz et al. [11]. Unfortunately, we are not aware of similar optimizations for ±5 and ±7 multiples required for the modified Booth 4, and relied on standard synthesis for that.

B. Partial Product Reduction

After the partial products are generated, the next step is to successively reduce them to only 2 partial products using reduction elements that produce fewer bits than they consume. For example a carry-save add consumes 3 bits and produces 2 bits and can reduce n partial products to 2 if applied $n-2$ times. We considered many design choices: the type of the reduction element, the interconnection of the reduction elements and the physical layout and optimization.

1) Reduction Element: Carry save adders (CSA) are the most prevalent element in multiplier design. CSAs are merely a special case of the general concept of $(2^r - 1, r)$ counters, where r-bit output is the count of the number of $2^r - 1$ inputs that are equal to 1 [12]. A CSA is customarily implemented as two successive XOR gates to produce "sum," and an AOI gate or NAND gates for "carry." Thus the sum calculation takes twice as long as the carry calculation and has inherent inefficiency due to an unbalanced XOR tree that reduces (only) three inputs in two logic levels (a full, balanced, XOR tree would have taken four inputs). Higher order parallel counters, such as optimized (7,3) counters, can be used to build a better balanced tree. A (7,3) takes seven inputs at once, and produces the sum bit in three XOR levels, and the carry bits in roughly four levels. As such (7,3) counters have twice the delay of a CSA but reduce the partial products count by a factor of $\frac{7}{3}$. Two levels of CSA's, in comparison, only reduce the partial

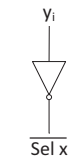
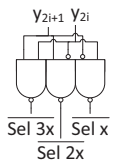
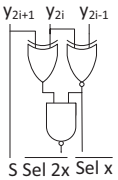
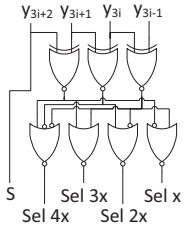
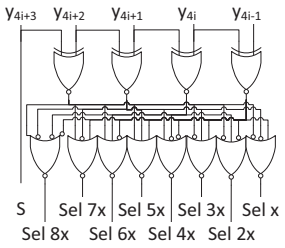
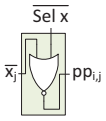
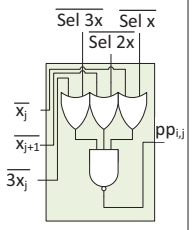
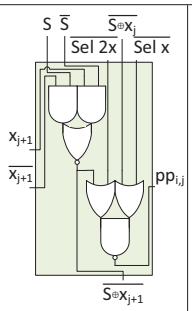
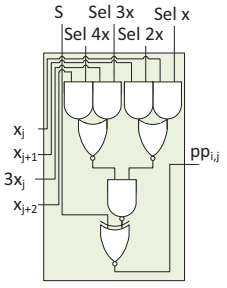
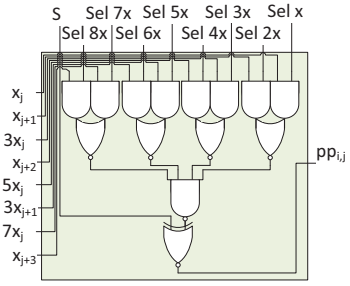
	Unsigned Group Encoding		Modified Booth		
	$m=1$	$m=2$	$m=2$	$m=3$	$m=4$
Encoder Function	$Selx_{i,k} = (k == y_{mi+m-1:mi})$		$Selx_{i,k} = (k == \pm(-2^m y_{mi+m-1} + y_{mi+m-1:mi} + y_{mi-1}))$ $S_i = y_{mi+m-1}$		
i^{th} Encoder					
Encoder Area	0.063	0.375	0.875	2	3.5
Encoder Delay	0.125	0.25	0.75	0.75	1
# of Encoders	n	$\lceil \frac{n}{2} \rceil$	$\lceil \frac{n+1}{2} \rceil$	$\lceil \frac{n+1}{3} \rceil$	$\lceil \frac{n+1}{4} \rceil$
Mux Function	$pp_{i,j} = \sum_{k=0}^{2^m-1} Selx_{i,k} k x_j$		$pp_{i,j} = S_i \oplus \sum_{k=0}^{2^{m-1}} Selx_{i,k} k x_j$		
j^{th} Cell of i^{th} Partial Product					
Mux Area	0.125	0.375	0.5	1	1.625
Mux Delay	0.25	0.5	0.5	1	1
# of Muxes	n^2	$(n+2)\lceil \frac{n}{2} \rceil$	$(n+1)\lceil \frac{n+1}{2} \rceil$	$(n+2)\lceil \frac{n+1}{3} \rceil$	$(n+3)\lceil \frac{n+1}{4} \rceil$
Encoders Area	$0.063n$	$0.375\lceil \frac{n}{2} \rceil$	$0.875\lceil \frac{n+1}{2} \rceil$	$2\lceil \frac{n+1}{3} \rceil$	$3.5\lceil \frac{n+1}{4} \rceil$
Hard Multiples Area		n		n	$3n$
Muxes Area	$0.125n^2$	$0.375(n+2)\lceil \frac{n}{2} \rceil$	$0.5(n+1)\lceil \frac{n+1}{2} \rceil$	$(n+2)\lceil \frac{n+1}{3} \rceil$	$1.625(n+3)\lceil \frac{n+1}{4} \rceil$
Tree Area	$n(n-2)$	$(n+2)\lceil \frac{n-4}{2} \rceil$	$(n+1)\lceil \frac{n-3}{2} \rceil$	$(n+2)\lceil \frac{n-5}{3} \rceil$	$(n+3)\lceil \frac{n-7}{4} \rceil$
Total Area	$1.125n^2 - 1.938n$	$0.688n^2 + 0.563n - 4$	$0.75n^2 - 0.75n - 0.813$	$0.667n^2 + 1.667n - 2$	$0.656n^2 + 4.5n - 3.156$
Booth Delay	0.37	0.5	1	1	1
Hard Multiples Delay		$1 + 0.5log_2 n$		$1 + 0.5log_2 n$	$1 + 0.5log_2 n$
Tree Delay	$log_{1.5}\frac{n}{2}$	$log_{1.5}\frac{n}{4}$	$log_{1.5}\frac{n}{4}$	$log_{1.5}\frac{n}{6}$	$log_{1.5}\frac{n}{8}$
Total Delay	$log_{1.5}n - 1.34$	$1.29log_{1.5}n - 1.92$	$log_{1.5}n - 2.42$	$1.29log_{1.5}n - 2.42$	$1.29log_{1.5}n - 2.13$

TABLE I

COMPARISON OF OPTIMIZED IMPLEMENTATIONS OF UNSIGNED GROUP ENCODING AND MODIFIED BOOTH ENCODING SCHEMES. ALL AREAS AND DELAYS ARE REPORTED IN COMPARISON TO THE AREA AND DELAY OF ONE CSA CELL. MODIFIED BOOTH 2 HAS THE LOWEST DELAY FOR MOST VALUES OF N WHILE MODIFIED BOOTH 3 PROVIDES ĩ0% IMPROVEMENT IN AREA AND THUS ENERGY FOR LARGE VALUES OF N. UNSIGNED GROUP ENCODING 2 AND MODIFIED BOOTH 4 GIVE SIMILAR AREA SAVINGS BUT HAVE WORSE DELAYS THAN MODIFIED BOOTH 3.

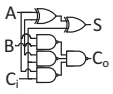
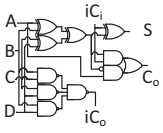
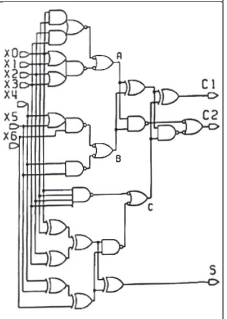
	3:2 Counter	4:2 Compressor	7:3 Counter [12]
Circuit			
Element Delay	1	1.5	2
Tree Delay	$0.75\log_{3/2} n = \log_{1.72} n$	$1.5\log_2 n = \log_{1.59} n$	$2\log_{7/3} n = \log_{1.53} n$
Element Area	1	2	4
#Elements	$n-2$	$\frac{n-2}{2}$	$\frac{n-3}{4}$
Tree Area	$n-2$	$n-2$	$n-3$

TABLE II

COMPARISON OF REDUCTION ELEMENTS. ALL DELAYS AND AREAS ARE RELATIVE TO THE DELAY AND AREA OF A CARRY SAVE ADDER. 3:2 COUNTERS ARE ASSUMED TO BE CONNECTED OPTIMALLY TO HAVE AN EFFECTIVE 1.5 CSA DELAYS FOR EVERY 2 LEVELS OF REDUCTIONS.

products count by $\frac{9}{4}$, seemingly a slight advantage for (7,3) counters. However as we show in section III-B3, carefully interconnecting the CSA's can be used to balance the delays of the XOR trees and reduce the delay of every two levels by 25%, an optimization that is superior to big parallel counters and even to 4:2 compressors. As such CSA's are the preferred reduction elements in our generator. Table II summarizes the properties of these reduction elements.

2) Reduction Topology: Having decided on CSA's as our preferred reduction element, the next design decision is how to interconnect these CSA's. There are two classes of topology: regular, where CSAs are connected in a replicated pattern; and irregular, where the connection is tweaked for minimal delay. Irregular topologies are faster, but harder to code and harder to lay out.

The most well-known topology, the *Wallace Tree,* is irregular but it achieves $\log_{3/2}(N)$ delay, the minimum possible delay using CSAs [13]. It also requires a large number of tracks, which is greater than $\log(N)$.

Regular topologies can be subdivided into array and tree topologies. A simple array connects CSAs serially, with minimum tracks but maximum delay. Higher order arrays can balance delay and the number of tracks. For example, a double array groups the PPs into two simple arrays which are added in parallel, with the results combined at the end. The resulting delay, about $N/2$ CSA levels, is almost half that of a simple array.

Tree topologies are 3D structures that have to be flattened. Examples of regular trees include binary, Zuras-McAllister (ZM) [14] and Overturned-Staircase (OS) [15] trees. A binary

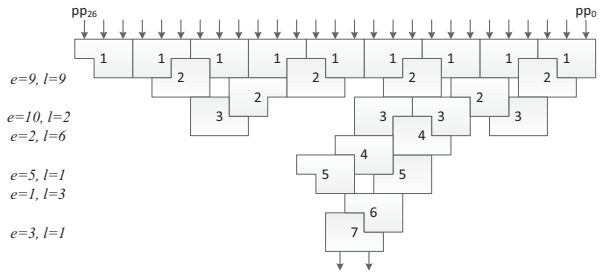


Fig. 3. Double Precision Wallace Booth 2 Multiplier Tree. Balancing the interconnection of sum and carries lets 7 levels of CSA take only 5.5 CSA delays. Boxes truncated from the bottom represent a CSA cell with early carry output while boxes truncated from top represent cells with late carry input. 'l' and 'e' are the numbers of late and early signals at each level of the tree.

tree divides the PPs in groups of 4 which are reduced to 2 using 4:2 compressors; the outputs are then reduced again and so on. This creates a delay of $\log_2(N/2)$ compressors. A problem with the binary tree is that, when flattened, it produces long wires. The ZM or Balanced Delay tree solves this problem by sacrificing more logic levels for shorter wires and fewer tracks. The OS tree recursively builds a tree of height k from a tree of height $k-1$ and a serial branch of height $k-2$ connected using two CSAs. Here k is the maximum number of CSAs connected serially. Our generator currently supports four topologies: array, Wallace, ZM and OS trees.

3) Delay Balancing: As mentioned earlier, the CSA's sum output is slow, taking two successive XOR2 gate delays while the carry calculation is faster and can be done using two levels of NAND2 and NAND3 gates. As a result, the total delay from any input to carry out is only 0.5 CSA delay. The delay from carry-in to any output is also 0.5 CSA delay.

Put differently, if all inputs arrive at the same time, the sum output takes 1 CSA delay and the carry takes 0.5 CSA delay. Several designs have optimized the interconnection of the CSA's to balance the delays and shorten the total delay [16], [17]. To balance the paths in the tree, if early signals from level i of the tree are fed to the slow inputs (i.e., not the carry-in input) of level $i+1$, and late signals are fed to the carry-in input of level $i+1$ (0.5 CSA delay later), then both the output sum and carry-out of level $i+1$ are produced after 0.5 CSA delay, halving the effective delay of level $i+1$. That means we can make every two consecutive stages take 1.5 CSA delays instead of two CSA delays! Figure 3 shows how this optimization can be used to implement the seven levels of CSAs required to reduce 27 partial products produced by Booth 2 for a double precision mantissa. After the first level, nine carry out outputs are early while nine sum results arrive after a full CSA delay. Taking 8 of these early signals with 4 other late ones, we can build 4 CSA's in the next level whose output will have 1.5 CSA delay instead of 2. As this optimization is applied repeatedly the delay paths among CSA's are balanced, reducing the total delay of the calculation.

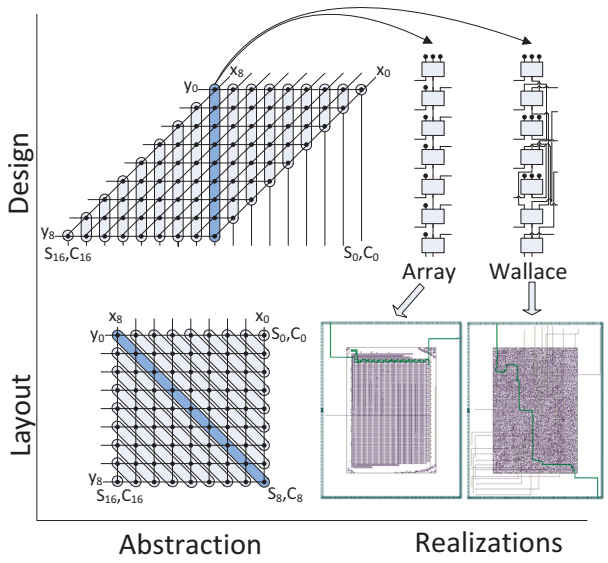


Fig. 4. Multiplier tree design flow

C. Tradeoffs in Multiplier Design

Putting the Booth type, the tree type, and the multiplier bit width together provides an interesting design tradeoff between power, area and performance. We have already seen that Booth 3 minimizes area, especially for larger bit widths, but it also adds latency. Because of the object oriented nature of generators, with constructors for each module, we are able to support and explore any combination of the three.

Not surprisingly we found that when delay is critical, a Wallace Booth 2 multiplier performs best. But a close look at the placed cells showed that there was a disconnect between how we imagined the tree structure and how the tools placed it. That meant that for a true comparison, we needed a way to incorporate layout information into the generator and down to the generated Verilog and the gate netlist. To encode the designer intent as layout hints, we planted *X* and *Y* coordinates into the name of each cell in the tree constructor. Our back-end scripts used that information to create a better "starting point" from which the automatic placer could start and further optimize. This abstraction enables a designer to experiment with new implementations and layout strategies by changing the (meta-language) calculation of *X* and *Y*.

Considering the conceptual parallelogram at the top left of Figure 4, the designer intent is for each bit column to be placed diagonally to the right, to fit in a square layout (bottom left of Figure 4). Additionally we want the Booth mux cells interleaved with the multiplier reduction tree, placed at the intersection of multiplicand and select signals. This placement results in the critical path being a clean run diagonally through the multiplier area (bottom right of Figure 4). In Figure 5 dashed vs. solid lines show the advantage of this semi-custom placement. In particular it points to designs that otherwise looked inefficient, like the OS1 tree in quad precision multipliers. OS1 is a regular structure, designed to minimize wire

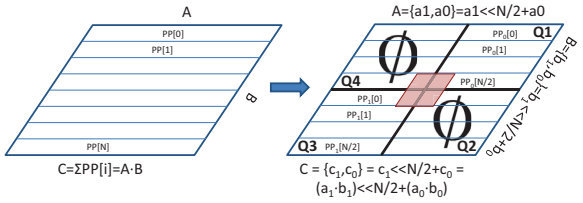


Fig. 6. Using double-precision multiplier hardware for two parallel single precision multiplications. Quadrants 2 and 4 must be zeroed, and carry chains must be cut. Highlighted center parallelogram is area of contention.

length and tracks which dominate higher n multipliers, thus maintaining/propagating the structural intent is critical.

IV. Extending The Multiplier Generator To Multi-precision

Programming languages offer a choice between *float* and *double* data types. For programmers, the trade-off is clear: floats are single precision, use only 32 bits in memory, and are sufficient for most day-to-day calculations, while doubles occupy twice the memory space, but provide more than twice the precision with roughly 2^{1821} times the dynamic range[1]. For most scientific computations, doubles are the default choice. Hardware manufacturers face a different trade-off. Adding both single- and double-precision units may not be the best utilization of silicon area. However, a single-precision unit emulating double-precision incurs a performance penalty and reduced accuracy [18]. Silicon area could be saved by leveraging a single double precision unit executing one double precision or two single precision operations per cycle.

Given the two single precision multiplications $c_1 = a_1 \cdot b_1$ and $c_0 = a_0 \cdot b_0$, it is easy to see that if we define $A = \{a_1, a_0\} = (a_1 << N/2) + a_0$ and $B = \{b_1, b_0\} = (b_1 << N/2) + b_0$ then $C = A \cdot B = \{a_1, a_0\} \cdot \{b_1, b_0\} = (a_1 \cdot b_1) << N + (a_1 \cdot b_0) << N/2 + (a_0 \cdot b_1) << N/2 + (a_0 \cdot b_0)$.

Therefore, each double precision multiplication already embodies four sets of single precision multiplications. All we need to do is to modify the existing hardware so that we remove the contribution of $a_1 \cdot b_0$ and $a_0 \cdot b_1$ and our double precision multiplier result becomes $C = \{c_1, c_0\}$.

While conceptually this is simple, Booth coding makes this task more complex. Our first task is straight-forward: force the PP in the $a_1 \cdot b_0$ and $a_0 \cdot b_1$ quadrants to zero by asserting the zero select signal in the relevant Booth encoder cells.

Second, we guarantee that no carry spills over from one operation to the other by isolating the single precision intermediates in quadrants 1 and 3. In the case of no Booth, the increase in mantissa width alone is sufficient to do that. For example, when going from half to single to double to quad the width of the mantissa grows as 11, 24, 53, 113 respectively.

[1]Here, dynamic range DR is the quotient of the highest representable number by the lowest non-zero number. Therefore $\frac{DR_{DP}}{DR_{SP}} = \frac{Max_{DP}/Min_{DP}}{Max_{SP}/Min_{SP}} = \frac{(1+(1-2^{52}))\times 2^{1023}}{(1+(1-2^{23}))\times 2^{127}} / \frac{2^{-1022-52}}{2^{-126-23}} \simeq 2^{1821}$

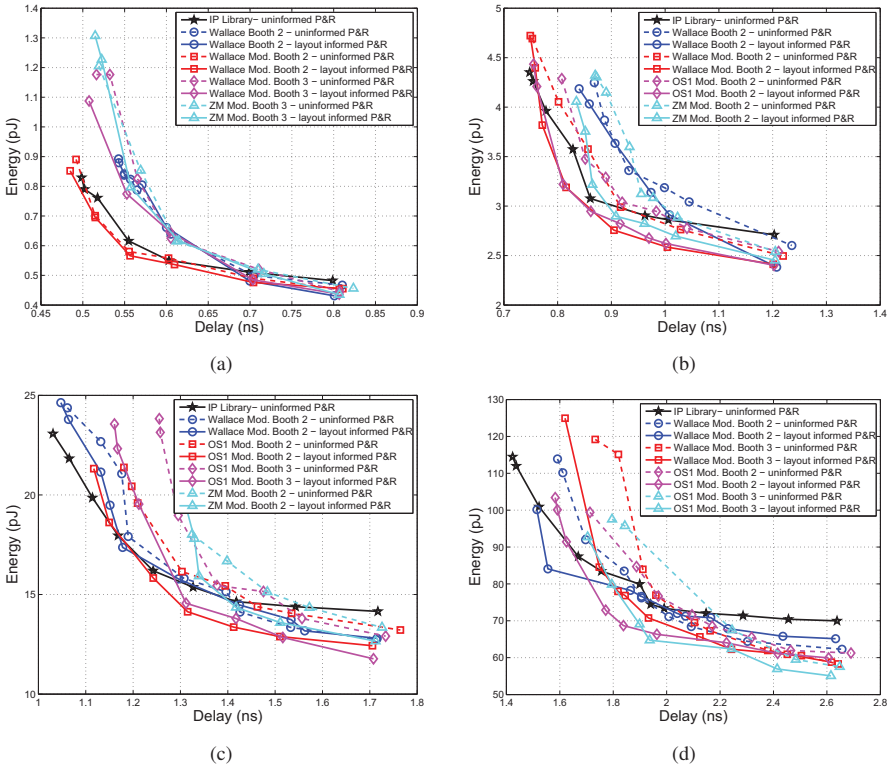


Fig. 5. Energy-Delay tradeoffs for floating point multiplier trees. As expected, Wallace Booth 2 multipliers mostly dominates for low delay. At high bit-widths however, wires become critical and OS1 structures perform better. Booth 3 designs match or exceed Booth 2 designs when power is critical. Numbers are measured at 0.9V using SVT cells. (a) Half precision (b) Single precision (c) Double precision (d) Quad precision

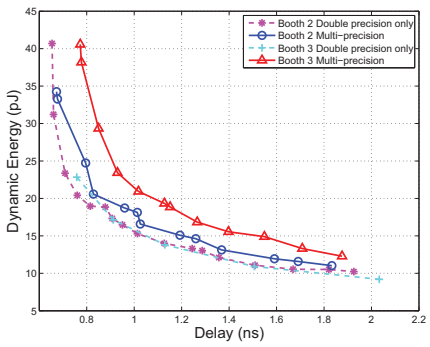
However Booth-2 and Booth-3 require a little more modification because the partial product widths increase to accommodate the sign bits. In Booth 2 for example, single precision partial products have $24+3\cdot2=30$ bits vs. $53+3\cdot2=59$ bits double precision partial products. However, because there is a logical shift of two bits between partial products, no further modification was needed to isolate the two single precision operations.

For Booth-3 the single precision partial products are up to 33 bits while the double precision partial products are 62 bits. Even after the three bits of logical shift, there is no isolation for the single precision operations. In this case either the carry can be explicitly killed, or the entire multiplier can be widened. If the multiplier is widened, then the double precision multiplicand A is padded with sign extenders to the left (i.e. $\{0..0, A\}$), and in the lower precision calculation, a_0 is padded with zeros (i.e., $\{a_1, 0..0, a_0\}$). We found that widening the partial products by a couple of bits has only modest impact on power and area, and no noticeable impact on performance as the tree depth remains the same.

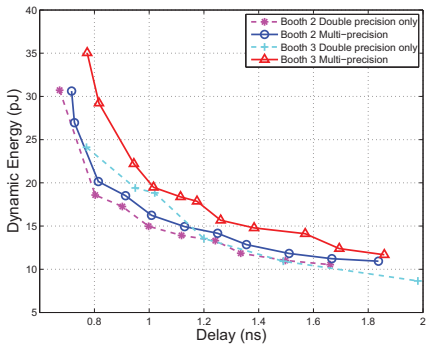
When adding features to a generator, the greatest challenge is maintaining their interactions with other features. In our designs we enable forwarding of unrounded results in the fused- or cascaded-multiply-accumulate unit as described in Section V. That means that in forwarding mode our multiplier actually implements $(A + inc_A) \cdot (B + inc_B)$ by adding one additional partial product, similar to [19]. Thus a Booth 3 DP multiplier with forwarding has 62 bit width and 19 partial products, while each Booth 3 SP requires 33 bits width and 10 partial products. The key problem here is that the original tree does not have enough PPs to accommodate the two lower precision trees combined. One of the partial products must be shared, breaking the nice quadrant separation described above, or a PP must be added.

An additional partial product has only modest impact on power and area (around 5% in this case), but moving from 19 to 20 partial products means that some tree structures may become one level higher. A Wallace tree has only six levels of summation for up to 19 PPs, but seven levels for 20 PPs. This is an 18% performance impact. Therefore a better solution for Booth 3 multipliers implemented with a Wallace tree is additional zero padding (i.e. using wider PPs). OS trees use seven levels to sum either 19 or 20 PPs, so additional zero padding (and complexity) is a waste for OS trees.

Figure 7 compares 1) power and delay for multi-precision multipliers using Wallace trees for Booth 2 and 3 versus 2) a double-precision only multiplier of the same Booth and tree type. One can easily see that the overhead of a multi-precision Booth 2 multiplier is modest, on the order of 5%-10%, while the overhead of a Booth 3 multi-precision multiplier is 10%-20%. For OS trees, the overhead is modest for both Booths.



(a)



(b)

Fig. 7. Energy-Delay tradeoffs for floating point multi-precision multipliers. Overhead for supporting either one double precision or two single precision (per cycle) is at about 10%-20% for the most part. For Wallace Booth 3 multiplier with forwarding, the overhead is higher due to an extra partial product that makes the summation tree one CSA level taller. (a) Wallace tree Booth 2&3. (b) OS1 tree Booth 2&3.

V. FMA Generator

Using the smaller generators described above, we turn now to putting it all together for the FMA unit in the heart of our FPU generator, and then exploring the design space. At this level, our generator is able to produce either a fused mul-add or a cascade architecture that allows early issue of accumulation-dependent instructions [20] with possibilities for more aggressive clock gating [21]. We support arbitrary pipeline depth using a combination of automatic retiming flows along with "designer guided" schemes as explained in Section V-A. We further support early forwarding of unrounded results as first implemented in the Power6 FPU [19]. To reduce dynamic power, all sub-units are clock gated based on instruction type for NOP, FADD, FMUL, FMADD. For example in the case of FADD, the clock to the multiplier is gated off. And for the cascade design, only one of the "close" or "far" path of the addition logic is clocked, to further save energy.

A. Pipelining of the FMA

For most datapath designs, automatic register retiming works well for pipelining the design. There, retiming is done by inserting a group of consecutive pipeline stages at a certain bisection of the design and allowing the synthesis tool to move

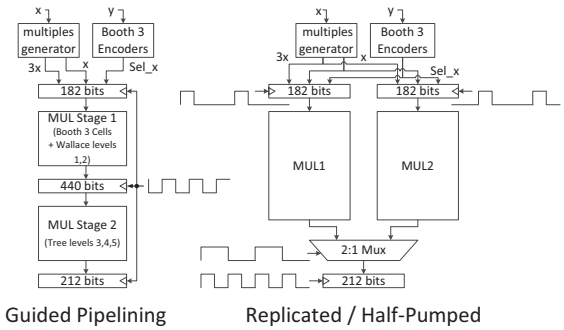
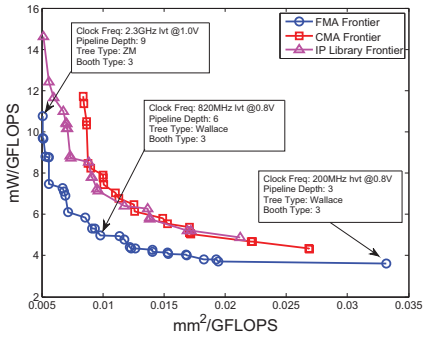


Fig. 8. Example "guided" and "replicated" implementations of pipelining. Numbers based on a double precision multiplier implemented using a modified Booth 3 and Wallace tree.

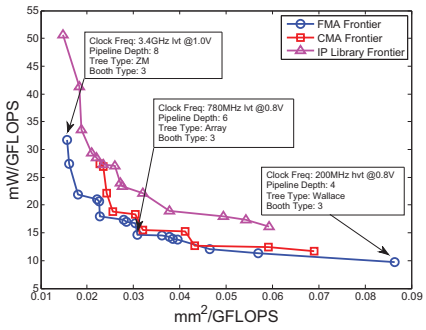
these pipeline stages across the combinational logic. For most parts of the FMA datapath this approach should provide good, better-than-human results as the retiming tool converts the circuit into a graph optimization problem and finds an optimal solution. There are at least two cases, however, where this approach does not work well. The first is when an architectural change can do better than just moving gates around. The second is when control logic is involved, because it is hard for the tool to guarantee that the timed signatures of signals maintain the intended state-action synchronization.

Architectural changes to facilitate retiming have an advantage when the number of signals in the path grows and shrinks substantially, because that path can be replicated rather than registered. In the FMA unit, the number of signals inside the multiplier scales quadratically. For example for a double-precision-mantissa modified Booth 2 multiplier, while the input signals are only 2×53 bits, the number of signals after the Booth encoding is 27×55 and decreases by a factor of 1.5 after every level of CSA until the output is 2×106 bits. As such, automatic retiming can result in an unacceptable overhead both in terms of energy and area. Therefore we devised three solutions to this problem

- Do nothing / automatic retiming: Place all pipe stages at the output of the multiplier and let the retiming tool optimize.
- Smart / guided retiming for low frequency designs: the multiplier part is more aggressively sized to fit within one clock cycle. One stage of pipelining is inserted immediately after the Booth select signals and the rest of the pipeline stages are put at the multiplier output. Retiming still pushes registers from the output inward, but that is the narrowest part of the tree thereby minimizing the overhead.
- Replication / half pumping for high frequency designs: The entire multiplier is replicated such that each replica has a multicycle path as shown in Figure 8. This minimizes the number of registers and allows for relaxed sizing of the multipliers at the cost of increased area.



(a)



(b)

Fig. 9. Throughput tradeoffs for FMA and CMA: (a) single precision (b) double precision.

B. Throughput Efficient Designs

Heavy throughput machines such as graphical processors provide a key use case for floating point units. Interestingly, Galal et. al. have shown that the key criteria for throughput oriented FMAs is the energy required per operation and the area required per operation [22], because performance is dominated more by the feasible number of FPUs, and not so much by the latency of a given FPU. Figure 9 shows how our fused and cascade architectures perform under that criteria (and compared to a state of the art IP library). Clearly fused architectures perform better. The reason is that cascade architectures trade additional logic for latency savings as we will see next. Moreover, the graphs shown here are the Pareto curves from thousands of runs exploring all the knobs discussed thus far. A close look at the designs that "made the cut" to the Pareto curve, shows that they were mostly Booth 3 for SP and almost solely Booth 3 for DP. This is because Booth 3 minimizes the area as discussed earlier, especially for large N. The tree structure, on the other hand, matters less as it is more prominent in determining the delay, not the area or energy. All SP Pareto designs had 3-8 pipe stages and DP designs had 4-8 pipe stages.

C. Latency Efficient Designs

Latency machines try to maximize performance under a certain power criteria. Area is less important since the FPU is not likely to be replicated many times. As we see here,

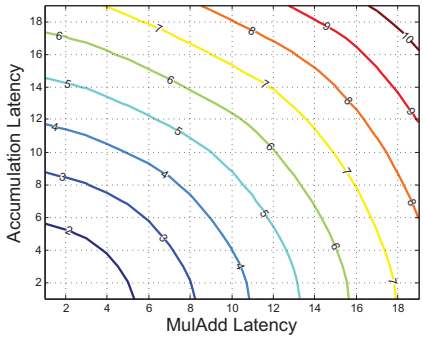
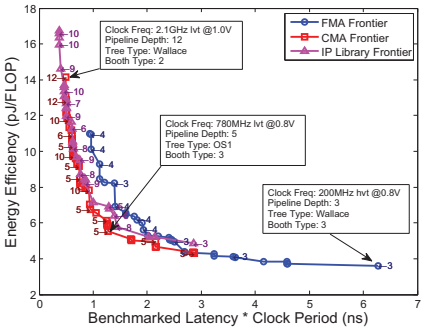


Fig. 10. Effective Latency of benchmarked FP application as function of the accumulation and mul-add latencies. Cascaded designs have shorter accumulation latency, around half the mul-add latency, unlike the equal latencies of traditional FMA designs.

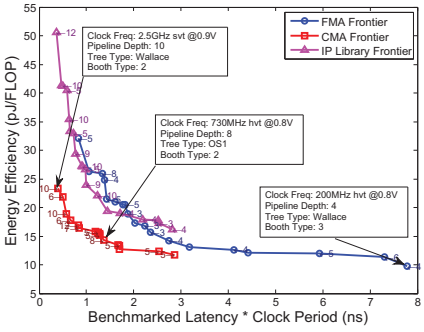
and as predicted by [21] the cascade design performs better because the latency of a dependent instruction is smaller than in the fused architecture case, and because the latency of add (without mul) instructions is also reduced.

In light of the asymmetric latencies for accumulation/multiplication dependent operations, to make fair comparisons of cascade and fused designs, or even designs of different pipe depths, we had to understand how each latency impacted the CPU performance. We examined execution traces of SPEC 2000 floating point benchmarks compiled for PowerPC and running on an M5 in-order processor simulator [23]. We recorded the distance of FMADD, FADD and FMUL instructions from the instruction they were dependent upon. The information is recorded in a matrix of histograms of dependencies through accumulation or mul-add. For example if an FMADD instruction is dependent on multiplicands produced 3 and 5 cycles earlier and an addend produced 2 cycle earlier, a 1 is added to the (2,3) position of the histogram. Meanwhile, an FADD instruction whose addends are produced 4 and 7 cycles earlier is recorded at (4,0). Finally this information is averaged over different benchmarks. We then created a latency penalty function that uses this matrix to calculate how many cycles on average each design will stall for accumulation latency x and mul-add latency y as illustrated in Figure 10. We note however that the numbers here depend on compiler optimizations and may be skewed a little because the PowerPC compiler assumes six cycle mul-add latency.

Once the penalty for add, mul and mul-add is known, we calculate a weighted average latency based on the instruction mix, and multiply by the achieved clock period to find a single pseudo delay number. Figure 11 shows the Pareto curves for cascade and fused designs (and compared to a state-of-the-art IP library). We clearly see here how the cascade designs dominate. Moreover, like before, these represent thousands of runs among various configurations, only this time the designs that made it to the Pareto curve are mostly Wallace trees (with a few OS1 entries), and mostly Booth 2. Pipe depth varied from 5 to 12 and even 16. About half the designs on the efficient frontier had replicated multi-cycle multipliers.



(a)



(b)

Fig. 11. Latency tradeoffs for FMA and CMA: (a) single precision (b) double precision.

VI. Conclusion

We presented a new floating point generator that allowed us to fairly compare different proposed floating point implementations. Our results show the importance of using optimized building blocks for researching new FPU improvements. For example once CSAs are wired to balance delays, it becomes clear that higher radix counters (7-3) do not make faster combining networks. Using this system we can generate optimal FPU implementations for different applications, and interestingly there is not one best architecture. While Booth 2 Wallace was generally the highest performance solution, we were surprised to find that for throughput machines, where FLOPS/area is critical, Booth 3 is the best solution, since it yields smaller multiplier arrays and Wallace was often not the top combining network. The result is not only a better understanding of the trade-offs in FPU design, but also a software system that can generate optimized FPU implementations. In addition, during this project we added both new requirements (multi-precision multiplier), and new FP efficiency tricks, which indicates this framework is extendable as well. Using a generator approach has created a powerful tool for FPU designers and users, and we hope this technique can be extended to other areas as well. We welcome other researchers to download and extend our generator, "FPGen," suggest new optimizations, and test their real impact. FPGen is available at http://www-vlsi.stanford.edu/fpgen .

Acknowledgement

This material is based upon work supported by the Defense Advanced Research Projects Agency under Contract No. HR0011-11-C-0007. Any opinions, findings and conclusion or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Defense Advanced Research Projects Agency.

References

[1] B. Randell, "From analytical engine to electronic digital computer: The contributions of Ludgate, Torres, and Bush," *Annals of the History of Computing*, vol. 4, no. 4, pp. 327 –341, Oct.-Dec. 1982.

[2] O. Shacham, O. Azizi, M. Wachs, W.Qadeer, Z. Asgar, K. Kelley, J. Stevenson, A. Solomatnikov, A. Firoozshahian, B. Lee, S. Richardson, and M. Horowitz, "Why design must change: Rethinking digital design," *IEEE Micro*, vol. 30, no. 6, pp. 9 –24, nov.-dec. 2010.

[3] O. Shacham, S. Galal, S. Sankaranarayanan, M. Wachs, J. Brunhaver, A. Vassiliev, M. Horowitz, A. Danowitz, W. Qadeer, and S. Richardson, "Avoiding game over: Bringing design to the next level," in *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, june 2012.

[4] "Creating Chip Generators Using Genesis2," Stanford, http://genesis2.stanford.edu/.

[5] J. Bachrach, H. Vo, B. Richards, K. Asanovic, and J. Wawrzynek, "Chisel: Constructing hardware in a Scala embedded language," in *Proceedings of the 49th Design Automation Conference (DAC)*, 2012.

[6] F. de Dinechin and B. Pasca, "Designing custom arithmetic data paths with flopoco," 2011.

[7] "Synopsys DesignWare Library - Datapath and Building Block IP," Synopsys, http://www.synopsys.com/dw/buildingblock.php.

[8] M. Aharoni, S. Asaf, L. Fournier, A. Koifman, and R. Nagel, "Fpgen - a test generation framework for datapath floating-point verification," in *In Proc. IEEE International High Level Design Validation and Test Workshop 2003 (HLDVT03)*, 2003.

[9] M. J. Flynn and S. F. Oberman, *Advanced Computer Arithmetic Design*. Wiley-Interscience, 2001.

[10] O. Macsorley, "High-speed arithmetic in binary computers," *Proceedings of the IRE*, vol. 49, no. 1, pp. 67 –91, jan. 1961.

[11] G. A. Ruiz and M. Granda, "Efficient implementation of 3X for radix-8 encoding," *Microelectronics Journal*, 2008. [Online]. Available: http://dx.doi.org/10.1016/j.mejo.2007.10.006

[12] J. Swartzlander, E.E., "A review of large parallel counter designs," in *VLSI, 2004. Proceedings. IEEE Computer society Annual Symposium on*, feb. 2004, pp. 89 – 98.

[13] Wallace, "A suggestion for a fast multiplier," *IEEETC: IEEE Transactions on Electronic Computers*, vol. 13, pp. 13–17, 1964.

[14] D. Zuras and W. H. McAllister, "Balanced delays trees and combinatorial division in VLSI," *IEEE journal of solid-state circuits*, vol. SC-21, pp. 814–819, 1986.

[15] Z.-J. A. Mou and F. Jutand, ""overturned-stairs" adder trees and multiplier design," *IEEE Trans. Computers*, 1992.

[16] V. Oklobdzija, D. Villeger, and S. Liu, "A method for speed optimized partial product reduction and generation of fast parallel multipliers using an algorithmic approach," *Computers, IEEE Transactions on*, vol. 45, no. 3, pp. 294 –306, mar 1996.

[17] E. Schwarz, I. Averill, R.M., and L. Sigal, "A radix-8 cmos s/390 multiplier," in *Computer Arithmetic, 1997. Proceedings., 13th IEEE Symposium on*, jul 1997, pp. 2 –9.

[18] D. Goddeke, R. Strzodka, and S. Turek, "Performance and accuracy of hardware-oriented native-, emulated- and mixed-precision solvers in FEM simulations," in *International Journal of Parallel, Emergent and Distributed Systems*, 2007.

[19] S. D. Trong, M. Schmookler, E. Schwarz, and M. Kroener, "P6 binary floating-point unit," in *Computer Arithmetic, 2007. ARITH '07. 18th IEEE Symposium on*, june 2007, pp. 77 –86.

[20] D. Lutz, "Fused multiply-add microarchitecture comprising separate early-normalizing multiply and add pipelines," in *Computer Arithmetic (ARITH), 2011 20th IEEE Symposium on*, july 2011, pp. 123 –128.

[21] S. Galal and M. Horowitz, "Latency sensitive fma design," in *Computer Arithmetic (ARITH), 2011 20th IEEE Symposium on*, july 2011.

[22] ——, "Energy-efficient floating-point unit design," *Computers, IEEE Transactions on*, vol. 60, no. 7, pp. 913 –922, july 2011.

[23] N. Binkert, R. Dreslinski, L. Hsu, K. Lim, A. Saidi, and S. Reinhardt, "The M5 simulator: Modeling networked systems," *IEEE Micro*, vol. 26, no. 4, pp. 52–60, 2006.