Microarchitectural Comparison of the MXP and Octavo Soft-Processor FPGA Overlays

CHARLES ERIC LAFOREST and JASON H. ANDERSON, University of Toronto

Field-Programmable Gate Arrays (FPGAs) can yield higher performance and lower power than software solutions on CPUs or GPUs. However, designing with FPGAs requires specialized hardware design skills and hours-long CAD processing times. To reduce and accelerate the design effort, we can implement an overlay architecture on the FPGA, on which we then more easily construct the desired system but at a large cost in performance and area relative to a direct FPGA implementation. In this work, we compare the micro-architecture, performance, and area of two soft-processor overlays: the Octavo multi-threaded soft-processor and the MXP soft vector processor. To measure the area and performance penalties of these overlays relative to the underlying FPGA hardware, we compare direct FPGA implementations of the micro-benchmarks written in C synthesized with the LegUp HLS tool and also written in the Verilog HDL. Overall, Octavo's higher operating frequency and MXP's more efficient code execution results in similar performance from both, within an order of magnitude of direct FPGA implementations, but with a penalty of an order of magnitude greater area.

Categories and Subject Descriptors: C.1.3 [**Processor Architecture**]: Other Architecture Styles—*Adaptable architectures*; C.4 [**Performance of Systems**]: Measurement Techniques, Design Studies

General Terms: Design, Performance, Measurement

Additional Key Words and Phrases: Benchmarking, FPGA, soft-processor, multi-threading, vector, overlay

ACM Reference Format:
Charles Eric LaForest and Jason H. Anderson. 2017. Microarchitectural comparison of the MXP and Octavo soft-processor FPGA overlays. ACM Trans. Reconfigurable Technol. Syst. 10, 3, Article 19 (May 2017), 25 pages.
DOI: http://dx.doi.org/10.1145/3053679

1. DESIGNING WITH FPGAS

Despite being easier to design with than ASICs, having a higher performance than CPUs, and using less power than GPUs (for a given performance level), implementing designs on FPGAs remains difficult: Only a minority of engineers (about 1-in-16) possess the specialized skills necessary to do hardware-level design work [United States Bureau of Labor Statistics 2012][1] Furthermore, as the size of the FPGA devices keeps increasing but without corresponding speedup from the Computer-Aided Design (CAD) tools [Choong et al. 2010], compiling ever-larger designs takes many hours or days (e.g.,

[1]In the US: roughly 1,361,700 computer software engineers/programmers vs. 83,300 computer hardware engineers.

Funding came from the University of Toronto ECE Department, the Queen Elizabeth II World Telecommunication Congress Graduate Scholarship in Science and Technology, the Walter C. Sumner Foundation, and NSERC.
Authors' addresses: C. E. LaForest and J. H. Anderson, Edward S. Rogers Sr. Department of Electrical and Computer Engineering, University of Toronto, 10 King's College Road Toronto, ON M5S 3G4 Canada.

DOI: http://dx.doi.org/10.1145/3053679

3–30 hours [Krasnov and Schultz 2007], 5h [Tsoi and Luk 2010], "one or more CPU hours" [Ludwin and Betz 2011], "over 35 hours" [Kindratenko et al. 2007], and so on [Wu and McElvain 2012]). Detailed CAD time breakdowns can be found in Murray et al. [2013]. These growing design cycles also strain our efforts at simulation and verification. The lengthening design cycle, sometimes dubbed the *productivity gap* [ITRS 2011], eats away at the two main advantages of FPGAs: a fast time-to-market and lower non-recurring engineering design costs.

1.1. The FPGA Overlay Design Process

FPGA overlays primarily differ from Hardware Description Languages (HDLs) such as VHDL and Verilog and High-Level Synthesis (HLS) from C or other programming languages by providing a persistent layer of abstraction between the implementation of a system and the underlying FPGA hardware. Implementing a system on an FPGA overlay does not require using FPGA CAD tools, avoiding their long design cycles and hardware-specific details. By analogy, an interpreted language (e.g., Python) provides an overlay over the CPU hardware, and the CPU itself acts as an overlay over VLSI logic, with the same costs and benefits:

—We can use higher-level abstractions to describe the system.
—The design cycle reduces to a rapid re-compilation to the overlay.
—A system design becomes mostly portable across different implementations of the same overlay.
—We pay a performance and size penalty relative to the system underlying the overlay.
—We can incrementally augment the overlay with custom "accelerators" to improve application-specific performance.

Because overlays provide a layer of abstraction, multiple approaches exist with varying programming models. Briefly, existing overlays include scalar soft-processors (e.g., Nios II/f [Altera 2014], iDEA [Cheah et al. 2014], and GRVI [Gray 2016]), vector processors (e.g., VectorBlox' MXP [Severance and Lemieux 2013b; Severance et al. 2014]), multi-threaded soft-processors (e.g., Octavo [LaForest and Steffan 2012, 2013; LaForest et al. 2014], CUSTARD [Dimond et al. 2005], UTMT II [Fort et al. 2006], and others [Labrecque and Steffan 2007; Labrecque et al. 2008; Labrecque and Steffan 2009]), "virtual" FPGA fabrics (e.g., ZUMA [Brant and Lemieux 2012]), and Coarse-Grain Reconfigurable Arrays (CGRAs) (e.g., QuickDough [Liu et al. 2015] and [Capalija and Abdelrahman 2013]).

In this work, we compare the MXP and Octavo soft-processor overlays, with some results presented on a Nios II/f soft-processor where necessary, since they can all be implemented on the same Altera FPGA platform (see Section 3). The other multi-threaded soft-processors pre-date our FPGA platform and would have required re-design work. We did not compare the "virtual" FPGA fabrics and CGRAs, as we wanted to compare soft-processors specifically. Finally, the remaining scalar soft-processors, while having high performance, are limited to Xilinx FPGA devices: iDEA [Cheah et al. 2014] is built around the unique features of the DSP48E1 block, while GRVI [Gray 2016] is carefully technology-mapped and floorplanned for a Xilinx FPGA.

Overlay Design Cycle. In a nutshell, the overlay design cycle moves most of the iterations to software development instead of hardware development, avoiding the time-consuming Place-and-Route (P&R) of circuits onto the FPGA after each design change. P&R consumes the majority of FPGA CAD processing time. Instead, the target application drives the selection of a particular instance of an overlay architecture, which gets synthesized and P&R'ed onto the FPGA. The designer then implements the application as software on the overlay, with rapid design cycles. If the overlay

lacks a necessary feature for functionality, performance, or ease-of-use, only then does the design process go through a hardware iteration to produce a new overlay. Given that each P&R run can take several hours, on top of the actual hardware design effort, using overlays leads to a faster and easier overall design cycle as both the software and hardware cycles progress more incrementally, with feedback between them: The software designer can write simpler software on overlay hardware tailored to the application domain, and the hardware designer is freed from implementing and verifying an entire application in hardware, providing only the "heavy lifting" components required to meet performance constraints, leaving complicated state and coordination to software.

1.2. Contributions

This work evaluates micro-architectural benchmarks on the MXP and Octavo soft-processors, with the area and performance penalties relative to the underlying FPGA hardware provided by HLS and HDL hardware implementations of the same micro-benchmarks.

—We describe Octavo, a multi-threaded soft-processor architecture for constructing FPGA overlays, summarizing previous works [LaForest and Steffan 2012, 2013; LaForest et al. 2014] and further describing an extended address space and an instruction predication mechanism that improves Octavo's high performance and flexibility.
—We compare the micro-architectural performance of a SIMD-extended Octavo and the MXP soft vector processor via sequential and parallel micro-benchmarks. For reference, we also evaluate hardware implementations from the LegUp HLS platform and manually written Verilog.

1.3. Organization

Section 2 briefly overviews the Octavo and MXP soft-processor micro-architectures. Section 3 describes the sequential and parallel benchmarks we use to compare implementations. Section 4 describes adaptations to Octavo to support specific benchmarks. Section 5 presents and analyzes the benchmark results. Section 6 summarizes the significant micro-architectural differences between the soft-processors, and Section 7 points out salient design guidelines for soft-processor overlays. Appendix A details the current micro-architecture of the Octavo soft-processor, with improvements since its original publication.

2. THE OCTAVO AND MXP OVERLAYS

As overlays, we compare two soft-processors: our own Octavo [LaForest and Steffan 2012, 2013; LaForest et al. 2014], MXP [Severance and Lemieux 2013a; Severance et al. 2014], a vector soft-processor derived from UBC's VENICE [Severance and Lemieux 2012], and commercialized by VectorBlox Computing. We compare the performance of Octavo and MXP implementations with 1 to 32 lanes, denoted as L1 to L32 for Octavo and V1 to V32 for MXP. The notation reminds us that Octavo uses independent SIMD lanes with private memories, while MXP uses vector lanes accessing a common banked vector scratchpad. MXP works on 32-bit values, while Octavo uses 36-bit values, matching the FPGA's Block RAMs.

2.1. The MXP Soft Vector Processor

Figure 1 [Severance and Lemieux 2013a] shows the block diagram of a four-lane example configuration (denoted as MXP-V4) of the MXP soft vector processor. The MXP vector lanes operate under the control of a Nios II/f scalar soft-processor, which sends

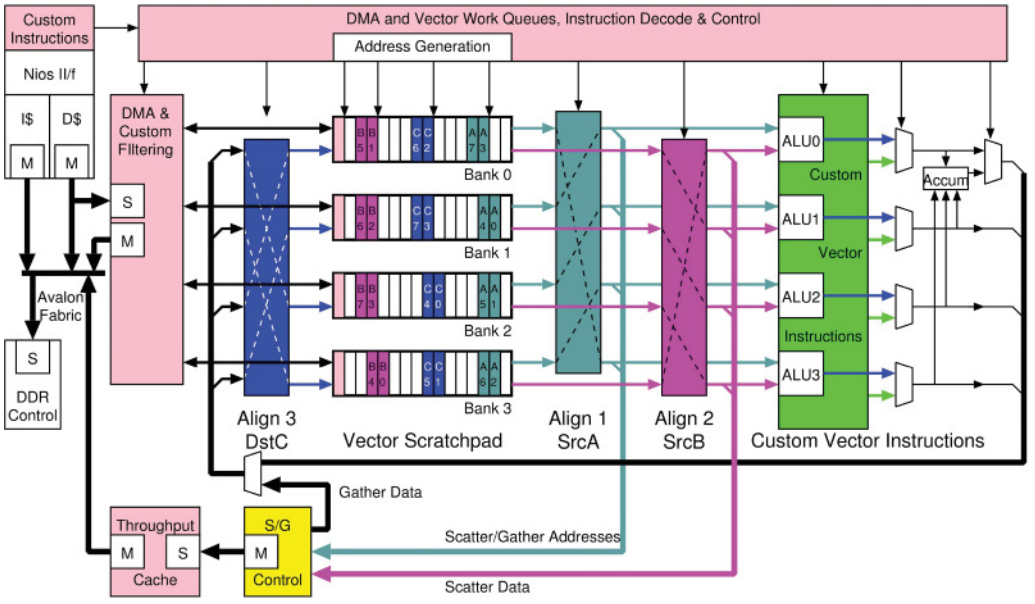


Fig. 1. MXP block diagram.

vector instructions to the various MXP units to perform vector DMA transfers to/from main memory, scatter/gather addressing [Severance and Lemieux 2013b], and optional custom vector instructions for specific tasks.

For benchmarking purposes, MXP's most relevant architectural features include the following.

Double-pumped Vector Scratchpad. MXP's vector scratchpad operates internally at twice the system clock frequency to multiplex the two ports of its constituent Block RAMs into four. This configuration enables multiple simultaneous transfers each cycle: two vector reads, one vector write, and a DMA transfer to/from main memory. However, the double-pumped scratchpad ultimately limits MXP's overall maximum operating frequency (F_{max}).

Vector Alignment Networks. The alignment networks are MXP's most significant feature, enabling arbitrarily aligned vector addressing. They operate independently on reads and writes to the scratchpad and perform simple rotations, not full cross-bar permutations. Figure 1 illustrates them aligning two source vectors A and B before reaching the ALUs, and one destination vector C before storage back into the scratchpad.

Vector-to-Scalar Accumulator. MXP contains one accumulator, located after the vector ALUs and taking input from each vector lane to enable pipelined Multiply-Accumulate operation.

Vector Conditional Move. MXP provides a variety of vector conditional move instructions that provide basic conditional execution in the absence of vector branches.

2.1.1. Programming Model. The MXP programming model strongly depends on vectorizing the target application. The 8- to 10-stage vector pipeline does not support forwarding nor branching, and the Nios II/f takes 2 to 4 cycles to issue a complete vector instruction. Thus, the vector lanes must operate on vectors at least 8× as wide as

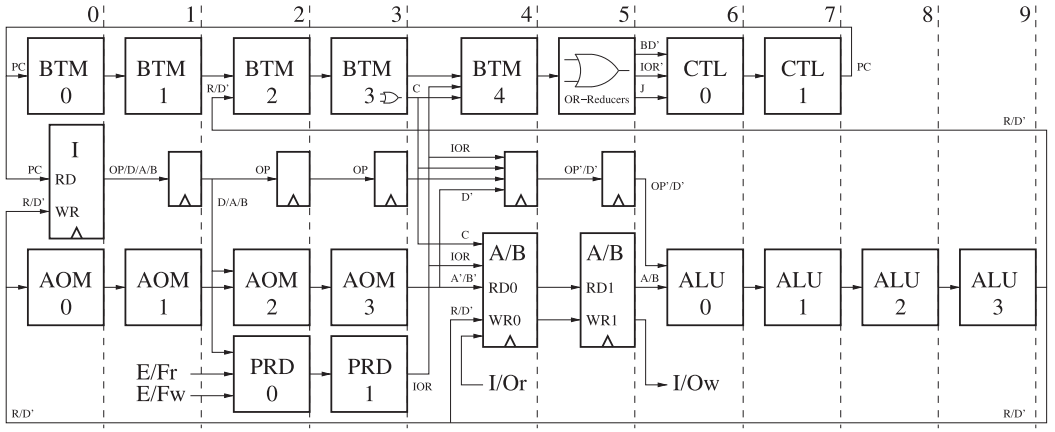


Fig. 2. Octavo block diagram.

the number of vector lanes to avoid consuming vector instructions faster than Nios II/f can supply them and to avoid pipeline bubbles. On the other hand, once initiated, MXP's vector operations proceed efficiently, with automatic loop counters and two-dimensional (2D) vector addressing, and scale effectively with the number of vector lanes. Additionally, the same MXP vector program portably compiles to MXP instances with any number of lanes, hiding the distribution of data and division of labour from the designer.

2.2. The Octavo Multi-Threaded Soft-Processor

Figure 2 shows the block diagram of a scalar (one-lane) Octavo soft-processor (denoted as Octavo-L1). At a high level, Octavo divides its 10-stage pipeline among 8 round-robin hardware threads, with instructions moving left to right. The output of the Instruction Memory (I) addresses the Data Memories (A/B), which also contain memory-mapped I/O ports (I/Or and I/Ow). In parallel, the Branch Trigger Module (BTM) enables multi-way zero-cycle branches based on the result of the previous instruction of the current thread, possibly cancelling the current instruction if the branch does not go as statically predicted. It provides its output to the Controller (CTL), which issues the next Program Counter (PC) value for each thread. Also in parallel, the Address Offset Module (AOM) modifies the instruction operands to enable indirect and offset memory addressing to share code across threads and can post-increment addresses after an indirect memory read/write. The I/O Predication Module (PRD) automatically annuls and replays instructions which attempt to access I/O ports that are not ready. We discuss Octavo's architecture in detail in Appendix A.

For benchmarking purposes, Octavo's relevant architectural features include the following.

SIMD Lanes. By replicating the datapath (A/B data memories and the ALU), and feeding the copies a common instruction stream, we create optional SIMD lanes. Each lane has private A/B memories with memory-mapped I/O ports and follows the control flow of the original datapath. All datapaths execute the same code, so we add $n - 1$ datapaths to support n-way SIMD parallelism. We logically partition each additional SIMD lane to preserve maximum operating frequency (F_{max}) [LaForest and Steffan 2013].

Memory-Mapped I/O Ports. Instead of loads/stores, Octavo maps I/O ports into the A/B data memories, addressing them using the instruction operands like any other

memory location. This tightly-coupled external interface simplifies the implementation of custom functional units. We can control these units in software with little overhead, and we do not have to alter Octavo's datapath to integrate them.

2.2.1. Programming Model. The Octavo programming model strongly depends on multi-threading the target application. Each of the eight threads executes an instruction in turn, at an effective $\frac{1}{8}$th of the actual F_{max}. Thus, to use Octavo's full capacity, all eight threads must do useful work, and the division of labor depends on the nature of the workload. For example, we can divide the *code* of sequentially dependent calculations into multiple functions, with each thread executing a different function, implementing pipeline parallelism. Alternately, we can divide the *data* of data-parallel operations among all threads, all running the same shared code. Adding SIMD parallelism simply extends this model, since each additional datapath executes the same eight threads but operating on different data memories. Once initiated, all threads proceed efficiently without branching and addressing overhead or pipeline hazards. Each program must be tailored to the particular Octavo configuration that will execute it, manually distributing data to each thread before computation.

2.3. Hardware Reference Implementations

To measure the performance and area penalties of the Octavo and MXP soft-processor overlays relative to the underlying FPGA hardware, we also implement the micro-benchmarks directly on the FPGA, both manually written and automatically synthesized.

Hardware Description Language. For each benchmark, we manually create a hand-optimized Verilog-2001 implementation that aims for the highest performance possible without changing the underlying algorithm. All HDL implementations work with 32-bit data and narrower words as required for addresses, counters, and so on. All data are stored in Block RAMs (BRAMs) and accessed in the same sequence and with the same amount of parallelism as the algorithm describes. Loops are not unrolled but pipelined as needed to maximize operating frequency (F_{max}).

High-Level Synthesis. There exist multiple HLS systems, but we chose the LegUp [Canis et al. 2013] platform as it synthesizes plain C code, making the benchmarks more portable from MXP and Octavo than the alternatives. We use LegUp 3.0, with some in-progress improvements to pipelining and local memories [Canis et al. 2014; Fort et al. 2014]. LegUp uses 32-bit values for synthesis, automatically optimized to narrower widths by the FPGA CAD tool (Quartus) where possible.

3. BENCHMARK DESCRIPTIONS

In this section, we describe the MXP, Octavo, LegUp, and HDL implementations of sequential and parallel micro-benchmarks.

Experimental Framework. All systems target the Altera Stratix IV `EP4SGX230KF40C2` device on the Terasic DE4-230 board. All benchmarks execute entirely from on-chip memory to avoid the complications of a memory hierarchy, with the exception of some Nios II/f results which access external memory via a direct-mapped cache. The MXP system uses GCC 4.1.2 to compile code to the Nios II/f controlling processor, and synthesizes the vector lanes using Altera 13.0sp1. We use Quartus 13.1 to synthesize Octavo, with an aggressively tuned configuration for maximum F_{max}.[2] LegUp uses Quartus

[2]We report the unrestricted F_{max}, which ignores the speed limitation on the FPGA BRAMs (the slowest block type in Stratix IV), as some HDL implementations are able to exceed that limit.

10.1sp1 for synthesis, while the HDL implementations use Quartus 13.1. Both also use the same aggressive configuration as the Octavo synthesis.

Data Types. All soft-processor benchmark implementations operate on wordwide signed integers: 32 bits for MXP and 36 bits for Octavo. The soft-processors also use whole words for addressing and counting. The HDL implementations use 32 bit words for data, except for the Finite-State Machine (FSM) benchmarks, which process characters of 8 bits and use the minimum word width required for addressing and counting. Note that the LegUp `C`-based implementations use 32-bit `int`s for all data, loop counters, and other datapath components (except 8-bit `char`s for the aforementioned FSMs). Quartus cannot always optimize the unneeded bits, increasing the area of LegUp circuits.

Working Set Size. To present each platform in the best light possible, we scale up the working set size of each benchmark as necessary to minimize setup overheads and avoid artificial inefficiencies such as pipeline hazards, while still fitting into on-chip memory. Therefore, we report benchmark results per processed working set item ("unit of work") to abstract away the different raw working set sizes. In general, the HDL and LegUp implementations work on sets of 100 items, while Octavo works on sets of 1,024 items, evenly divisible into per-thread working sets of 128 elements, and replicates the working set for each additional SIMD lane. MXP makes the working set equal to the size of its scratchpad divided by 4 to leave space for all input, output, and temporary vectors. For example, MXP-V1 and MXP-V2 use an 8kB scratchpad, with 4 bytes per word (2 kilowords), yielding a working set of 512 items. For larger numbers of vector lanes, MXP multiplies this working set by the number of vector lanes (i.e., MXP-V4 works on 1,024 items, MXP-V8 on 2,048, and so on...). *All working set data is stored in Block RAMs (BRAMs)*, except for some LegUp cases where the CAD tool inferred distributed LUT-based memory instead. We address these cases individually.

3.1. Sequential Benchmarks

The Sequential Benchmarks represent tasks that do not parallelize easily or at all, and often present a worst-case scenario for SIMD/vector processing, as well as stress intrinsic sources of overhead such as pipeline hazards, memory accesses, and flow-control.

```
1 for i = 0 to COUNT-1                              // For each vector element
2    temp_vector[(COUNT-1)-i] = data_vector[i]; // Store at other end of temp vector
3 data_vector = temp_vector;                        // Write back to original vector
```

Listing 1. Reverse-2D MXP Pseudo-Code.

Reverse-3. Reverses an array of wordwide integers using the conventional three-step approach, which assumes all elements are local to the same memory: $A \rightarrow t$, $B \rightarrow A$, $t \rightarrow B$. MXP cannot implement a three-step swap directly but instead fills its pipeline with vector elements, storing them at the correct computed address to perform the reversal. A 2D addressing mode automatically computes both source and destination addresses in parallel. Listing 1 outlines the MXP "Reverse-2D" algorithm, which we consider equivalent to Reverse-3, since it can also only move one item at a time. The MXP read/write alignment networks cannot reverse a vector, since they only rotate vector elements. The HDL implementation sequentially reads the A and B elements from a single BRAM, places them in a short pipeline, and then stores them back in reverse order.

Hailstone-S. Computes one step of a Hailstone sequence: If n is even: $n = n/2$, else $n = (3n + 1)/2$. We apply this function over many (100 or more, depending on

the platform) random positive integer initial seeds to evenly distribute time spent in the even and odd branches and to fill pipelines with independent calculations. The Hailstone benchmark performs a variety of bit-level and arithmetic calculations, as well as unpredictable branching. The HDL implementation sequentially reads seeds from a BRAM, computes both even and odd branches in parallel, and then writes the correct result back into memory.

Hailstone-A. Computes the entire Hailstone sequence of the seed $77,031$, which has the longest Hailstone sequence (222 terms) of all seeds $<$100,000. However, we accelerate the calculations via pre-computed table lookups, which allows each step to calculate the 8th value after the current seed rather than the immediately consecutive value. Thus, we can sequentially pre-compute the first 8 members of the Hailstone sequence then use these to seed 8 interleaved instances of the accelerated table-based calculations to compute the entire sequence 8 members at a time. Each instance only needs to compute 28 steps of a total of 224 steps (28×8), producing two extra terminal sequence results $(\ldots, 2, 1)$ after the 222 terms of the Hailstone sequence. This benchmarks resembles table lookups in cryptographic functions and also allows full usage of the MXP and Octavo pipelines. The HDL implementation sequentially reads one of eight seed values from a BRAM, and performs two parallel table lookups based on the seed (two BRAMs) and a multiplication to generate the next value and then stores it back to seed memory. This HDL implementation requires two DSP blocks and three BRAMs.

FSM-S and FSM-A. Executes a Finite-State Machine (FSM) that recognizes simple floating-point numbers (e.g., 0.5, -9., .3, 4.2, and so on...) from a stream of characters,[3] reaching either the `ACCEPT` or `REJECT` states. The input set of 103 characters contains 25 valid numbers, which each exercise one of all the possible paths to `ACCEPT`, plus one invalid number that reaches `REJECT` to ease verification. Two versions of the source code exist: FSM-S was written in conventional "structured" `C` using nested if-statements and state variables, while FSM-A was written in a more "assembly" style of `C` with lower overhead, using `goto` statements to change state. Both FSM benchmarks present worst-case data-dependent branching behaviour, with basic blocks of only two or three instructions and no regular loops. The HDL implementation uses the "structured" version, reading a character from a BRAM each cycle and raising `ACCEPT` or `REJECT` signals when the state machine (implemented as nested if-statements) reaches those states.

3.2. Parallel Benchmarks

The Parallel Benchmarks represent tasks that conventionally parallelize well, even if their implementations have more initial overhead than the equivalent non-parallelizable Sequential benchmarks. These tasks reveal how the work divides across parallel execution units and how well the implementation scales.

Increment. Adds 1 to each of 10 wordwide integer array elements, 10 times, for a total of 100 iterations designed to test simple independent data operations and loop overhead. We do not unroll the loops in the source but allow the LegUp HLS to pipeline them automatically. The HDL implementation sequentially reads each of the 10 values from a BRAM, increments them by 1, and then stores them back. These steps are sufficiently pipelined to allow them to run concurrently. Contrary to other benchmarks, Increment focuses on looping itself and thus has a small working set size.

Reverse-4. Reverses an array of wordwide integers using a generalized swap that does not assume that the elements reside in the same memory: $A \rightarrow t_1$, $B \rightarrow t_2$, $t_1 \rightarrow B$, $t_2 \rightarrow A$.

[3]Actually, 32 and 36-bit `int`s for MXP and Octavo, and 8-bit `char`s for HDL and LegUp.

This approach allows dividing the array into two or more memories, enabling parallel transfers. MXP cannot implement Reverse-4 directly, since its alignment networks can only rotate vector elements, not arbitrarily permute them. Instead, MXP implements a "Reverse-Recursive" algorithm that uses conditional moves (`CMOV`) to recursively swap quadratically increasing large subsets of a vector. This algorithm performs approximately $2log_2n$ vector swaps for a vector of length n. We first set a vector mask that allows `CMOV` to only write every other vector element. We then copy all the odd-numbered elements to the even-numbered locations in a temporary vector and then copy the even-numbered vector elements into the temporary odd-numbered locations, thus swapping their positions. We can then adjust the mask and the vector addresses to swap pairs of elements, quartets, and so on. We consider Reverse-Recursive comparable to Reverse-4, since they both move multiple items at a time and similarly scale with the number of SIMD/vector lanes. The HDL implementation evenly divides the array across two BRAMs for parallel operation. Each cycle, we read and write to both BRAMs, exchanging array values until the entire array is reversed.

Hailstone-N. Functions identically to the sequential Hailstone-S benchmark, calculating one step of the Hailstone sequence over a working set of about 100 random positive integer initial seeds (depending on platform) but implemented as non-branching code to allow SIMD and vector parallelization. All versions compute both the even and odd branches and then store the desired result based on the parity of the seed (i.e., even or odd). The LegUp and HDL versions use AND-OR Boolean masking (`((odd_branch & mask)|(even_branch & !mask))`) for best synthesis, while Octavo uses a masked XOR (`(((odd_branch ⊕ even_branch) & mask)⊕ even_branch`) to save a cycle, and MXP uses conditional moves (`CMOV`). The HDL implementation is identical to the Hailstone-S HDL implementation, except it selects the even or odd result using explicit Boolean AND-OR masking instead of a multiplexer inferred from a conditional statement.

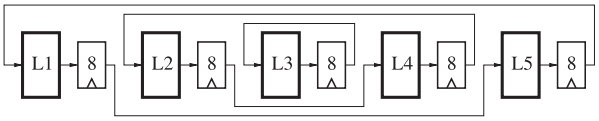
```
1 for i = 1 to num_samples
2    dotp = 0;
3    for k = 1 to num_coeff
4       dotp = dotp + (coeff[k] * in[i+k]); // Apply all coefficients to each input
5    out[i] = dotp;
```

Listing 2. Conventional FIR Filter MXP Pseudo-Code.

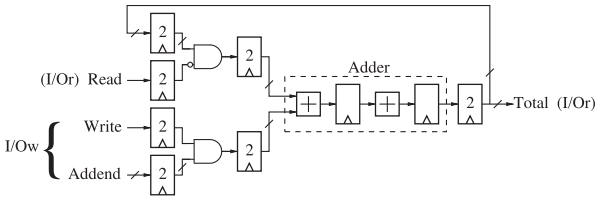
```
1 for i = 1 to num_samples
2    out[i] = 0;
3 for k = 1 to num_coeff
4    const = coeff[k];
5    for i = 1 to num_samples
6       out[i] = out[i] + (const * in[i+k]); // Apply each coefficient to all inputs
```

Listing 3. Transposed FIR Filter MXP Pseudo-Code.

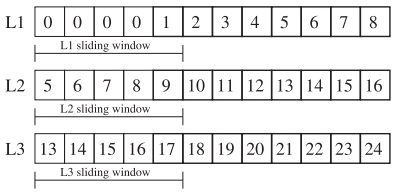
FIR. Computes an eight-tap Finite Impulse Response (FIR) filter over an input array of approximately 100 wordwide integer elements (depending on platform) into a similar and separate output array. For best performance, the HDL implementation uses a separate register buffer to hold past input values for convolution, while the MXP, Octavo, and LegUp implementations use a pointer into a sliding window over input memory. The HDL implementation separates input and output data into two distinct BRAMs to simplify implementation, performs all 8 multiplications in parallel, and sums the products via a pipelined tree of adders. The MXP FIR filter implementation



(a) Octavo-L5 Lanes with 8-stage Array Reversal Channels



(b) Pipelined Per-Lane Accumulator, with 2-stage delays



(c) Data and extra headers for a 5-tap FIR filter on an Octavo-L3, over 24 input data points.

Fig. 3. Accelerators attached to Octavo.

depends on the number of lanes and filter taps. The final Accumulator (upper right in Figure 1) can only validly sum up to the number of taps, despite receiving values from all vector lanes. When the number of vector lanes exceeds the number of filter taps, we can transpose the FIR algorithm to support vector parallelism at the cost of using a separate vector summation instead of the Accumulator.

For an eight-tap filter, for 16 lanes and fewer, MXP concurrently performs each of the coefficient multiplications (one per vector lane) and then uses the final Accumulator to sum the products into a scalar result, implementing an efficient pipelined Multiply-Accumulate operation (Listing 2). For more than 16 lanes, MXP instead successively multiplies each coefficient over an entire input vector and sums the products into an accumulation vector (Listing 3). This transposed FIR algorithm applies the coefficients over multiple vector multiplications, followed by a vector sum, but calculates multiple convolutions at once. Thus, the parallelism scales up to the entire input data set length,[4] not just the number of filter coefficients, at the price of some extra overhead. In this case, the parallelism exceeds the overhead past 16 lanes, so we use the transposed FIR at 32 lanes.

In contrast, Octavo parallelizes an n-tap FIR filter by dividing the input and output data across all SIMD lanes and having each lane apply the FIR filter to each subset. We accelerate each convolution by providing each SIMD lane with an Accumulator to allow chaining multiplications and sums (Section 2.2). To avoid introducing gaps in the output data, we prepend a header of the last $n - 1$ input data items from the previous lane, or zeros at the first lane, to allow the sliding window to immediately contain the necessary data to compute the next consecutive output value. Figure 3(c) illustrates the headers and data for a hypothetical Octavo-L3 implementing a five-tap FIR filter over 24 input data points (in the bold boxes).

4. ADAPTING OCTAVO TO SPECIFIC BENCHMARKS

In this section, we discuss how to adapt Octavo to particular benchmarks by augmenting it with small, custom hardware accelerators without sacrificing programmability. Contrary to MXP and other soft-processors [Jones et al. 2005; Saghir et al. 2006; Hameed et al. 2010; Severance et al. 2014] that place their custom functional units

[4]MXP's Direct Memory Access (DMA) controller would transfer data as needed for continuous, un-windowed filtering.

inside the ALU or directly in the datapath pipeline, Octavo attaches them as simple memory-mapped I/O devices, directly addressable by instruction operands (Section 2). We added Octavo accelerators that implement functionality present in MXP but absent in Octavo. These accelerators enable new data movements (for Reverse-4) or pull some operations out of a loop and execute them in parallel (for FIR). Octavo implicitly supports table lookups, required by the sequential Hailstone-A benchmark, via its per-lane data memories. Conversely, the designers of MXP added Custom Vector Instructions in the datapath to support per-vector-lane table lookups (Figure 1). To approximate MXP's vector accumulator and alignment networks on Octavo, we attached an Accumulator to the I/O ports of each datapath to provide a chained Multiply-Accumulate operation for the parallel FIR benchmark. We also similarly inter-connected the datapaths with pipelined Array Reversal Channels to support the parallel Reverse-4 benchmark. We describe these accelerators below.

4.1. Array Reversal Channel

When reversing an array, the element with the lowest index is exchanged with the element with the highest index, the next-lowest element is exchanged with the next-highest element, and so on, towards the center of the array. We can distribute an array across communicating vector or SIMD lanes to parallelize this process. MXP connects its vector lanes via read/write alignment networks that can offset a vector across lanes as desired. In contrast, Octavo's SIMD lanes have no interconnections and could not execute an equivalent parallel array reversal benchmark. Therefore, we implemented an accelerator to specifically support the required communication pattern.

Figure 3(a) shows the eight-stage channel connections for parallel array reversal across the five lanes of an Octavo-L5: If we distribute an array across all five lanes, then the first array segment exchanges and reverses itself with the last, and so on, in decreasing circles until the centre lane L3 reverses its own segment. The number of channels scales with the number of SIMD lanes, thus the number of steps to reverse an array remains constant if the number of lanes scales with the total size of the array. We need 16 pipeline stages total to contain all the writes from Octavo's 8 threads before we begin reading them back (in reverse order). Octavo's pipelined datapath provides the first 8 stages as memory and I/O writes happen 8 cycles after reads, leaving us to add 8 stages between lanes to pipeline the channels.

4.2. Accumulator

For a FIR filter, MXP uses an Accumulator to sum the filter coefficient products from each vector lane to a final scalar value, resulting in a pipelined Multiply-Accumulate (MAC) operation. Octavo cannot simply copy this cross-lane vector reduction, as each SIMD lane runs the same code in lockstep and would need complicated memory-mapping tricks to force storing the final scalar sum into one SIMD lane only. Duplicating the final sum into all lanes would waste too much memory. Instead, we enable each thread to chain multiplications and accumulation sums together, much like MXP does, by attaching an Accumulator, sufficiently pipelined to support all threads, to the I/O ports of the datapath.

Figure 3(b) shows the Accumulator implementation. In the centre, we have a copy of Octavo's Adder taken from its ALU, which takes two pipeline stages. We add additional pipeline registers, in groups of two for convenience, to add up to eight pipeline stages. After writing to the Accumulator, the new Total sum shows up at the output eight cycles later and circulates back into the Accumulator. Thus, from a thread's perspective, the new Total sum becomes available one instruction after the previous write to the Accumulator. Other operations easily fill this gap to avoid the Read-After-Write (RAW) hazard.

The Accumulator attaches to Octavo via one I/O write port (I/Ow) and one I/O read port (I/Or). To add a new value to the Total sum, a thread must write it (the Addend) to I/Ow, which also raises the port's write enable line (Write). When not asserted, the Write line zeroes-out the Addend, effectively adding zero to the Total sum. To read out the Total sum, a thread must read from I/Or, which also raises the port's read enable line (Read). When asserted, the Read line zeroes-out the circulating Total sum for that thread, restarting the accumulation process. Calculating one output of an n-tap FIR filter now requires only $n + 1$ steps, n multiplications followed by 1 Accumulator read, instead of $2n - 1$ with separate Multiply and Accumulate instructions.

Following Octavo's multi-threaded programming model, as we replicate the datapath for SIMD parallelism, each new SIMD lane also receives an Accumulator. Thus, although Octavo cannot multiply all the filter coefficients and sum them together in a single pipelined n-tap MAC operation like MXP, we can efficiently perform chained MACs for any number of taps and divide the work across any number of SIMD lanes.

5. BENCHMARK RESULTS

For each benchmark, we measure the following values and calculate their ratio relative to those of a scalar Octavo core (Octavo-L1) as the baseline for all speed and area comparisons. Not all implementations are feasible in all cases.

—The number of cycles per unit of work (i.e., computing one result) and the cycle count speedup. We describe what constitutes a unit of work with the results of each benchmark. Typically, the computation of one output value defines one unit of work.
—The maximum operating frequency (F_{max}) of the implementation. We choose the instance with the *highest* F_{max} of 10 random seed P&R runs to compare the best possible case of each implementation.
—The wall-clock time per unit of work (in nanoseconds) and their speedups.
—The area of the implementation, measured as the count of *equivalent Adaptive Logic Modules* (eALMs) in use, which include the equivalent silicon area of hard blocks such as M9K BRAMs (28.7 ALMs each) and DSP blocks (29.75 ALMs each) as reported by Wong et al. [2014, 2011]. We also show the area shrink (or growth, to keep the ratios simple to read) of each implementation.
—The Area-Delay product as eALMs·ns (time per unit of work), along with shrink or growth, which indicates relative implementation efficiency. For parallel benchmarks, a relatively constant Area-Delay as the parallelism and working set size increase indicates a scalable implementation, whose amount of work produced per unit time increases linearly with the area.

5.1. Sequential Benchmarks

For the sequential benchmarks, we compare single HDL and LegUp instances, and Octavo and MXP instances with 1 or 2 lanes, as instruction-level parallelism allows, showing how their F_{max} and cycle counts per result compare. Because MXP's vector lanes do not support branching, some sequential benchmarks must default to running on the controlling Nios II/f processor (denoted as "MXP-Nios"). Note that the F_{max} of Nios matches that of the MXP vector lanes and could run somewhat faster (approximately 240MHz) by itself [Altera 2014].

5.1.1. Reverse-3. Table I shows the results of the Reverse-3 benchmark on a scalar Octavo instance (Octavo-L1), a single-lane MXP instance (MXP-V1), and the LegUp and HDL implementations. We define the unit of work as swapping two array elements.

Octavo executes Reverse-3 sequentially, executing a six-instruction loop 64 times to reverse a 128-element array. Of the six instructions, three perform the swap, and

Table I. 3-Step Array Reverse (Reverse-3) Measurements

	Cycles/Unit		F_{max}	Time/Unit		Area		Area-Delay	
Reverse-3	Cycles	Speedup (Slower)	MHz	ns	Speedup (Slower)	eALMs	Shrink (Grow)	eALMs·ns	Shrink (Grow)
Octavo-L1	3.07	—	525	5.85	—	2203	—	12,888	—
MXP-V1	1.12	2.74	221	5.07	1.15	4429	(2.01)	22,455	(1.74)
LegUp	1.06	2.90	468	2.26	2.59	90	24.5	203.4	63.4
HDL	1.03	2.98	612	1.68	3.48	177.7	12.4	299	43.1

Table II. Sequential Hailstone (Hailstone-S) Measurements

	Cycles/Unit		F_{max}	Time/Unit		Area		Area-Delay	
Hailstone-S	Cycles	Speedup (Slower)	MHz	ns	Speedup (Slower)	eALMs	Shrink (Grow)	eALMs·ns	Shrink (Grow)
Octavo-L1	4.55	—	525	8.67	—	2203	—	19,100	—
MXP-Nios	7.42	(1.63)	221	33.6	(3.88)	4429	(2.01)	148,814	(7.79)
LegUp	1.05	4.33	331	3.17	2.74	2889	(1.31)	9,158	2.09
HDL	1.04	4.38	600	1.73	5.01	126.7	17.4	219	87.2

the other 3 deal with a decrementing pointer (not supported by the AOM) and the loop counter (not supported by the BTM). The loop branch overlaps with the last ALU instruction.

In contrast, MXP automates all the addressing and looping, swapping one array item every cycle (plus some setup overhead), but its lower F_{max} counters the lower cycle count, resulting in only a 15% speedup over Octavo. With this moderate speedup, MXP's 2.01× larger area also contribute to a 74% increase in Area-Delay product.

The LegUp and HDL implementations approach the ideal of 1 cycle per swap, improve on Area and Area-Delay, but perform less than 3–4× faster than Octavo. The LegUp implementation performs at 0.74× the rate of the HDL implementation but uses only half as much Area due to lack of pipelining, yielding a lower net Area-Delay product.

5.1.2. Hailstone-S. Table II shows the results of the Hailstone-S benchmark. We define the unit of work as computing one step of a Hailstone sequence. Octavo-L1 uses three overlapping and cancelling branches to compact the loop counter decrement and both even/odd branches into five instructions. MXP's vector lanes cannot compute Hailstone-S, since they do not support branches, and thus we executed the benchmark on the Nios II/f controlling processor only, which suffers from a larger cycle count and lower F_{max}.

Both LegUp and HDL compute both even and odd branches in parallel and then multiplex the desired output. Surprisingly, the Quartus CAD tool failed to infer a Block RAM (BRAM) during synthesis of the LegUp implementation and implemented the entire benchmark as registers and logic gates, ballooning its area and likely limiting its performance and Area-Delay improvements over Octavo-L1. In contrast, the HDL uses a single M9K BRAM, reducing its area and maximizing its speed but still only achieving a 5× speedup over software running on Octavo.

5.1.3. Hailstone-A. Table III shows the results of the Hailstone-A benchmark. As with other Hailstone benchmarks, we define the unit of work as computing one step in a Hailstone sequence. Both Octavo and MXP distribute over their entire pipeline the computation of each of the eight independent "slices" of the entire Hailstone sequence. MXP requires only six instructions, but a few hazards cause them to take seven cycles to execute on MXP-V1. On MXP-V2, the second lane can execute some non-dependent instructions in parallel, reducing the total cycle count back to 6, improving performance but not improving the Area-Delay. Larger MXP vector lane counts (not shown) only worsen the results due to longer and emptier pipelines (+1 stage at V4 and V16). Note

Table III. Accelerated Hailstone (Hailstone-A) Measurements

	Cycles/Unit		F_{max}	Time/Unit		Area		Area-Delay	
Hailstone-A	Cycles	Speedup (Slower)	MHz	ns	Speedup (Slower)	eALMs	Shrink (Grow)	eALMs·ns	Shrink (Grow)
Octavo-L1	7.04	—	525	13.4	—	2203	—	29,520	—
MXP-V1	7.02	1.00	221	31.8	(2.37)	4429	(2.01)	140,842	(4.77)
MXP-V2	6.01	1.17	239	25.1	(1.87)	5533	(2.51)	138,878	(4.70)
MXP-Nios	22.3	(3.17)	221	101	(7.54)	4429	(2.01)	447,329	(15.2)
LegUp	1.64	4.29	177	9.27	1.45	896.2	2.46	8308	3.55
HDL	1.04	6.77	510	2.04	6.57	312.6	7.05	638	46.3

Table IV. Structured FSM (FSM-S) Measurements

	Cycles/Unit		F_{max}	Time/Unit		Area		Area-Delay	
FSM-S	Cycles	Speedup (Slower)	MHz	ns	Speedup (Slower)	eALMs	Shrink (Grow)	eALMs·ns	Shrink (Grow)
Octavo-L1	92.0	—	65.6	1402	—	2203	—	3,088,606	—
MXP-Nios	141	(1.53)	221	638	2.20	4429	(2.01)	2,825,702	1.09
LegUp	27.3	3.37	401	68.2	20.6	150	14.7	10,230	302
HDL	3.96	23.2	444	8.92	157	65.7	33.5	586	5271
HDL (pipe'd)	5.04	18.3	530	9.51	147	60.7	36.3	577	5353

Table V. "Assembly" FSM (FSM-A) Measurements

	Cycles/Unit		F_{max}	Time/Unit		Area		Area-Delay	
FSM-A	Cycles	Speedup (Slower)	MHz	ns	Speedup (Slower)	eALMs	Shrink (Grow)	eALMs·ns	Shrink (Grow)
Octavo-L1	28.9	—	65.6	440.5	—	2203	—	970,422	—
MXP-Nios	55.3	(1.91)	221	250	1.76	4429	(2.01)	1,107,250	(1.14)
LegUp	18.3	1.58	366	50.0	8.81	219	10.06	10,950	88.62

that MXP requires custom support for vector lane table lookups, while Octavo simply stores the tables in local lane memory. Both store a copy of the lookup tables per lane. We also included the Nios II/f results that highlight the positive impact of having all data in on-chip scratchpads. Most of the 22.3 cycles/unit are spent doing table lookups from (cached) main memory.

LegUp appears to perform all eight slices concurrently, as it requires eight DSP Blocks, suggesting eight parallel multiplications, and six M9K BRAMs, suggesting concurrent table lookups for each of the eight "slices." However, LegUp does not seem to fully pipeline the process, resulting in a relatively low F_{max}, albeit with a 45% net speedup. In contrast, the HDL implementation fully pipelines the eight sets of calculations, using only three M9K BRAMs (two for tables and one for seeds) and re-using Octavo's high-speed Multiplier (two DSP Blocks). The HDL implementation cannot reach maximum speed, since we are limited to eight pipeline stages (one for each sequence "slice"), and the final adder becomes a critical path. Adding a ninth pipeline stage could conceivably increase F_{max} to 600MHz (+18%), but at the cost of a pipeline bubble every ninth cycle, subtracting 11 percentage points. Again, a software implementation on Octavo approaches within an order of magnitude the performance of custom HDL (6.57×), albeit at a similar increase in area (7.05×).

5.1.4. FMS-S and FSM-A. Tables IV and V show the results for both the "Structured" and "Assembly" versions of the floating-point recognizer FSM benchmark. We define the unit of work as recognizing a number as either valid (`ACCEPT` state) or invalid

(REJECT state), with the cycle count for each unit taken as the average cycle count for all 26 numbers.

The FSM-S implementation uses conventionally structured code but at the cost of adding a layer of interpretation by storing the FSM state in a variable and updating it using nested if-statements inside an outer loop. In contrast, the FSM-A implementation encodes the state in the Program Counter by immediately jumping to code that implements a given state without having to go through a global outer loop. MXP's vector lanes cannot implement FSMs, since they do not support branching, so we default to the Nios II/f controlling processor. Since we cannot distribute the work of the FSM over multiple Octavo threads,[5] we list the effective F_{max} of a single Octavo-L1 thread: 65.6MHz. The other seven threads remain idle.

We can gauge the impact of FSM software implementation by comparing across both tables: On Octavo-L1, FSM-S requires 3.18× more cycles per result than FSM-A, since the additional work of handling of a state variable in FSM-S prevented executing the setup of the next branch in parallel with the current branch, increasing cycle count. For LegUp, although a user would normally write structured code, going from FSM-S to FSM-A reduced F_{max} by 8.7% but also reduced cycle count by 33%. For HDL, we cannot implement FSM-A as the HDL implementation does not have a Program Counter. Pipelining the HDL implementation of FSM-S had little benefit: The cycle count increases to offset the higher F_{max}, without significantly changing the area.

Since we can only use a single Octavo thread, it performs extremely poorly compared to LegUp and HDL hardware implementations and still about 2× slower than Nios II/f. However, with seven threads remaining, there remains a lot of margin for additional work "for free" on Octavo.

5.2. Parallel Benchmarks

For the parallel benchmarks, we compare single HDL and LegUp instances, and Octavo and MXP instances with 1 to 32 lanes, showing how their F_{max} and cycle counts per result scale. Where we extrapolate parallel HDL and LegUp implementations, we assume multiple individual instances running at the same F_{max}. Octavo's F_{max} degrades gradually and monotonically, due to partitioning [LaForest and Steffan 2013], as the number of lanes increases. In contrast, MXP's F_{max} first increases but then drops as the routing in the widening double-pumped vector scratchpad becomes the critical path.

5.2.1. Array Increment. Table VI shows the results of the Increment benchmark, representing the simplest parallel case and showing the best-case speedup as MXP and Octavo scale. We define the unit of work as incrementing one array location. MXP achieves near-ideal performance and scaling (1 increment per cycle per lane) while Octavo, which lacks a hardware loop counter, must spend more than twice as many cycles to do the same. However, Octavo's greater F_{max} recoups most of the difference and approaches the performance of the LegUp and HDL implementations.

Parallel Area-Delay Scaling. The Increment benchmark also uncovers a pattern we see throughout the parallel benchmarks: Octavo's Area-Delay remains relatively constant as the number of lanes (and thus the parallelism and working set size) increases, until F_{max} begins to decrease faster at 32 lanes. In contrast, MXP's Area-Delay *reduces* as we scale up, improving efficiency until it also falters on decreasing F_{max} at 32 lanes. This difference in Area-Delay scaling highlights MXP's emphasis on effective large-scale parallelism with a uniform vector programming model, while Octavo focuses on efficient small-scale parallelism and its composition to address larger problems at the cost of a lower-level multi-threaded programming model.

[5]We do not consider transformations which convert a FSM into multiple concurrent FSMs.

Table VI. Array Increment (Increment) Measurements

	Cycles/Unit		F_{max}	Time/Unit		Area		Area-Delay	
Increment	Cycles	Speedup (Slower)	MHz	ns	Speedup (Slower)	eALMs	Shrink (Grow)	eALMs·ns	Shrink (Grow)
Octavo-L1	2.35	—	525	4.48	—	2203	—	9,869	—
Octavo-L2	1.17	2.01	514	2.28	1.96	2976	(1.35)	6,785	1.46
Octavo-L4	0.587	4.00	474	1.24	3.61	6004	(2.73)	7,445	1.33
Octavo-L8	0.293	8.02	455	0.644	6.96	11,133	(5.05)	7,170	1.38
Octavo-L16	0.147	16.0	420	0.350	12.8	21,288	(9.66)	7,451	1.33
Octavo-L32	0.0734	32.0	370	0.198	22.6	42,015	(19.1)	8,319	1.19
MXP-V1	1.02	2.30	221	4.62	0.970	4429	(2.01)	20,241	(2.05)
MXP-V2	0.512	4.59	239	2.14	2.09	5533	(2.51)	11,841	(1.20)
MXP-V4	0.256	9.18	242	1.06	4.23	9740	(4.42)	10,324	(1.05)
MXP-V8	0.129	18.2	206	0.626	7.16	14,813	(6.73)	9,273	1.06
MXP-V16	0.0640	36.7	206	0.311	14.4	28,229	(12.8)	8,779	1.12
MXP-V32	0.0320	73.4	168	0.190	23.6	55,310	(25.1)	10,509	(1.07)
LegUp	1.12	2.1	335	3.34	1.34	135	16.32	450.9	21.9
HDL	1.03	2.28	600	1.72	2.60	157.7	13.97	271.2	36.4

Table VII. Non-Branching Sequential Hailstone (Hailstone-N) Measurements

	Cycles/Unit		F_{max}	Time/Unit		Area		Area-Delay	
Hailstone-N	Cycles	Speedup (Slower)	MHz	ns	Speedup (Slower)	eALMs	Shrink (Grow)	eALMs·ns	Shrink (Grow)
Octavo-L1	11.0	—	525	21.0	—	2203	—	46,263	—
Octavo-L2	5.49	2.00	514	10.7	1.96	2976	(1.35)	31,843	1.45
Octavo-L4	2.75	4.00	474	5.80	3.62	6004	(2.73)	34,823	1.33
Octavo-L8	1.37	8.03	455	3.01	6.98	11,133	(5.05)	33,510	1.38
Octavo-L16	0.687	16.0	420	1.64	12.8	21,288	(9.66)	34,912	1.33
Octavo-L32	0.343	32.1	370	0.927	22.7	42,015	(19.1)	38,948	1.19
MXP-V1	6.05	1.82	221	27.4	(1.30)	4429	(2.01)	121,355	(2.62)
MXP-V2	3.04	3.62	239	12.7	1.65	5533	(2.51)	70,269	(1.52)
MXP-V4	1.52	7.24	242	6.28	3.43	9740	(4.42)	61,167	(1.32)
MXP-V8	0.762	14.4	206	3.70	5.68	14,813	(6.73)	54,808	(1.19)
MXP-V16	0.380	28.9	206	1.84	11.4	28,229	(12.8)	51,941	(1.12)
MXP-V32	0.190	57.9	168	1.13	18.6	55,310	(25.1)	62,500	(1.35)
LegUp	2.06	5.34	338	6.09	3.45	166.4	13.2	1013	45.7
HDL	1.04	10.6	600	1.73	12.1	131.7	16.7	228	203

5.2.2. Hailstone-N. Table VII shows the results for the Hailstone-N benchmark. We define one unit of work as computing one step in a Hailstone sequence. All versions compute both even and odd branches and then select the desired result based on the parity of the seed (i.e., even or odd). MXP selects using conditional moves, while all other implementations use Boolean operations to mask and merge the results. As in the Array Increment benchmark, MXP, and Octavo scale effectively, with Octavo's slightly less-than-doubled cycle count and more-than-doubled F_{max} giving it an advantage over MXP. Both MXP and Octavo require 16 lanes to approximately match the performance of HDL hardware, and 4 lanes to match that of LegUp, at a considerable area cost.

5.2.3. Reverse-4. Table VIII shows the results of the Reverse-4 benchmark. We define the unit of work as swapping two array elements. Octavo-L1 requires precisely 0.5 more cycles/unit than the Reverse-3 benchmark, accounting for the extra fourth step in the swap operation, operating over the exact same amount of data. As we increase

Table VIII. Four-Step Array Reverse (Reverse-4) Measurements

	Cycles/Unit		F_{max}	Time/Unit		Area		Area-Delay	
Reverse-4	Cycles	Speedup (Slower)	MHz	ns	Speedup (Slower)	eALMs	Shrink (Grow)	eALMs·ns	Shrink (Grow)
Octavo-L1	3.57	—	525	6.80	—	2203	—	14,980	—
Octavo-L2	1.79	1.99	514	3.48	1.95	2976	(1.35)	10,356	1.45
Octavo-L4	0.893	4.00	474	1.88	3.62	6004	(2.73)	11,287	1.33
Octavo-L8	0.446	8.01	455	0.980	6.94	11,133	(5.05)	10,910	1.37
Octavo-L16	0.223	16.0	420	0.531	12.8	21,288	(9.66)	11,304	1.33
Octavo-L32	0.112	31.9	370	0.303	22.4	42,015	(19.1)	12,731	1.18
MXP-V1	1.78	2.01	221	8.05	(1.18)	4429	(2.01)	35,653	(2.38)
MXP-V2	1.81	1.97	239	7.57	(1.11)	5533	(2.51)	41,885	(2.80)
MXP-V4	1.26	2.83	242	5.21	1.31	9740	(4.42)	50,745	(3.39)
MXP-V8	0.854	4.18	206	4.15	1.64	14,813	(6.73)	61,474	(4.10)
MXP-V16	0.623	5.73	206	3.02	2.25	28,229	(12.8)	85,252	(5.69)
MXP-V32	0.345	10.3	168	2.05	3.32	55,310	(25.1)	113,386	(7.57)
LegUp	0.550	6.49	483	1.14	5.96	896	2.459	1,021	14.7
HDL	0.520	6.87	607	0.857	7.93	157.4	14.00	135	111

the number of lanes, we also increase the array size to keep the amount of data per lane constant. Reverse-4 on Octavo scales as expected, with maximal speedup in all cases.

In contrast, MXP's "Reverse-Recursive" equivalent has a higher setup cost (no significant speedup until MXP-V4) and suffers from the increasing pipeline depth (+1 at V4 and V16) preventing the expected halving of cycles/unit, greatly limiting its total speedup (6.77× slower than Octavo at 32 lanes). We can also see the poor scaling of the recursive algorithm from the *increasing* Area-Delay product as the benchmark scales up.

Both the LegUp and HDL implementations uncover the implicit parallelism in the generalized four-step swap operation, moving two items per cycle instead of one. Again, Quartus fails to infer BRAMs in the LegUp implementation and instead implements the entire storage as registers, increasing its area and likely slowing down its operation.

5.2.4. FIR. Table IX shows the results of the eight-tap FIR filter benchmark. We define the unit of work as computing one filtered output value. For 1 to 8 lanes, MXP and Octavo scale as expected, halving the number of cycles/unit when doubling the lane count and converging towards 1 cycle/unit with 8 lanes. For a single lane, MXP-V1 requires 8 cycles for 8 pipelined MAC operations (Listing 2), while Octavo-L1 requires 12 as it must also manually re-initialize the post-incrementing pointer for the sliding window, decrement the loop counter, and read out the per-lane Accumulator values to write into the output arrays.

MXP-V16 has the same performance as MXP-V8, since it still uses the same Accumulator. MXP-32 switches to the transposed FIR algorithm (Listing 3) as the number of coefficients (8) limits any further increase in parallelism, and the transposed parallelism exceeds its extra overhead, improving performance. On the other hand, Octavo continues scaling without interruption.

The LegUp implementation uses the expected eight DSP blocks to multiply all coefficients in parallel but unexpectedly required five BRAMs rather than the expected two (input and output). LegUp also fails to pipeline the multipliers and accumulators, resulting in a low F_{max}. Nonetheless, its lower cycles/unit result in identical performance to an entire Octavo-L1, at less than one quarter the area. In contrast, the HDL implementation uses two BRAMs and re-uses Octavo's double-pipeline Multiplier and

Table IX. Eight-Tap FIR Measurements

	Cycles/Unit		F_{max}	Time/Unit		Area		Area-Delay	
FIR	Cycles	Speedup (Slower)	MHz	ns	Speedup (Slower)	eALMs	Shrink (Grow)	eALMs·ns	Shrink (Grow)
Octavo-L1	12.1	—	525	23.0	—	2203	—	50,669	—
Octavo-L2	6.05	2.00	514	11.8	1.95	2976	(1.35)	35,117	1.44
Octavo-L4	3.03	3.99	474	6.39	3.60	6004	(2.73)	38,366	1.32
Octavo-L8	1.52	7.96	455	3.34	6.89	11,133	(5.05)	37,184	1.37
Octavo-L16	0.756	16.0	420	1.80	12.8	21,288	(9.66)	38,318	1.32
Octavo-L32	0.378	32.0	370	1.02	22.5	42,015	(19.1)	42,855	1.18
MXP-V1	8.02	1.51	221	36.3	(1.58)	4429	(2.01)	160,773	(3.17)
MXP-V2	4.02	3.01	239	16.8	1.37	5533	(2.51)	92,954	(1.84)
MXP-V4	2.01	6.02	242	8.31	2.77	9740	(4.42)	80,939	(1.60)
MXP-V8	1.01	12.0	206	4.90	4.69	14,813	(6.73)	72,584	(1.43)
MXP-V16	1.01	12.0	206	4.90	4.69	28,229	(12.8)	138,322	(2.73)
MXP-V32	0.537	22.5	168	3.20	7.19	55,310	(25.1)	176,992	(3.49)
LegUp	4.10	2.95	180	22.8	1.01	513	4.29	11,696	4.33
HDL	1.11	10.9	564	1.97	11.7	1164	1.89	1292	39.2

pipelined Adder to implement a fully parallel 10-stage pipelined FIR filter at twice the area cost of LegUp but over 11× the performance.

6. OCTAVO AND MXP MICRO-ARCHITECTURAL COMPARISON

From the benchmark results, we can make some micro-architectural comparisons between the MXP and Octavo soft-processor overlays and identify opportunities for improvement.

6.1. MXP

MXP's vector processing provides a scalable and portable programming model. However, MXP sometimes suffers from poor initial scaling due to pipeline hazards with short vectors (re: FIR benchmark) or more complex setup (re: Reverse-4 benchmark), reducing overall throughput and reducing later performance gains under larger vector parallelism. However, MXP's address generators and vector loop counters significantly improve its per-cycle performance relative to Octavo. Additionally, MXP can efficiently select one of two results with a single conditional move (`CMOV`) instruction based on an implicit or pre-computed flag in one or two cycles total.

We can see a difference in small-scale and large-scale parallelism focus by comparing the scaling of the Area-Delay product for Octavo and MXP. Notwithstanding significant drops in F_{max} at 32 lanes, Octavo's Area-Delay tends to remain constant as the number of its SIMD lanes increases, while MXP's Area-Delay starts higher but decreases as the number of its vector lanes increases.

6.2. Octavo

Overall, Octavo benefits most from its raw high F_{max} enabled by constraining its threads to a strict round-robin order, which eliminates the need to forward results across pipeline stage with costly backwards paths and multiplexers. Octavo's multi-threaded programming model works efficiently at a small scale and remains efficient on larger problems.

In several benchmarks (Reverse-3, Hailstone-A, Increment, Hailstone-N), Octavo approaches or exceeds MXP's performance only because Octavo's high F_{max} offsets its greater cycle count. These extra cycles come from manually decrementing counters and pointers due to lack of support in Octavo's Branch Trigger Module (BTM) and Address

Offset Module (AOM). The BTM does not yet include a branch condition based on a counter, and the AOM currently applies a single-bit post-increment of 0 or +1.

Similarly, without a `CMOV` instruction, selecting one of two results without branching requires computing a wordwide mask (all-zeros or all-ones) to cancel out terms of a Boolean expression of both results, taking a total of 5 instructions. These 5 cycles represent 45% of the 11 cycles/result of the Octavo Hailstone-N benchmark. This overhead ratio remains constant with SIMD scaling.

7. FPGA OVERLAY DESIGN THEMES

From our MXP and Octavo benchmarking, we can draw two general design themes for soft-processor overlays: ease of modification and separation of flow-control, addressing, and arithmetic.

To increase the performance of soft-processors for given types of tasks, we only need to implement in hardware simpler, control-free custom operation pipelines representing costly or repetitive computations. We can quickly check the initial correctness and performance of these simple pipelines and complete their verification by using the benchmark software itself. This hybrid design process reduces both hardware and software design efforts and suggests that future soft-processor overlays be augmented via HLS-generated accelerators, which is essentially LegUp's approach.

The high performance of HLS/HDL benchmark implementations comes from parallelizing many computations executed sequentially as software on soft-processors. These computations include array indexing, loop counting, calculating branch conditions, and acting on them. Both MXP and Octavo have special hardware modules for these computations that are essential to approaching hardware-like performance, as they free the ALU to do useful work. These modules do come at a cost in generality and more complicated software support.

When fully utilized, both MXP and Octavo execute the benchmarks within an order of magnitude of the performance of HLS and HDL solutions but with a corresponding order of magnitude area increase. This area increase affects energy efficiency, depending on the particular soft-processor architecture, and is another dimension along which soft-processors overlays should be compared in the future.

APPENDIX

A. THE OCTAVO SOFT-PROCESSOR MICRO-ARCHITECTURE

In this appendix, we detail an updated Octavo soft-processor [LaForest and Steffan 2012] to place previously published work in context [LaForest et al. 2014], fix some minor errors, and to describe previously unpublished improvements to its address space and instruction predication.

Figure 2 (Section 2) shows Octavo's overall architecture split into 10 pipeline stages (0–9). In stage 0, at the far left, the Instruction Memory (I) issues 36-bit instructions, composed of a 4-bit opcode (OP), a 12-bit destination operand (D), and two 10-bit source operands (A/B). In parallel, the BTM computes branches, the AOM generates the final operands (D′/A′/B′), and the I/O Predication Module (PRD) checks the read/write Empty/Full bits (E/Fr and E/Fw) of the I/O ports and asserts the I/O Ready (IOR) signal if all ports used by the instruction are ready. If IOR is not set, then the instruction is annulled into a no-op and will re-issue later.

Starting at stage 4, in the middle, the A′/B′ operands read into the 36-bit word-wide Data Memories (A/B), which also contain the I/O ports, and feed their data to the ALU, which then writes its result (R) to the destination address (D′) in all memories and modules. The ALU computes basic integer functions: addition/subtraction, full-word

multiplication, and bitwise Boolean operators. Replicating this datapath enables SIMD operation.

At the end of the control path, at top-right, the Controller (CTL) receives the BTM's Branch Destination (BD′) and Jump decision (J) and either issues the appropriate next PC value or, if IOR is not set, re-issues the same PC value again. It takes two cycles to read A/B (RD0 and RD1), four cycles for the ALU to produce a result, and two cycles to write the result back to A/B (WR0 and WR1). Thus, to avoid pipeline hazards, we issue instructions from eight separate hardware threads in strict round-robin order, without variation. Within each thread, each instruction completes before the next in the same thread. However, writes to the Instruction Memory and BTM/AOM take effect after one thread instruction of latency, since these write destinations are more than eight stages away from the ALU output and do not complete before the next instruction in that thread issues.

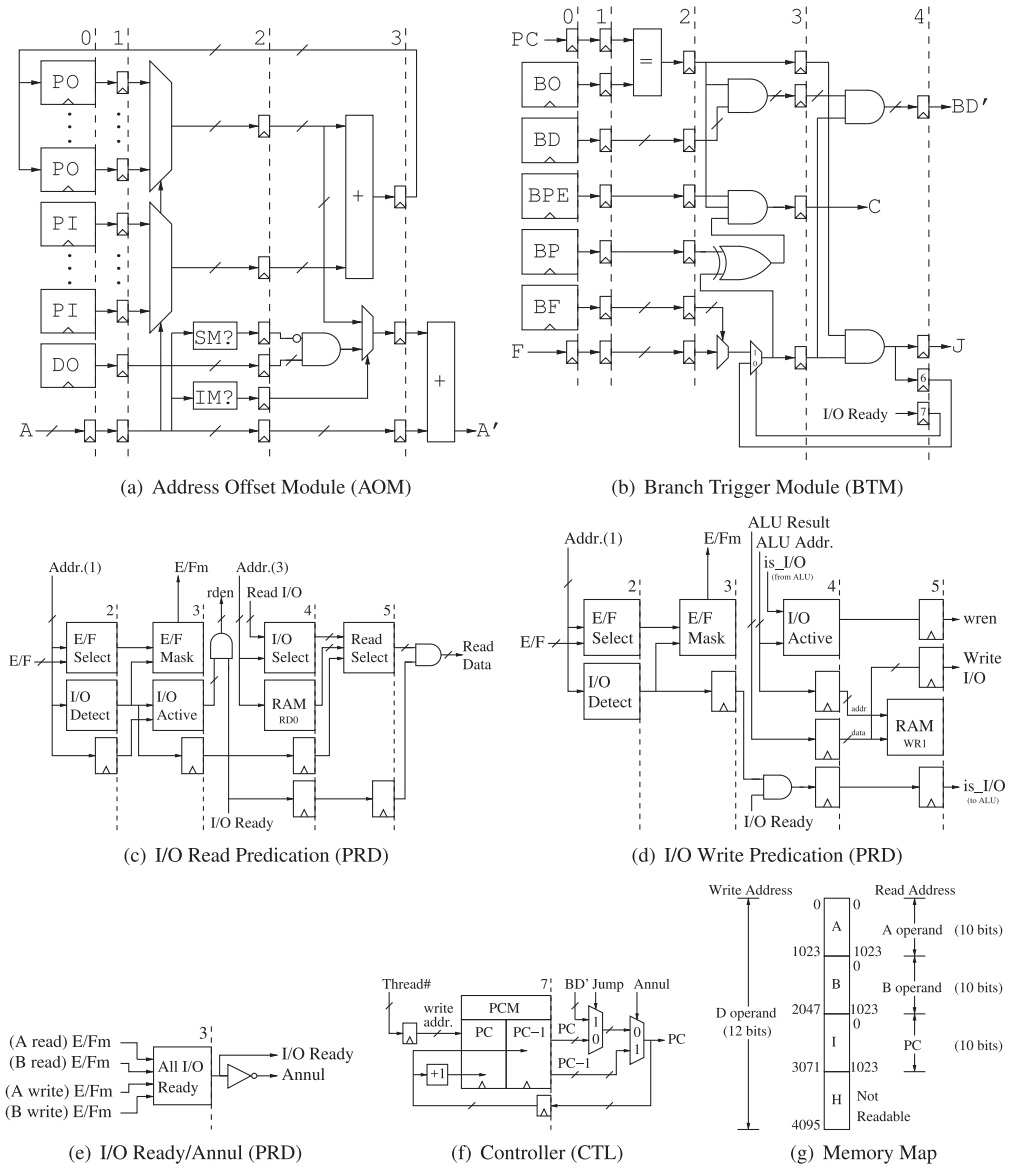
AOM. Figure 4(a) shows the AOM, which automates addressing calculations. We need three instances of the AOM, one for each instruction operand (D/A/B), as any one may access a pointer or other special memory at any time. The AOM adds a selected offset to the address (A) contained in its associated instruction operand. The offset added may be a constant Default Offset (DO), a Programmed Offset (PO) with automatic Programmed Increment (PI), or zero. The ability to add a constant offset to addresses is useful for threads that share program code but read/write from different regions of memory. Post-incrementing offsets are useful for operations such as walking through an array. Zero offsets are useful for addresses that are shared across all threads, such as addresses of memory-mapped hardware.

The Shared Memory (SM?) and Indirect Memory (IM?) logic modules decode part of the processor's address space, set at design time, defining fixed ranges of memory that either act as absolutely addressed shared resources (i.e., I/O ports) or act as pointers. Accessing shared memory gates the DO to zero, while accessing indirect memory overrides the DO to use one of the PO, selected by a subset of the address (A). Since pointers exist at fixed addresses, simply adding a PO can make them point to any other address. Running different programs does not require resynthesis of the AOM, only agreement on the memory map of pointers and shared resources. In this instance of Octavo, the three AOMs, one for each instruction operand, each implement two PO and PI entries, for a total of six pointers per thread.

BTM. Figure 4(b) shows the BTM, which executes branches in parallel with instructions. The BTM monitors the PC and some flags (F) based on the result of the previous thread instruction: If the PC matches the Branch Origin (BO) address, and the flags (F) match the desired Branch Flag (BF) condition (e.g., greater-than-zero), then the BTM generates a Branch Destination (BD′) address and signals the Controller (Figure 4(f)) to perform a jump (J).

Additionally, we can also statically predict the outcome of the branch by setting the Branch Predict Enable (BPE) bit and also the Branch Predict bit to Taken/Not-Taken. Then, if the branch does not go as predicted, the BTM raises the Cancel (C) signal, which gets used by other parts of Octavo to convert to a no-op the ALU instruction in parallel with the branch (which still proceeds). Since the BTM can both optionally and statically predict branches either way, we can always place a useful instruction in parallel with a branch, never requiring an explicit no-op.

Otherwise, if either the PC does not match BO, or F does not match BF, then all BTM outputs remain at zero, which allows us to take the bitwise `OR` of the output of multiple BTMs. This design both avoids slower multiplexers and enables concurrent BTMs to implement multi-way branches, with necessarily mutually exclusive branch conditions, in parallel with a single instruction.



(a) Address Offset Module (AOM)
(b) Branch Trigger Module (BTM)
(c) I/O Read Predication (PRD)
(d) I/O Write Predication (PRD)
(e) I/O Ready/Annul (PRD)
(f) Controller (CTL)
(g) Memory Map

Fig. 4. Major blocks of the Octavo soft-processor.

However, a branch depends on the result of the previous thread instruction, and the current instruction may get annulled (along with any parallel branches) and re-issued for other reasons. Since a re-issued instruction always has itself (as a no-op) as its previous instruction, we have lost the original previous instruction result. A branch in parallel with the re-issued instruction might then proceed incorrectly. Our solution, seen in the lower-right of Figure 4(b), preserves copies of the original Jump (J) and I/O Ready flags out of Stage 3 and delays them to synchronize them with the next thread instruction. When the next thread instruction reaches Stage 3, we use the delayed I/O Ready to determine if this instruction is a re-issue of the previous one. If so (i.e., I/O

Ready is not asserted), then we re-use the delayed Jump flag instead of the current selected flag, re-creating the original conditions for the parallel branch.

We need one BTM instance for each branch we wish to execute in parallel. If the number of branches in a program exceeds the number of BTM instances, then we can manage the BTMs in a manner similar to register allocation, keeping only the "hottest" branches (e.g., inner loops) in the BTMs and re-loading one BTM for the less-frequently encountered branches. In this instance of Octavo, we implement four BTMs to support up to four parallel branches per thread.

PRD (Read). Figure 4(c) shows the I/O Read Predication module connected to one of the A/B Data Memories in stages 4 and 5. In stage 2, we receive the operand address from stage 1 and the Empty/Full bits (E/F) for all the I/O read ports in the corresponding A/B memory. The E/F Select block simply selects one of the E/F bits based on the address, while the I/O Detect block signals if the address refers to an I/O port.

In stage 3, the E/F Mask block lets the selected E/F bit pass through if the address refers to an I/O port. Otherwise, the access refers to memory (which is always ready by definition) and the block will mask the E/F bit (E/Fm) to the "Full" state, indicating available data. Concurrently, the I/O Active block raises the read enable (rden) line for the accessed I/O port, if any. However, the I/O Ready signal, which indicates if *all* accessed read/write I/O ports are ready, will disable the rden signal if not asserted, preventing the completion of the read transaction.

Stages 4 and 5 use the post-AOM read address out of stage 3 to both select Read I/O port data and perform the memory read, then select the correct read value depending if the address refers to an I/O port, signalled by the pipelined I/O Detect output. Finally, the I/O Ready signal will zero-out the Read Data if not asserted. We require two I/O Read Predication modules, one for each of the A and B memories, each addressed by their corresponding A/B instruction source operand.

PRD (Write). Figure 4(d) shows the I/O Write Predication module, connected to one of the A/B Data Memories in stages 4 and 5. In stage 2, we receive the operand address from stage 1 and the Empty/Full bits (E/F) for all the I/O write ports in the corresponding memory. The E/F Select block simply selects one of the E/F bits based on the address, while the I/O Detect block signals if the address refers to an I/O port. In stage 3, the E/F Mask block lets the selected E/F bit pass through if the address refers to an I/O port. Otherwise, the access refers to memory (which is always ready by definition) and the block will mask the E/F bit (E/Fm) to the "Empty" state, indicating the previously written data was received.

Even though the actual I/O or memory write will not occur until later in the pipeline, after the ALU, we check the I/O write port status at the same time as the I/O read ports to keep the predication process simple. If we predicated writes at the point where they occur later in the pipeline, then we could end up in a situation where we have already performed the I/O reads, potentially causing side effects (e.g., consuming I/O data), and cannot complete the instruction due to a full I/O write port. We would then need to save the instruction state at the write port, possibly from up to eight threads. Also, the next thread instruction will have issued before then, and we would have to annul it until the previous I/O write completes, with potentially slow signalling across multiple pipeline stages.

Thus, in stages 4 and 5, we mask the I/O Detect signal if I/O Ready is not asserted and pass the resulting is_I/O signal to the ALU, where it will propagate alongside the instruction until the ALU writes back its result in stage 4. There, the I/O Active block uses the ALU Address (D′ in Figure 4) and the propagated is_I/O bit to enable the appropriate write enable (wren) line, while the ALU Result proceeds to both the RAM and Write I/O port. We require two I/O Write modules, one for each of the A and B memories,

either of which may have their I/O write ports accessed by the D instruction destination operand. As originally published [LaForest and Steffan 2012], the Write I/O port output was at stage 4 instead of 5. This decision seemed to save a cycle on I/O, but each thread issues an instruction every eight clock cycles, while the I/O Reads and Writes within a thread instruction were only seven cycles apart. This difference sometimes made the design of accelerator pipelines awkward, particularly if they contained loops.

PRD (Ready/Annul). Figure 4(e) shows the origin of the crucial I/O Ready (IOR) signal. The All I/O Ready block receives the selected and masked Empty/Full bit (E/Fm) from each I/O Read and Write Predication (PRD) module, all generated within pipeline stage 3. It asserts I/O Ready if all the read E/Fm bits are "Full" and all the write E/Fm bits are "Empty," indicating that all I/O performed by the instruction can proceed at once. We express the Annul bit as the inverse of I/O Ready.

CTL. Figure 4(f) shows the Controller, which issues instruction addresses. A small thread number counter (not shown) indexes the Program Counter Memory (PCM) to read the PC of the next instruction in the thread, as well as that of the current instruction (PC-1). The Jump signal from the BTM replaces the PC with the branch target address BD′. Additionally, if the current instruction was annulled, the Annul signal selects PC-1 as the final output PC, which will re-fetch the current annulled instruction from I Memory. The PCM then replaces its stored PC-1 with the output PC, and its stored PC with the output PC+1, becoming ready to fetch the next instruction or to re-try the current one if it remains annulled. We must register the path from the output PC back to the PCM to avoid a critical path, and thus we must also delay the value of the thread counter by one cycle to write back to the correct PCM entry.

Memory Map. Figure 4(g) illustrates Octavo's memory map, which treats the 4 kiloword write address space of the D operand as four 1 kiloword pages: one for each of the A, B, and I memories, and a new "High" memory area (H) within which we can map new hardware extensions (e.g., the AOM and BTM) without requiring I/O ports but at the price of being "write-only." The A/B read operands index into their corresponding A/B data memories and the PC indexes into I memory (not readable by A/B). We decode writes from the ALU over a single contiguous write address space, enabling writes to only the necessary memories.

ACKNOWLEDGMENTS

The authors acknowledge the influence of Guy Lemieux (UBC) on execution overhead reduction and of Aaron Severance (UBC), who provided the MXP benchmark data and architectural descriptions. The authors also thank Altera, and Blair Fort in particular, for providing support in many ways: funding, Quartus licenses, support, and hardware.

REFERENCES

Altera. 2014. Nios II Performance Benchmarks. Retrieved August 2014 from http://www.altera.com/literature/ds/ds_nios2_perf.pdf.

Alexander Brant and Guy G. F. Lemieux. 2012. ZUMA: An open FPGA overlay architecture. In *Proceedings of the International Symposium on Field-Programmable Custom Computing Machines (FCCM'12)*. 93–96.

A. Canis, S. Brown, and J. H. Anderson. 2014. Modulo SDC scheduling with recurrence minimization in high-Level synthesis. In *Proceedings of the International Conference on Field-Programmable Logic and Applications (FPL'14)*. 1–8.

Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Tomasz Czajkowski, Stephen D. Brown, and Jason H. Anderson. 2013. LegUp: An open-source high-level synthesis tool for FPGA-based processor/accelerator systems. *ACM Trans. Embed. Comput. Syst.* 13, 2 (Sept. 2013), Article 24, 27 pages.

D. Capalija and T. S. Abdelrahman. 2013. A high-performance overlay architecture for pipelined execution of data flow graphs. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL'13)*. 1–8.

Hui Yan Cheah, Fredrik Brosser, Suhaib A. Fahmy, and Douglas L. Maskell. 2014. The iDEA DSP block-based soft processor for FPGAs. *ACM Trans. Reconfig. Technol. Syst.* 7, 3 (Sept. 2014), Article 19, 23 pages.

Alexander Choong, Rami Beidas, and Jianwen Zhu. 2010. Parallelizing simulated annealing-based placement using GPGPU. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL'10)*. 31–34.

Robert Dimond, Oskar Mencer, and Wayne Luk. 2005. CUSTARD - A customisable threaded FPGA soft processor and tools. In *Proceedings of the International Conference on Field Programmable Logic (FPL'05)*. 1–6.

B. Fort, A. Canis, J. Choi, N. Calagar, R. Lian, S. Hadjis, Y. T. Chen, M. Hall, B. Syrowik, T. Czajkowski, S. Brown, and J. H. Anderson. 2014. Automating the design of processor/accelerator embedded systems with legup high-level synthesis. In *Proceedings of the IEEE International Conference on Embedded and Ubiquitous Computing (EUC'14)*.

B. Fort, D. Capalija, Z. G. Vranesic, and S. D. Brown. 2006. A multithreaded soft processor for SoPC area reduction. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'06)*. 131–142.

J. Gray. 2016. GRVI phalanx: A massively parallel RISC-V FPGA accelerator accelerator. In *Proceedings of the 2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM'16)*. 17–20.

Rehan Hameed, Wajahat Qadeer, Megan Wachs, Omid Azizi, Alex Solomatnikov, Benjamin C. Lee, Stephen Richardson, Christos Kozyrakis, and Mark Horowitz. 2010. Understanding sources of inefficiency in general-purpose chips. In *Proceedings of the International Symposium on Computer Architecture (ISCA'10)*. 37–47.

ITRS. 2011. International Roadmap For Semiconductors: Design. Retrieved from http://www.itrs.net/Links/2011itrs/2011Chapters/2011Design.pdf.

Alex K. Jones, Raymond Hoare, Dara Kusic, Joshua Fazekas, and John Foster. 2005. An FPGA-based VLIW processor with custom hardware execution. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA'05)*. 107–117.

Volodymyr V. Kindratenko, Robert J. Brunner, and Adam D. Myers. 2007. Mitrion-C application development on SGI altix 350/RC100. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'07)*. 239–250.

A Krasnov and A Schultz. 2007. RAMP blue: A message-passing manycore system in FPGAs. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL'07)*. 54–61.

M. Labrecque and J. G. Steffan. 2007. Improving pipelined soft processors with multithreading. In *Proceedings of the International Conference on Field-Programmable Logic and Applications (FPL'07)*. 210–215.

Martin Labrecque and J. Gregory Steffan. 2009. Fast critical sections via thread scheduling for FPGA-based multithreaded processors. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL'09)*. 18–25.

Martin Labrecque, Peter Yiannacouras, and J. Gregory Steffan. 2008. Scaling soft processor systems. In *Proceedings of the International Symposium on Field-Programmable Custom Computing Machines (FCCM'08)*. 195–205.

Charles Eric LaForest, Jason Anderson, and John Gregory Steffan. 2014. Approaching overhead-free execution on FPGA soft-processors. In *Proceedings of the International Conference on Field-Programmable Technology (FPT)*.

Charles Eric LaForest and John Gregory Steffan. 2012. OCTAVO: An FPGA-centric processor family. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA'12)*. 219–228.

Charles Eric LaForest and John Gregory Steffan. 2013. Maximizing speed and density of tiled FPGA overlays via partitioning. In *Proceedings of the International Conference on Field-Programmable Technology (FPT'13)*. 238–245.

C. Liu, H. C. Ng, and H. K. H. So. 2015. QuickDough: A rapid FPGA loop accelerator design framework using soft CGRA overlay. In *Proceedings of the 2015 International Conference on Field Programmable Technology (FPT'15)*. 56–63.

Adrian Ludwin and Vaughn Betz. 2011. Efficient and deterministic parallel placement for FPGAs. *ACM Trans. Des. Autom. Electr. Syst.* 16, 3 (June 2011), 1–23.

K. E. Murray, S. Whitty, S. Liu, J. Luu, and V. Betz. 2013. Titan: Enabling large and complex benchmarks in academic CAD. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL'13)*. 1–8.

Mazen A. R. Saghir, Mohamad El-Majzoub, and Patrick Akl. 2006. Datapath and ISA customization for soft VLIW processors. In *Proceedings of the IEEE International Conference on Reconfigurable Computing and FPGAs (ReConfig'06)*. 1–10.

Aaron Severance, Joe Edwards, Hossein Omidian, and Guy Lemieux. 2014. Soft vector processors with streaming pipelines. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'14)*. 117–126.

Aaron Severance and Guy Lemieux. 2012. VENICE: A compact vector processor for FPGA applications. In *Proceedings of the IEEE International Conference on Field Programmable Technology (FPT'12)*. 261–268.

A. Severance and G. G. F. Lemieux. 2013a. Embedded supercomputing in FPGAs with the vectorblox MXP matrix processor. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'13)*. 1–10.

Aaron Severance and Guy Lemieux. 2013b. TputCache: High-frequency, multi-way cache for high-throughput FPGA applications. In *Proceedings of the International Conference on Field Programmable Logic (FPL'13)*, 1–6.

Kuen Hung Tsoi and Wayne Luk. 2010. Axel: A heterogeneous cluster with FPGAs and GPUs. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA'10)*, 115–124.

United States Bureau of Labor Statistics. 2012. *Occupational Outlook Handbook*.

Henry Wong, Vaughn Betz, and Jonathan Rose. 2011. Comparing FPGA vs. custom CMOS and the impact on processor microarchitecture. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA'11)*. 5–14.

H. Wong, V. Betz, and J. Rose. 2014. Quantifying the gap between FPGA and custom CMOS to aid microarchitectural design. *IEEE Trans. VLSI* 22, 10 (Oct. 2014), 2067–2080.

Qinghong Wu and Kenneth S. McElvain. 2012. A fast discrete placement algorithm for FPGAs. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA'12)* (2012), 115–118.

Received January 2016; revised January 2017; accepted February 2017