Hardware Accelerator for Exact Dot Product

David Biancolin and Jack Koenig
ASPIRE Laboratory
University of California, Berkeley

In this paper we present a coprocessor capable of computing a dot product exactly by use of a "complete register" (CR) that encodes a fixed point representation of the complete IEEE754 double precision space. We explore the design space of the coprocessor by running simulations on large numbers of distinct configurations. Since only the accumulation register is represented exactly, we demonstrate that EDP is realizable in silicon , requiring additional 11% over Rocket's area. In addition, the accelerator showed speedups of 3-6x over a conventional dot product and matrix multiplication while providing both exactness and reproducibility.

1 Introduction

Floating point computation is essential to the fields of scientific computing and engineering. Despite near universal use, most people are unaware that floating point is imprecise by its very nature. Where arithmetic operations like addition and multiplication exhibit the communicative and associative properties, intermediate rounding means the corresponding floating point operations do not. This poses a significant problem in safety critical areas like civil and nuclear engineering where human lives may depend on accurate and reproducible calculations.

Fortunately, there are solutions to this problem. The GNU Multiple Precision Floating-Point Reliably (MPFR) library [1] is a software solution that provides arbitrary precision floating point arithmetic. Unfortunately, software solutions like MPFR are 2-3 orders of magnitude slower than performing the equivalent operations in hardware. In applications where precision and performance are both first-order requirements, scientists and engineers must rely on algorithms with known, but non-zero error.

An alternative solution to this problem is to support complete (ie. No loss of precision) floating point arithmetic in hardware. Unfortunately, the cost of such a proposal is huge: IEEE double precision floating point has an exponent range from -1022 to 1023, which means more than 2045 bits are required to represent a single double exactly! In addition, the hardware cost of supporting every floating point operation on so many bits is enormous.

In this paper, we propose a compromise: a hardware accelerator for dot product without loss of precision. We amortize the cost of the hardware over long sequences of floating point operations. Given the ubiquity of dot product in mathematics, we believe providing this particular operation at high speed with no loss of precision could greatly benefit scientific computation and engineering.

2 Background

The idea of an exact dot product goes back to the days of mechanical calculators. In addition to the four elementary operations (addition, subtraction, multiplication, and division), calculators often had a fifth operation: "running total." The result register was much wider than the input registers, allowing for accumulation without loss of precision. This is very similar functionality to our proposed exact dot product accelerator.

Hardware acceleration of exact dot product has its own history as well. Ulrich Kulisch has been making the case for such acceleration for over two decades and was instrumental in the creation of the XPA 3233 co-processor in the early 1990s [2]. The XPA 3233 was made of 207,000 transistors and was connected to the PC through a PCI-bus. The latency of the PCI-bus greatly limited the usefulness of the XPA 3233.

Today, CPUs are comprised of billions of transistors, but dark silicon prevents all of the transistors from being powered at the same time. On-die hardware accelerators are a popular approach to improving performance despite this problem [3]. Our proposed accelerator fits right into this category. Transistors are cheap but calculations are expensive; we propose dedicating some of the excess area to accelerating a key operation while providing both exactness and reproducibility.

The highly parameterizable hardware construction language CHISEL [4] and the open-source RISC-V ISA and RoCC co-processor interface [5] provide us with the ability to explore a large space of realistic design points for a potential accelerator.

2.1 Principal of Operation

The basis of our proposed Exact Dot Product Accelerator (EDPA) is an exact representation of entire space produced by the product of two floating point numbers. For the product of two arbitrary floating points with the representation $(-1)^s * m \times 2^e$, $2*(e_{bits} + m_{bits})$ bits are required to represent the product exactly. To prevent overflow over a long accumulation, an additional *k* bits are added.

For IEEE Double Precision:

$m_{bits} = 53$, $e_{bits} = 2047$, $k = 92$: $CR_{bits} = 4288$ (67 words)

For IEEE Single Precision:

$m_{bits} = 24$, $e_{bits} = 256$, $k = 86$: $CR_{bits} = 640$ (10 words)

Unless otherwise specified, all further examples assume double precision.

To manage this size, additions to the CR are done in place. Thus, to add $A*B, A,B \in FP$ to the CR, all of the studied implementations do the following (illustrated in Fig. 1).

1. C = A * B is computed without truncation. $m_{bits,prod} = 2*m_{bits,op} = 106$, $e = e + 1 = 12$
2. m_{prod} is shifted into alignment with the word boundaries of the CR, based on the lower $\log_2(W_{CRW})$ bits of e_{prod}, where W_{CRW} is the width of a CR segment.
3. The remaining bits of e_{prod} address the particular words of the CR the product to which the $m_{prod,shifted}$ will be added.
4. The sum of the CR segments and m_{prod} are computed and written back in place.
5. If necessary, any carry or borrow produced by the initial sum is resolved.

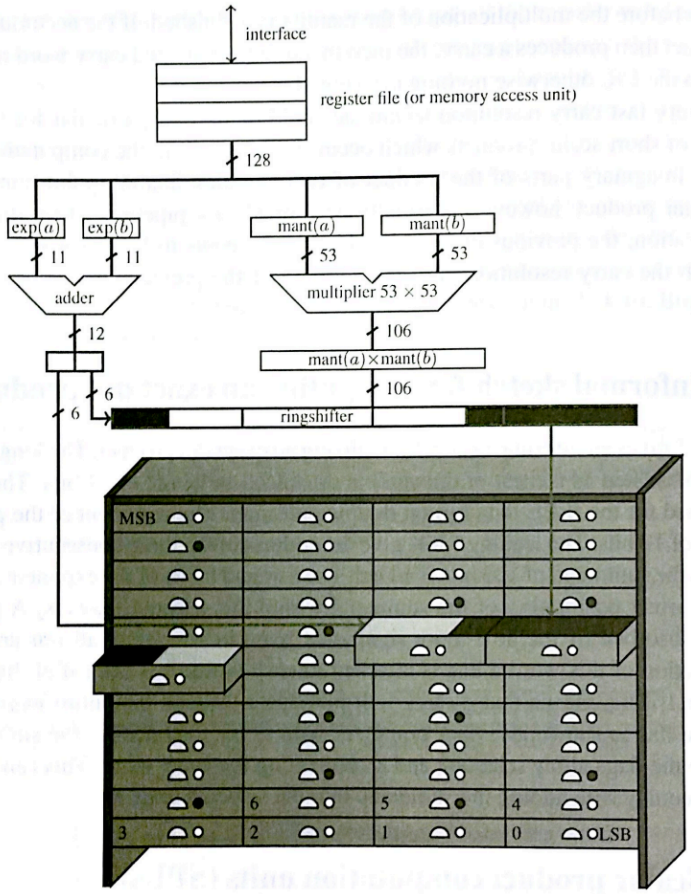

Fig. 1. High-Level Exact Dot Product Architecture

2.2 Architecture

The EDP implementations presented throughout the rest of this paper are based about the RoCC co-processor interface, with the EDP accelerator accepting custom RoCC instructions from a RISC-V Rocket CPU. The accelerator also has a dedicated port to the L1 Data Cache. Fig. 2. below shows a high-level diagram of the configuration

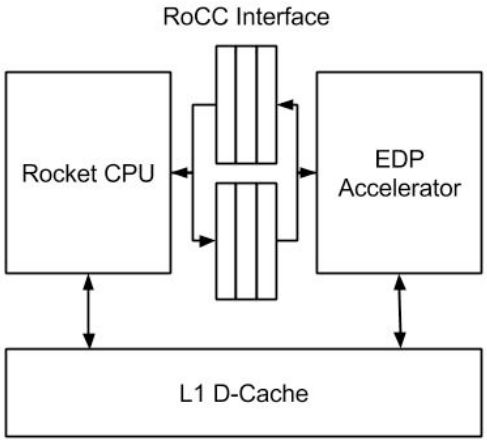

Fig. 2. High-level Rocket-EDPA System Diagram

In order to control the accelerator, we implemented the following RoCC instructions:

31 25	24 20	19 15	14	13	12	11 7	6 0	
0000000	00000	00000	0	0	0	00000	001011	CLR_CR
0001110	00000	00000	1	0	0	rd	001011	RD_DBL rd
0010000	00000	rs1	0	1	0	00000	001011	LD_CR rs1
0010100	00000	rs1	0	1	0	00000	001011	ST_CR rs1
0100000	rs2	rs1	0	1	1	00000	001011	PRE_DP rs1,
0100100	00000	rs1	0	1	0	00000	001011	RUN_DP rs1

Fig. 3. EDP Accelerator RoCC Instruction Encodings

Hence, the typical use case of the EDP accelerator involves sequences of Reset, Prepare DP, Run DP, and Read Double. While the user is not precluded from reading intermediate values between separate dot products, the EDPA cannot be interrupted while computing a dot product.

3 MICROARCHITECTURE

Every implementation of the presented EDPA, can be broken down into three major components:

1. Control: Responsible for decoding RoCC instructions and fetching operands from memory to feed the accelerator.
2. Front-end Datapath: Evaluates products of incoming operands and shifts them into alignment with CR.
3. Accumulator CR : Accumulates incoming products in place to the CR.

Full descriptions of each component and their architectural parameters follow.

3.1 Control

The control unit is responsible for accepting commands from the host processor and funneling data from memory to the datapath. It also rounds the CR in response to Read commands, loads the CR from memory, and stores the CR to memory. The control unit has three parts: the control state machines, rounding logic, and memory control logic.

3.1.1 State Machines

For simplicity and verifiability, we implemented mutually exclusive state machines (SMs) for each of the supported instructions. This decision led to some functional and state replication, but made designing and debugging significantly simpler.

3.1.2 Rounding Logic

The rounding logic takes two 64-bit words from the read SM and outputs a rounded double. Using a priority encoder to find the most significant bit, the rounding logic selects the appropriate most significant 52 bits for the mantissa (ignoring the implicit most significant bit of 1). It negates the mantissa if necessary since the CR is a signed fixed point number while the mantissa of a floating point number is unsigned. The exponent is calculated as shown in (1).

$$(1)\quad exp = (CR_{MSW} - W_{ZERO}) << 6 + bias + msb - b_{ZERO}$$

CR_{MSW} is the index of the 64-bit word in the CR containing the most significant bit of the current sum. W_{ZERO} refers to the 64-bit word of the CR that contains the bit representing 2^0. The *bias* comes from the IEEE standard. *msb* is the index of the most significant bit in the most significant word in the CR. b_{ZERO} refers which bit in W_{ZERO} represents 2^0. These are constants that depend on the floating point standard. For IEEE double precision, $W_{ZERO} = 33$ and $b_{ZERO} = 37$.

3.1.3 Memory Unit

The memory unit in Fig. 4. shows the logic and state elements required to move data from memory to the datapath as quickly as possible. The control unit alternates issuing loads from each array. When a load is issued for a particular array, the memory request tag is enqueued into the respective Memory Request Tag Queue (MRTQ) and the unified Memory Response Tag Store (MRTS). As the cache responds to load requests, the responses are stored in the Memory Response Data Store (MRDS). The front element of each MRTQ is matched against the tags in the MRTS; when the data corresponding to the matching tag becomes valid in the

MRDS, the data is forwarded to the respective Datapath Staging Queue (DPQ) and the entries in the MRTQ, MRTS, and MRDS are freed. When both DPQs have a valid entry, the DPQs simultaneously issue their respective floating point data into the datapath.

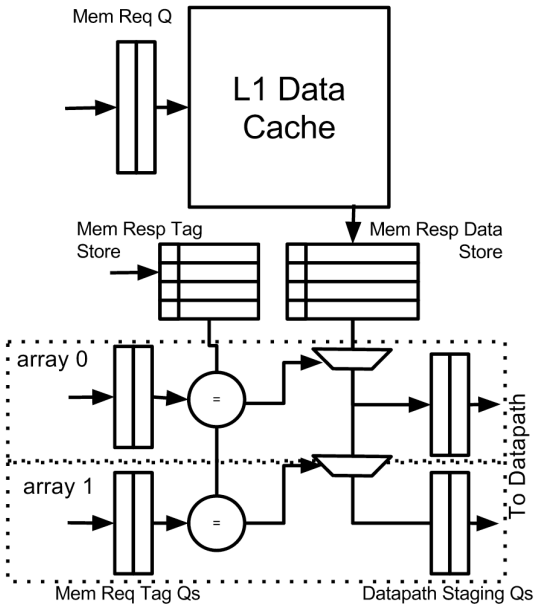

Fig. 4. Control Memory Unit

Given how predictable the data access pattern is for the EDPA, one of the biggest optimizations we implemented was a prefetcher. The idea is simple, every time the control unit issues a load to the start of a cache block, the prefetcher issues a prefetch request to a future block. The number of blocks ahead to prefetch is parameterized because different cache configurations will require prefetching further ahead.

3.2 Front-end Datapath

The front-end consists of three modules:

1. Exponent Adder (ExpAdder)
2. Multiplier
3. Shifter

These modules are straightforward. Aside from some initial floating point decoding (eg. prepending one to the mantissa or zero properly handle denormal numbers), the functionality of these blocks was expressed with Chisel's +, *, << operators respectively. The only parameterization in these modules was the generation of input and output registers. We found that design compiler would properly infer the correct synthetic module based on the amount of registering we appended or prepended to each block. Note that addition and multiple occur in parallel and thus have the same latency, hence $L_{tot} = L_{mult} + L_{shift}$.

Architectural Parameters:

1. Shifter Latency: 1 – 2 cycles
2. Multiply Latency: 1 – 4 cycles

3.3 Accumulator – Complete Register (CR)

The complete register is perhaps the most sophisticated component of the EDPA. We present two different schemes, one with adders distributed across the CR's length (Segmented) and one with a shared adder must read the appropriate subwords of the CR upon accumulation (Centralized).

3.3.1 Segmented Accumulator CR

This implementation slices up the CR into segments of width k, each with its own *k* bit adder. In a single cycle, a segment adds a portion of the m_{prod} and incorporates an incoming carry or borrow if present. If a carry or borrow is produced it is latched, where it be further propagated in the next cycle, akin to the implementation of a carry-save multiplier.

To ease carry propagation, and for most significant word detection, each segment includes two flags to denote if the segment is all high or all low.

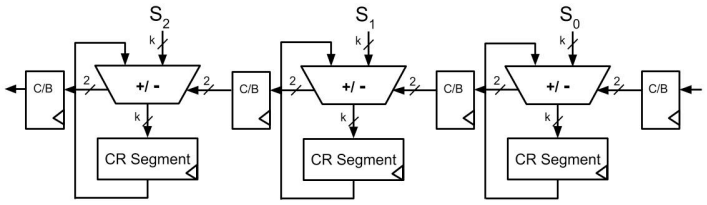

Fig. 5. Segmented Accumulator CR

Architecture Parameters:

- Segment Width: 16, 32, 64 bits

3.3.2 Centralized Accumulator CR

Given that the multiplicand for double precision floating point is 106 bits, we decided it would be prudent to implement a CR where a single adder is shared for accumulation. The centralized accumulator microarchitecture is shown in Fig. 6. below.

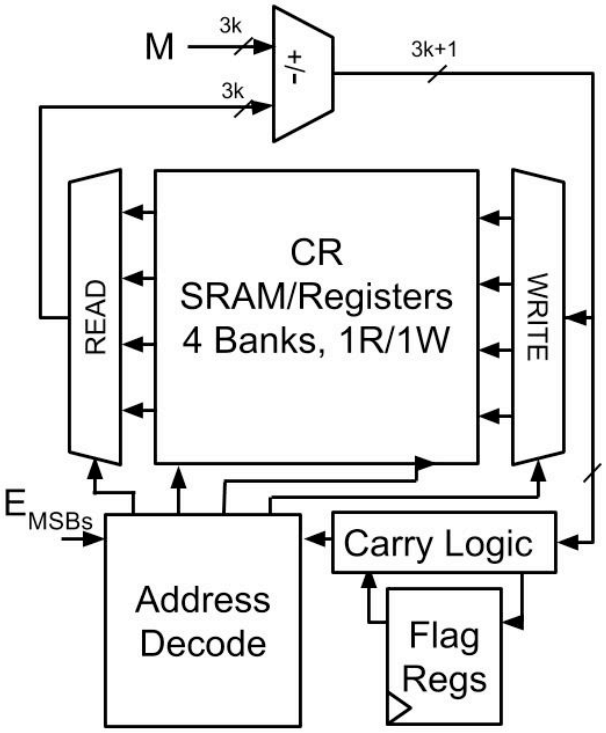

Fig. 6. Centralized Accumulator CR

The accumulator accepts the 192-bit summand from the shifter (106-bits with zero padding on either side), which has already been aligned to the 64-bit words in the CR. It reads the appropriate 4 words (as indicated by the summed exponents), and adds the summand to the bottom 3. The 4th word is incremented or decremented in the event of a carry or borrow respectively. The resulting sum is then written back to the CR during the next cycle.

It is possible for a carry or borrow to propagate beyond the 4th word. A common example of this is when the sign of the entire CR flips requiring carry/borrow propagation all the way past the top word. We accelerate this process by introducing two registers: allZeros and allOnes. Each register has 1 bit for each word in the CR and indicates whether the word is filled with all zeros or all ones respectively. When a carry propagates beyond the 4th word, the carry logic uses the allOnes register to find the next word in the CR that is not all ones. If there is such a word, the pipeline is stalled while that word is read, incremented, and then written back. Every word the carry propagated past (because it was all ones) is switched to being indicated as all zeros without any reads or writes to the CR. If the carry propagates past the highest word, then no pipeline stall is necessary as the carry is handled by modifying the all ones and all zeros registers. Borrows work in the same way except that borrows propagate past words that are all zeros, turning those words to all ones. One subtle point is that any time a word is read from the CR, the allZeros and allOnes registers are used to select whether the output of the CR or the appropriate all zero or all ones constant is used.

3.3.2.1 Register Accumulator CR

Implementing the CR as registers is the most naïve approach and serves as a good baseline for comparing to SRAM configurations. The register accumulator follows the above description exactly except that the summand can be written back in the same cycle it was calculated.

3.3.2.2 SRAM—8T, 1 Read, 1 Write

The most obvious energy and area efficient CR implementation is a dual-ported SRAM that can support reading the CR for an accumulation while writing back the previous summand. The only modification to the description above is that there must be a forwarded path from writeback to the read port since back-to-back accumulates may touch the same words in the CR.

3.3.2.3 SRAM—6T, 1 Read/Write Port

The 1 read/write port SRAM is very similar to the previous implementation except that reading and writing the CR can only occur in separate cycles. Thus, this configuration of the accumulator CR can only accept 1 accumulate every other cycle. It also takes 2 cycles to propagate a carry/borrow (unless it propagates beyond the top word as previously discussed). Given a standard 64-bit memory interface like the one provided by RoCC, each accumulate requires 2 reads anyway. Thus, accumulating every other cycle is sufficient in such a memory-bound system.

4 RESULTS

To evaluate the effectiveness of various implementations of the EDPA, we used two distinct flows:

4.1 System Level Benchmarking

To avoid the lengthy runtimes associated with running the proxy kernel (and with the intention of eventually running gate-level power estimation), we ran the bulk of the succeeding benchmarks bare-metal, using a framework derived from riscv-tools/riscv-benchmarks. Because the full system memory interface was limited to 64-bits, we also ran some EDP-level benchmarks to measure accelerator performance with a perfect cache.

4.2 Accelerator-level VLSI Flow

Initially, EDP design space was explored by pushing the entire system (Rocket + EDP) through both synthesis and place and route. However, since we found that our architectural knobs trivially changed benchmark performance (cycle count) and we were not able to get gate-level simulation working, we compiled the bulk of our design points without Rocket. This allowed us to aggressively use JackHammer to sweep over our design space for multiple clock rate targets, using both RVT and multi-VT flows. All area, timing and power numbers were gathered from icc after chip-finishing, power measurements, therefore use ICC default activity factor of 10%.

4.3 Benchmarking Results

While the primary purpose of the EDPA is exact and reproducible floating point dot product, its performance is also central to our evaluation. Unfortunately, the RoCC memory interface is limited to a single 64-bit request per cycle, so it was impossible for us to saturate our datapath in full system simulation. Nevertheless, our accelerator vastly outperformed Rocket+FPU despite less than 50% utilization. Fig. 7. below shows the number of cycles required to execute a set of benchmarks on the four primary configurations of our design as well as on Rocket. DP N stands for dot product with vectors of length N. Matmul NxN stands for matrix multiply of two matrices, each of size NxN.

	Segmented CR		SRAM 1R 1W CR		Rocket
	No Prefetch	Prefetch	No Prefetch	Prefetch	
DP 1,000	11,737	5,711	11,533	5,721	30,657
DP 10,000	105,625	39,477	105,630	39,632	296,956
MatMul 10x10	8,432	7,969	8,522	8,035	26,307
MatMul 100x100	9,770,484	5,792,404	9,765,090	5,781,974	29,437,604
Kahan Sum DP 1,000	NA	NA	NA	NA	45,903
Kahan Sum DP 10,000	NA	NA	NA	NA	449,519

Fig. 7. Full System Simulation Performance (cycles)

As Fig. 7. illustrates, on each of the common benchmarks, the EDPA performed very well. Prefetching increased performance over 2x for dot product and still a significant amount for large matrix multiplication (and the memory system was still far from saturated). An important distinction here is that while the dot products and matrix multiplications run on the EDPA gave the correct result, every benchmark on Rocket gave incorrect results. We included the Kahan Summation dot products to show that algorithms that reduce error add overhead yet are still less accurate that our accelerator.

While we were unable to run MPFR on Rocket, we did compare MPFR running on an x86 server to hardware floating point in order to get an idea of the overhead of exact floating point in software. We found that MPFR ran 2-3 orders of magnitude slower than hardware. EDPA runs 3-6x faster than hardware floating point, yet retains the accuracy of MPFR which runs on the order of 100-1000x *slower*. To be fair, MPFR does much more than dot product and is not intended to accelerate that function alone; however, this comparison does illustrate the overhead of exact dot product in software.

One aspect Fig. 7. does not show is any differentiation between the segmented CR and the dual-ported SRAM CR implementations. This is due to the full system memory's inability to saturate the EDPA. It is fundamentally limited by a 64-bit interface. We also ran lone accelerator simulations with a perfect cache to determine accelerator performance when it is fully saturated. See Fig. 8.

	Segmented	1R 1W SRAM	1RW SRAM
64-bit Memory	20106	20108	20110
128-bit Memory	12605	12608	20108

Fig. 8. 64-bit vs. 128-bit Memory Interface (Perfect Cache)

Each configuration executed a dot product with vectors of length 10,000. As expected, with only half utilization, each configuration took just over 20,000 cycles to complete. The interesting point is when the memory is capable of reading two loading per cycle. Both the segmented and 1R,1W ported SRAM are designed to handle a multiply-accumulate every cycle, thus they both finish about as fast as the control could pass the data. The 1RW ported SRAM can only accept 1 multiply-accumulate every other cycle, so it still took the same amount of time despite the increased memory bandwidth.

4.4 VLSI Results

Summarized in Fig. 8, are power and area estimates for three full-system design points, Rocket alone, Rocket with Segmented EDP, and Rocket with n Centralized-SRAM EDP. The target clock period was 3.5 ns, using only RVT cells.

	Area - ICC			Power - ICC		
Design Point	**Total *um²***	**EDP *um²***	**%**	**Power *mW***	**EDP *mW***	**%**
Rocket	860978	N/A		83.8	N/A	N/A
Segmented	1021157	180679	17.8	111	25.1	22.5
Centralized	969622	106282	11.0	102	18.4	18

Fig. 9. Total Area and ICC Power estimations for a complete Rocket + EDPA system

Fig. 10 below presents energy estimations for 119 parameterizations of the EDPA, across a variety of frequencies. It took considerably less effort for ICC to place and route the centralized designs over segmented ones, even though a couple segmented parameterizations with small segmented widths successfully closed timing at 571 MHz – comparable to some of the faster centralized implementations. We suspect dedicated floorplans for Segmented implementations would meet tighter timings.

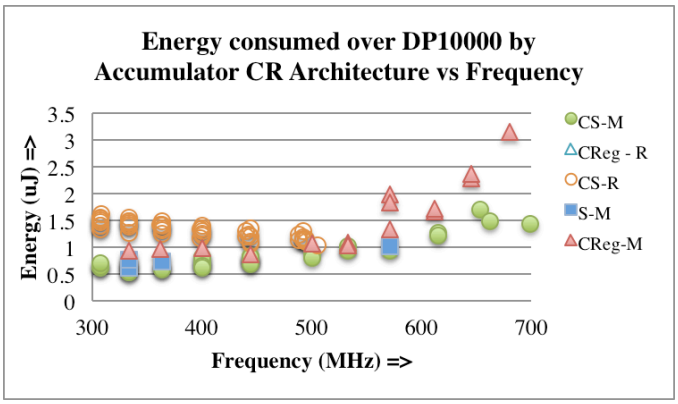

Fig. 10. Energy consumption for a single run of DP10000. Note, C = Centralized-SRAM, S = Segmented, CReg = Centralized-Registered. M = MVT, R = RVT cells

Perhaps the most important result of Fig. 10 is that the centralized-SRAM implementations lie exclusively along the pareto-efficient energy frontier, though segmented implementations come close at the minimum-energy point. Moreover, multi-threshold voltage designs significantly outperform the standard except around 500MHz. This is perhaps because the increasing use of LVT cells counteracts the gains once provided by HVT, roughly equating to an all-RVT design.

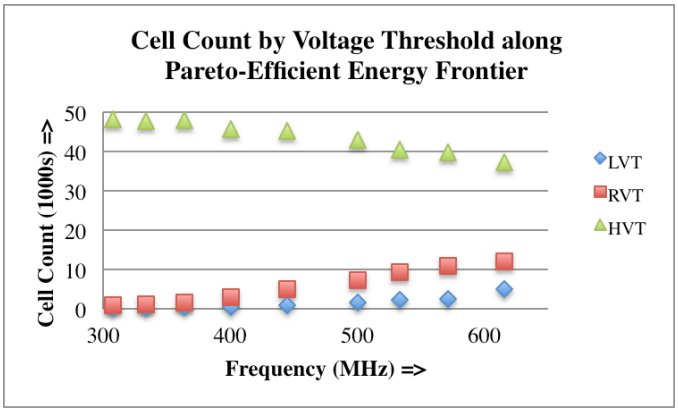

Fig. 11. Cell count by threshold voltages for parameterizations of Centralized-SRAM.

For the remaining figures, we study centralized-SRAM configurations more carefully. We found that using an multi-threshold voltage flow was critical in enabling the flow to meet timing past 500 MHz. Fig. 11, above demonstrates the increased presence of LVT cells in faster designs.

Additionally, deeply pipelining the front-end datapath was crucial beyond 500 MHz, with parameterizations with pipeline depths of 5 or 6 stages through the multiplier performing best at these frequencies. (It was difficult to meet timing otherwise.) Fig. 12 below speaks to this increase in pipelining – while accumulator-CR and control power dissipation hold steady and fall respectively, the multiplier, shifter and exponent adder (not shown) all increase.

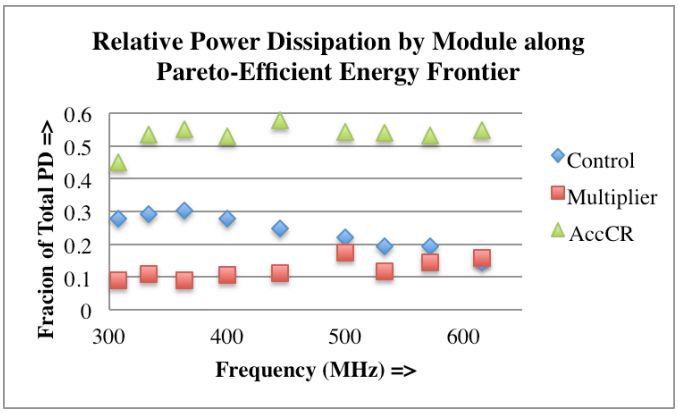

Fig. 12. Fraction of power dissipation in three largest consumers.

5 Conclusion and Future Work

In this paper, we proposed a solution to the need for high-speed, reproducible, exact floating point. We explored the design space of an exact dot product

accelerator that significantly outperformed a standard floating point unit while maintaining the complete precision of the product. While we are please with the way the project came together, there are several directions we hope to take the project.

First, our power results are mostly from standard icc activity factor guesses. We would like to get full system benchmarking running in PrimeTime PX to get a better idea how our energy efficient our accelerator is compared to standard floating point operations and software based solutions. It is also important to get software libraries like MPFR ported to RISC-V for more direct comparison.

Second, given the memory bandwidth limitations of the current RoCC interface, we would like to connect our accelerator directly to the L2 cache in the hope we might better saturate the datapath.

Third, we are interested in how exact dot product might be implemented in a vector unit. The RISC-V Hwacha vector co-processor is a potential platform for such evaluation. It may be that most of the benefits of our accelerator can achieved by a more general purpose computing platform.

Fourth, implementing exact dot product as a co-processor is an interesting experiment, but it still suffers from the issue of complicating the floating point programming model when precision and reproducibility are important. We would like to integrate our accelerator into the CPU's floating point unit and see how we might be able to simplify the programming model (eg. by saying that FP register 0 is exact) and share hardware with the FPU, thus saving area and power.

This project has yielded interesting results and shown that complete floating point arithmetic is possible in a high-performance and energy-efficient way. It is not yet clear if an EDPA is the *best* solution, but we hope to find the answer to that question in the coming months.

6 REFERENCES

[1] *The GNU MPFR Library* [Online]. Available http://www.mpfr.org/

[2] U. Kulisch, "Scala products and complete arithmetic," in *Computer Arithmetic and Validity,* 2nd ed. Berlin, Germany: de Gruyter, 2013, pp 249-304.

[3] Vo, Huy. "A Case for OS-Friendly Hardware Accelerators", *7th Annual Workshop on the Interaction between Operating System and Computer Architecture (WIVOSCA-2013), at the 40th International Symposium on Computer Architecture (ISCA-2013)*, Tel Aviv, Israel, June 2013.

[4] Chisel [Online]. Available https://chisel.eecs.berkeley.edu/

[5] RISC-V [Online]. Available http://riscv.org/