High-Speed Soft-Processor Architecture for FPGA Overlays

by

Charles Eric LaForest

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy
Graduate Department of Electrical and Computer Engineering
University of Toronto

Abstract

High-Speed Soft-Processor Architecture
for FPGA Overlays

Charles Eric LaForest
Doctor of Philosophy
Graduate Department of Electrical and Computer Engineering
University of Toronto
2015

Field-Programmable Gate Arrays (FPGAs) provide an easier path than Application-Specific Integrated Circuits (ASICs) for implementing computing systems, and generally yield higher performance and lower power than optimized software running on high-end CPUs. However, designing hardware with FPGAs remains a difficult and time-consuming process, requiring specialized skills and hours-long CAD processing times. An easier design process abstracts away the FPGA via an "overlay architecture", which implements a computing platform upon which we construct the desired system. Soft-processors represent the base case of overlays, allowing easy software-driven design, but at a large cost in performance and area. This thesis addresses the performance limitations of FPGA soft-processors, as building blocks for overlay architectures.

We first aim to maximize the usage of FPGA structures by designing Octavo, a strict round-robin multi-threaded soft-processor architecture tailored to the underlying FPGA and capable of operating at maximal speed. We then scale Octavo to SIMD and MIMD parallelism by replicating its datapath and connecting Octavo cores in a point-to-point mesh. This scaling creates multi-local logic, which we preserve via logical partitioning to avoid artificial critial paths introduced by unnecessary CAD optimizations.

We plan ahead for larger Octavo systems by adding support for architectural extensions, instruction predication, and variable I/O latency. These features improve the efficiency of hardware extensions, eliminate busy-wait loops, and provide a building block for more efficient code execution.

By extracting the flow-control and addressing sub-graphs out of a program's Control-Data Flow Graph (CDFG), we can execute them in parallel with useful computations using special dedicated hardware units, granting better performance than fully unrolled loops without increasing code size.

Finally, we benchmark Octavo against the MXP soft vector processor, the NiosII/f scalar soft-processor, and equivalent benchmark implementations written in C synthesized with the LegUp High-Level Synthesis (HLS) tool, and direct Verilog Hardware Description Language (HDL) implementations. Octavo's higher clock frequency offsets its higher cycle count, performing roughly on-par with MXP, and within an order of magnitude of the performance of LegUp and Verilog solutions, but with an order of magnitude area penalty.

Dedication

To my family and friends, near and far.

It's been a life-long road, and you all helped me travel it.

I am to all in your debt.

In Memoriam:

John Gregory Steffan

August 27, 1972 - July 24, 2014

Acknowledgements

Since I was a teenager, I've always wanted to design a computer. You would not be reading this without the help and influence of many.

Firstly, my wife Joy. Do I need to expound on the love, support, and patience of a spouse over a decade spent in academia, with her own career, while raising a family?

Secondly, to Greg Steffan. I can credit pretty much my entire academic success to his ability to pull me back to the big picture after fruitfully letting me get lost in the details, arguing for my own direction. Back in 2008, he proposed I do my MASc on a seemingly small little problem, which grew into a sub-field of its own after we published, and won us a significant award. Right from the beginning, while he co-supervised my undergraduate BIS thesis in 2006, before he even had tenure, we talked about what kind of computer architectures would work best on an FPGA. The work you are holding right now stems from those conversations, and I think I can say we figured out a good solution. Neither of us knew how far it would all go, and I am sorry he did not get to see it completed.

Jason Anderson stepped up to carry the supervision torch, and gave his time and effort to make sure the LegUp benchmarks were great. I am lucky to have had his support, unflagging enthusiasm, and invaluable thesis-writing feedback.

Guy Lemieux, of UBC, influenced the later stages of this work enormously. The entire work on execution overhead originates from a single conversation we had. Guy always took the time to help and explain things, criticizing honestly, greatly stimulating my thinking. One of his students, Aaron Severance, gave his time and effort also to provide MXP benchmark data, better than any I could have done myself, and patiently explained how the machinery worked. Any mistakes or omissions are mine alone.

I have had the luck of also working with Vaughn Betz and Tarek Abdelrahman on my thesis committee, both bringing contrasting and broad perspectives on my research. In fact, every ECE and DCS Faculty member I have spoken with have answered my questions and given help freely.

I have also had the pleasure of great student colleagues, too numerous to mention all, but especially Ruediger Willenberg, Xander Chin, Braiden Brousseau, Robert Hesse, Ming G. Liu, Zimo Li, Tristan O'Rourke, Mihai Burcea, Andrew Shorten, Jason Lu, Henry Wong, Mark Jeffreys, Jasmina Vasiljevic, Ioana Baldini, and Charles Chiasson.

Some roots of this work came from conversations at MIT with Jack Dennis and some of Arvind's grad students in the summer of 2010. Joel Emer of Intel also provided some early feedback. Octavo's initial design improved a lot after speaking with Patrick Doyle, Mark Stoodley, and other people of IBM Markham.

I am also indebted to Blair Fort (as a student), Martin Labrecque, and Peter Yiannacouras. Their earlier soft-processor work laid a lot of the groundwork for my own. I am still learning things from it.

I got nothing but first-class IT tech support from Jaro Pristupa, Eugenia Distefano, and YongJoo Lee, who never blinked at all my software installation requests, and saved my data through RAID failures and my errors. Furthermore, Kelly Chan and Beverly Chu were always available for EECG administrative help, as were Judith Levene and Darlene Gorzo of the ECE Grad Office, and Jayne Leake who managed my TA assignments.

A lot of computational horsepower (over 24 CPU-years of it!), was provided by the GPC supercomputer at the SciNet HPC Consortium [83]. My special thanks to Ching-Hsing Yu and Dr. Daniel Gruner for their enthusiasm at helping me scale my computations, even when I unintentionally broke their system a little sometimes. ("*What do you mean, I used up all the file descriptors?!?*")

The folk at Altera have also provided support in many ways: funding, Quartus licenses, support, and hardware. My special thanks to Blair Fort (as Altera staff) for his freely given help.

Further funding came from the ECE Department, the Queen Elizabeth II World Telecommunication Congress Graduate Scholarship in Science and Technology, the Walter C. Sumner Foundation, and NSERC.

If you read this and feel forgotten, my apologies. Call me up and reminisce!

“He was alive, trembling ever so slightly with delight, proud that his fear was under control. Then without ceremony he hugged in his forewings, extended his short, angled wingtips, and plunged directly toward the sea. By the time he passed four thousand feet he had reached terminal velocity, the wind was a solid beating wall of sound against which he could move no faster. He was flying now straight down, at two hundred fourteen miles per hour. He swallowed, knowing that if his wings unfolded at that speed he’d be blown into a million tiny shreds of seagull. But the speed was power, and the speed was joy, and the speed was pure beauty.”

– “*Jonathan Livingston Seagull*”, Richard Bach

I was born to Rock n’ Roll.

Everything I need.

I was born with the hammer down.

I was built for speed.

– “*Built For Speed*”, Motörhead

It is by caffeine alone I set my mind in motion.

It is by the beans of Java that thoughts acquire speed,

the hands acquire shakes, the shakes become a warning.

It is by caffeine alone I set my mind in motion.

– variation on the “Mentat Mantra”
– attributed to Mark Stein, Arisia SF Convention, Boston, ca. 1993, Sunday morning.

Soundtrack by **deadmau5**.

Contents

List of Tables

List of Figures

Listings

Chapter 1

Introduction

We can think of Field-Programmable Arrays (FPGAs) as "Lego-on-a-chip": a large number of small building blocks which, although not very capable by themselves, we can assemble into larger, useful digital systems. In a nutshell, these building blocks include:

- small Look-Up Tables (LUTs), which can compute any Boolean function of a few inputs (e.g.: 4 to 6) or act as small memories;
- flip-flops, which hold state;
- various arithmetic circuits, such as adders and multipliers;
- larger memory blocks, for efficient bulk storage;
- I/O blocks, which implement low-level off-chip interfaces;
- and a very large amount of programmable interconnect, which allows us to connect all of these building blocks together.

Rather than implement digital systems directly in Application-Specific Integrated Circuits (ASICs), a much longer and more costly process, we can trade-off area, power, and performance, and instead implement such systems using an FPGA. Despite the trade-offs relative to ASICs, FPGAs still enable higher-performance and lower power solutions than even highly-optimized software implementations on high-end CPUs, and unlike ASICs, we can re-configure FPGAs as the application evolves. As power limits and the slowing of Moore's Law (through scaling limits or increasing costs) causes individual CPU performance to level-out after decades of accelerated growth, FPGAs present an avenue for higher performance without all the complications of large-scale parallel programming.

1.1 Motivation

Designing with FPGAs remains a difficult process, if a less costly one than with ASICs, with hours-long CAD processing times after laborious hardware design. At the most basic level, we can use Hardware Description Languages (HDLs) to describe logic fairly directly, but at a low level akin to assembly programming or a stripped-down `C` language.

Alternately, we can use High-Level Synthesis (HLS) techniques to allow higher-level system descriptions, which greatly eases design implementation, exploration, and verification. However, HLS must still translate the description down to HDL, with the same CAD processing time problems, and produces machine-generated "black box" implementations hard to relate to the original source code.

Finally, we can instead abstract away the FPGA itself by using it to implement a more conventional computing platform, and then implement our desired system onto this "overlay architecture". For example, a scalar soft-processor (e.g.: Altera's NiosII) represents the simplest instance of an overlay.

By using overlays, we pay a further price in performance, power, and area, but gain an even faster development cycle, mostly free from FPGA CAD processing time and amenable to high-level software descriptions. Contrary to conventional CPUs, an overlay still benefits from the FPGA's re-configurability, allowing us to incrementally customize the overlay to our application and recoup lost performance.

We propose that overlays already provide a better design process for FPGAs, minimizing the use of HDLs and HLS, and maximizing the return for a given development effort. Which leaves us with the question: *How do we improve the performance of overlays?*

1.2 Contributions

This dissertation focuses on improving the performance of soft-processors on FPGAs, as building blocks for FPGA overlay architectures. Specifically:

- We construct Octavo, a soft-processor architecture tailored to the underlying FPGA, capable of operating at maximal speed.
- We scale Octavo to SIMD and MIMD parallelism, and use partitioning to preserve operating speed under scaling.
- We refine Octavo to support architectural extensions, instruction predication, and variable I/O latency.
- We reduce Octavo's addressing and flow-control overheads, executing them in parallel with ALU instructions.

We expand on these contributions in the following sub-sections.

1.2.1 Chapter 3: The Octavo Soft-Processor

Chapter 3 addresses the problem of adapting overlay architectures to the underlying FPGA. The properties of an overlay architecture which maximizes the effective use of the FPGA structures lead us to a fine-grained multi-threaded soft-processor approach. Unlike prior multi-threaded processors, we use multi-threading to absorb internal circuit delay rather than external memory latency. We introduce a simple technique to robustly determine the amount of pipelining each major FPGA and processor structure requires to maximize use. A flattened memory hierarchy, without caches and directly addressed by the instruction operands, provides a basis for a simplified ISA and memory-mapped I/O ports for efficient future expansion. Despite the minimal ISA, we can synthesize memory indirection more efficiently than conventional MIPS-like ISAs. We present *Octavo*, a ten-pipeline-stage eight-threaded soft-processor, which can operate at the Block RAM maximum of 550MHz on a Stratix IV FPGA. Design space exploration shows that the entire family of Octavo designs scale well over word-width, memory depth, and number of supported threads, presenting a promising foundation for later improvements.

This work was originally published at FPGA 2012, Monterey [77].

1.2.2 Chapter 4: Tiling Overlay Architectures

Chapter 4 addresses problems that emerge when scaling overlay architectures via *tiling*, which replicates computing structures such as datapaths for SIMD parallelism, or entire soft-processors for MIMD parallelism. Tiling designs introduces *multi-localities* (multiple instances of equivalent logic), which the CAD tool optimizes down to single instances, creating artificial critical-path fanouts to physically distant tiles. Rather than manually controlling the optimization of multi-local logic via the CAD tool or HDL source annotations, a designer can use logical partitioning to preserve multi-locality and improve performance, with lower design effort and negligible CAD time cost. Preserving multi-localities improves operating frequency and compute density (i.e.: work per unit area), with benefits increasing with the amount of tiling. For example, partitioning a mesh of 102 scalar Octavo soft-processors improves its operating frequency 1.54x, from 284 MHz up to 437 MHz, while increasing its logic area by only 0.85%.

This work was originally published at ICFPT 2013, Kyoto [78].

1.2.3 Chapter 5: Planning for Larger Systems

Chapter 5 expands Octavo's memory addressing and adds support for annulling and re-issuing instructions. We extend the write address space to eliminate the overlapping of code and data address spaces, doubling the available data memory and removing all non-code data from instruction memory. The extended write address space also creates a "High" memory range to memory-map future hardware additions without consuming precious I/O ports. We add an Empty/Full bit to each I/O port and use them to annul and re-issue any instruction accessing I/O that is not ready. These predicated instructions eliminate busy-wait loops and enable powerful conditional branches in Chapter 6.

1.2.4 Chapter 6: Approaching Overhead-Free Execution

Chapter 6 addresses the intrinsic addressing and flow-control overheads of processors, which lead to a performance gap relative to custom FPGA implementations of algorithms, regardless of relative clock speeds. We extract parallel addressing, flow-control, and useful work sub-graphs out of the Control-Data Flow Graph (CDFG) of programs, and concurrently execute them in Address Offset Modules (AOMs), Branch Trigger Modules (BTMs), and the regular ALU datapath. The AOMs provide memory indirection, shared code across threads, and post-incrementing addressing. The BTMs eliminate all flow-control opcodes, execute branches in parallel with ALU instructions, optionally cancel instructions based on static branch prediction, and support multi-way branches. A program can hoist addressing and branching work out of loops and into the AOMs/BTMs. We compare our optimized micro-benchmark code against an ideal "perfect" MIPS-like CPU, which has no stalls or delay slots. Against this ideal CPU, using the AOMs/BTMs achieves speedups ranging from 1.07x for control-heavy code, to 1.92x for looping code, never performs worse than the original sequential code, and always performs better than a totally unrolled loop. The AOMs/BTMs reduce raw clock speed by only 6.5%.

This work was originally published at ICFPT 2014, Shanghai [74].

1.3 Organization

We organize the remainder of this dissertation as follows: Chapter 2 provides the relevant basic background on FPGA architecture, the system design processes on FPGAs and their limitations, a discussion of overlay architectures, and their potential for higher performance. We present the aforementioned main research contributions in Chapters 3–6, each as mostly stand-alone designs and experiments, and evaluate the resulting soft-processor in Chapter 7. Chapter 8 presents a number of remaining issues left for future work. Finally, Appendices A–H provide common experimental reference information, some benchmark details, and discussions of the design process difficulties encountered.

Chapter 2

Background

[. . .] it might be worth-while to point out that the purpose of abstracting is not to be vague, but to create a new semantic level in which one can be absolutely precise.

Edsger W. Dijkstra (1930–2002) [40]

2.1 FPGA Architecture

This work addresses the question of "*How do FPGAs want to compute?*" in the context of soft-processor overlay architectures. Thus, we take into account the architectural particularities of FPGAs (i.e.: as opposed to custom Application-Specific Integrated Circuits (ASICs)) when designing a high-performance soft-processor. In this section, we present a high-level overview of modern FPGA architecture to point out, and place in context, these significant features. We show generic structures, independent of particular vendor details or implementation technology. Interested readers can read more in Hauck and DeHon's survey book [53].

2.1.1 Logic Elements

Figure 2.1 illustrates the basic Logic Element (LE) which performs general logic computations in an FPGA. Each LE receives a number of single-bit inputs (a–f) into a Look-Up

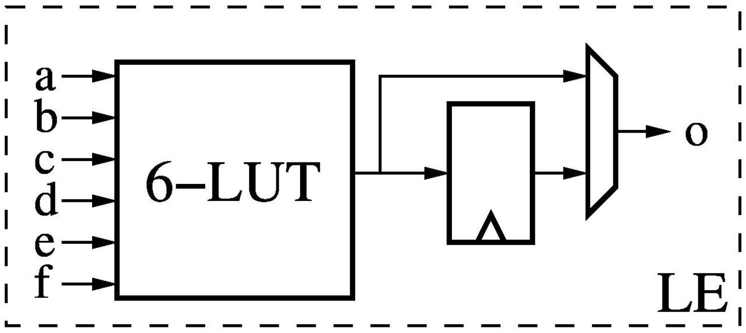
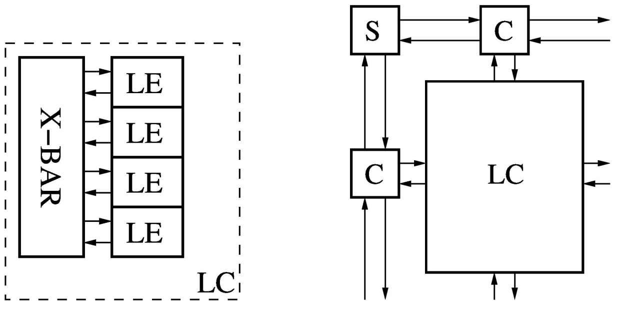


Figure 2.1: Generic FPGA Logic Element (LE)



(a) Logic Cluster (LC) (b) LC Interconnect

Figure 2.2: Generic FPGA Island-Style Building Blocks.

Table (LUT), in this case with 6 inputs (6-LUT). The 6-LUT acts as a small RAM, which we can load at configuration time with the 2^6 bits holding the Boolean truth table of the particular logic function we want to implement. The main output of the 6-LUT (o) then leaves the LE either directly as a combinational circuit or via a flip-flop.

Typically, FPGAs implement a 6-LUT as multiple 4-LUTs and 3-LUTs, allowing multiple outputs for sub-expressions or independent functions. However, we limit ourselves to thinking of the ensemble as a single, universal 6-LUT with a single output, implementing any Boolean function of 6 inputs. This simplification allows us to make good design-time guesses about blocks of logic: whether they will need multiple levels of LEs, and how to pipeline them.

2.1.2 Logic Clusters and Interconnect

Figure 2.2(a) outlines how we can group LEs into larger Logic Clusters (LC) to compute more complex logic functions, apply a given function to a wider bit vector, or locally connect a number of distinct logic functions. We place the LEs in a column, with some immediate nearest-neighbour connections (not shown). Each LE's inputs and outputs connect to a cross-bar (X-BAR) which allows arbitrary connections amongst LEs, fixed at configuration time. Optionally, some FPGAs allow grouping the 6-LUT Boolean truth table RAM within each LE in the Logic Cluster into a single, small Block RAM for more efficient storage than the same capacity implemented using LE output registers alone.

Figure 2.2(b) places the Logic Cluster into a global Interconnect. The Logic Cluster connects its cross-bar inputs and outputs to horizontal and vertical Interconnect via a Connection Block (C). This interconnect contain channels with large numbers of buffered wires, varying in length from a few Logic Clusters to the entire span of the FPGA. Between adjacent channels, and at their intersections, there exists a Switch Box (S), configured as fixed set of connections between channel wires. The Interconnect enables us to connect multiple Logic Clusters together.

2.1.3 Island-Style FPGA Architecture

Figure 2.3 tiles the Logic Clusters and Interconnect into a 5×4 example corner of an "Island-Style" FPGA architecture, where a "sea" of Interconnect surround each "island" of logic. We replace some Logic Clusters with dedicated circuitry to implement specific functions more efficiently than Logic Clusters can. Larger Block RAMs (BRAM) provide dense and fast storage, while Digital Signal Processing Blocks (DSP) provide fast arithmetic functions (e.g.: multipliers). We place each type of resource into a column spanning the height of the FPGA, usually placing multiple Logic Cluster columns together before placing a BRAM or DSP column. Finally, we attach I/O modules along the edges, each

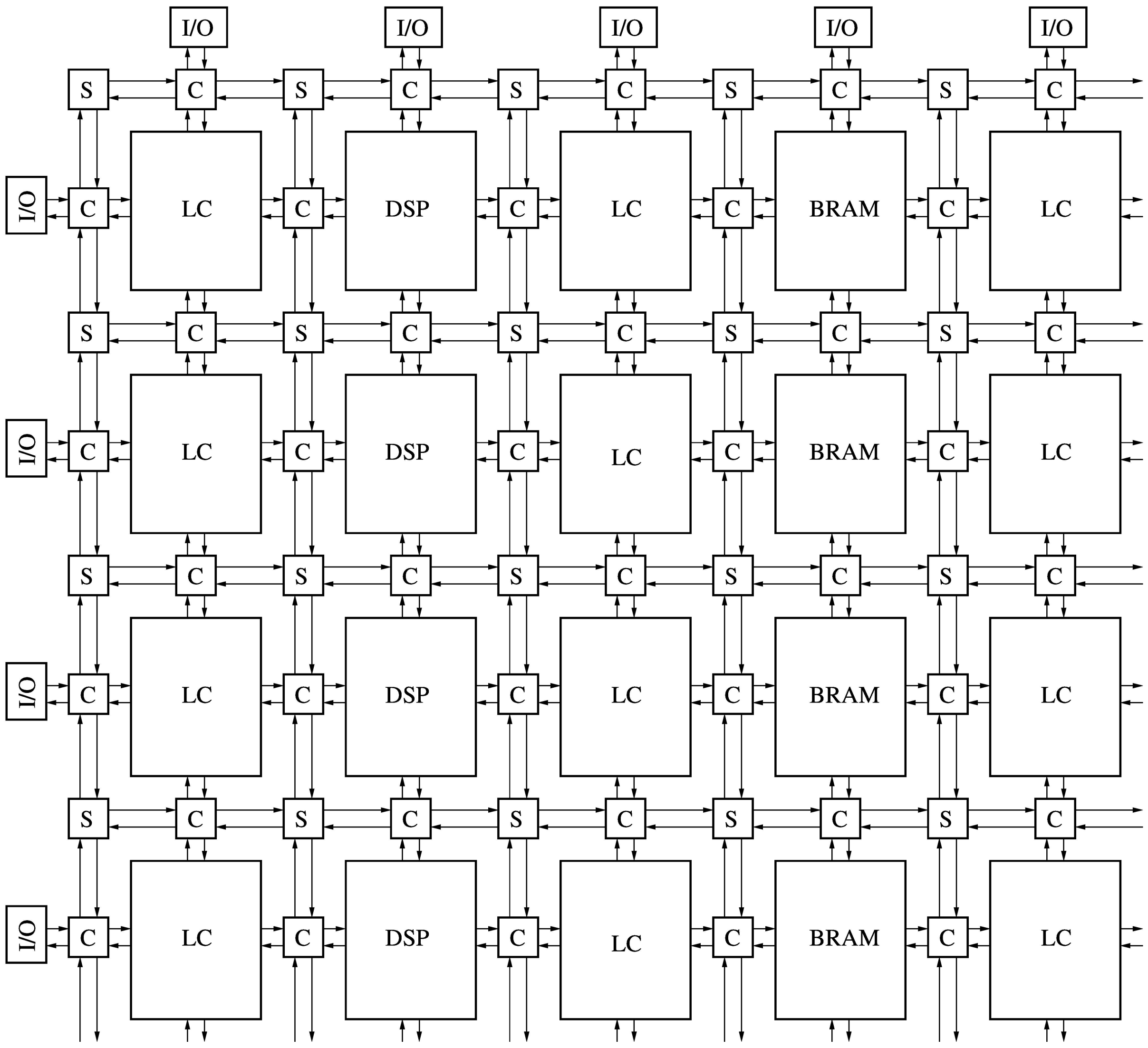


Figure 2.3: Generic Island-Style FPGA Architecture

possibly supporting a number of I/O functions (e.g.: differential signalling, clock recovery, encoding/decoding, protocol stacks, etc. . .).

2.2 Limitations of Design Processes with FPGAs

Despite their great ease relative to ASICs, implementing designs on FPGAs remains difficult: only a minority of engineers possess the specialized skills necessary to do hardware-level design work [118][1]. Furthermore, as the size of the FPGA devices keeps increasing with Moore's Law, but without corresponding speedup from the CAD tools [30], compiling ever-larger designs takes many hours or days (e.g.: 3–30 hours [69], 5 hours [116], "one or more CPU hours" [84], "over 35 hours" [65], etc… [127]). These growing designs also strain our efforts at simulation and verification. This lengthening design cycle, sometimes dubbed the *productivity gap* [59], eats away at the two main advantages of FPGAs over custom ASICs: fast time-to-market and lower Non-Recurring Engineering (NRE) design costs.

Broadly-speaking, an engineer can chose from three FPGA design processes:

- Hardware Description Languages
- High-Level Synthesis
- Overlays

2.2.1 Hardware Description Languages

Hardware Description Languages (HDLs) describe hardware fairly directly, with registers holding state and combinational logic connecting their inputs and outputs. The level of abstraction varies, ranging from low-level, gate-by-gate and bit-by-bit implementations up to behavioural descriptions in terms of Boolean operators, conditional expressions, and vectors of bits. Although several HDLs exist (e.g.: Abel, AHDL, VHDL, Verilog, SystemVerilog, FHDL, MyHDL), VHDL and Verilog represent the majority of HDL use today. This work uses Verilog-2001 as its implementation language.

While HDLs provide a reasonable design process for small, well-defined circuits, they do not scale well to larger systems. Optimized designs make use of vendor-specific FPGA blocks, which HDLs do not express portably. HDLs describe the circuit itself, rather than

[1]Surprisingly, the expected large ratio only applies to the US: roughly 1,361,700 software engineers vs. 83,300 hardware engineers (16:1). In Canada [112]: 146,910 software engineers vs. 27,310 hardware engineers (5:1). All numbers aggregate related jobs, an exclude electrical/electronic engineers to focus on computing

the higher-level system, leading to slow, un-necessarily detailed simulations and verifications, and long compilation time to the target FPGA. Finally, unlike many modern programming languages (e.g.: Python, Perl, Lua, C++), HDLs cannot construct increasingly abstract structures and functions to describe increasingly large and high-level systems, forcing low-level descriptions of high-level ideas.

Even the relatively small, single-person effort of this work suffered from the limitations of Verilog-2001 when designing highly-configurable sub-assemblies and composing them into larger ones. We detail these limitations in Appendix D.

2.2.2 High-Level Synthesis

In contrast to HDLs, High-Level Synthesis (HLS) aims to automatically translate a more practical High-Level Language (HLL) description of algorithms and systems down to HDL, analogous to C++ providing higher-level abstractions on top of the underlying C system. There have been too many attempts at HLS to list them all here, and most are defunct: they either never left the lab, are no longer available commercially, or not in active development. Currently, various groups maintain HLS systems based on various HLLs, such as Bluespec (based on Haskell) [95], SystemC (a set of C++ classes) [50], OpenCL (thread-parallel C code kernels) [109], LegUp (plain C to hardware synthesis) [22], Xilinx's Vivado, and others (see [16] for an overview). We benchmark against LegUp in Chapter 7.

While these HLS systems do provide an easier way to describe hardware systems, and considerably accelerate their simulation, interfacing with the FPGA vendors' CAD tools requires converting the HLL descriptions to HDL. Thus, these HLS tools still suffer from long compilation times, and the resulting "black-box" implementations are hard to debug when their behaviour differs from that of the simulated HLL.

2.2.3 Overlays

Overlays primarily differ from HDLs and HLS by providing a layer of abstraction between the implementation of a system and the FPGA hardware. Implementing a system on an FPGA overlay does not require using FPGA CAD tools, avoiding their long compile times and design process 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 trade-offs and programming models. We can divide the overlay approaches into scalar soft-processors, vector processors, and "virtual" FPGA fabrics and Coarse-Grain Reconfigurable Arrays (CGRAs).

Overlays: Design Cycle

In a nutshell, the overlay design cycle moves most of the iterations to software development instead of hardware development, which includes time-consuming Place-and-Route (P&R) of circuits onto the FPGA after each design change.

Place-And-Route presents the most intractable step and consumes the majority of CAD processing time: At a glance, placing circuits implemented in n Logic Clusters amongst m possible locations on the FPGA (for $m > n$) has on the order of $O(m! - (m - n)!)$ possible solutions. For example, in Section 4.7.4 (pg. 80), where we create a mesh of 102 soft-processors, P&R takes up 84% of the total 2-hour CAD time. Detailed CAD time breakdowns can be found in Murray, Whitty, Liu, *et al.* [92].

Instead, the target application initially suggests 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 poses excessive difficulties or lacks a necessary feature, 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.

Overlays: Scalar Soft-Processors

Single scalar soft-processors represent the degenerate case of overlays. They implement a conventional processor architecture onto the FPGA to allow a direct migration of existing software and easy addition of new hardware.

On the other hand, soft-processors cannot provide as high a performance as "hard" CPUs. For example, even on the highest speed grade Stratix IV devices, Altera's MIPS-like NiosII/f soft-processor hovers around 240 MHz [11], despite a reconfigurable logic rating of 800 MHz and the DSP and RAM blocks ratings of up to 600 MHz [10]. Because of their relatively low F_{max}, even if we fully populate an FPGA with soft-processors, and somehow keep it entirely busy, virtually all the hardware would spend about half a clock cycle waiting for a few critical paths. Even in this best-case scenario, we use only about half of the potential computing capacity of the FPGA.

Nonetheless, soft-processors are in wide use. FPGA vendors often provide their own soft-processor designs, while several engineering firms provide reproductions of legacy CPUs such as the MOS Technology 6502 and Zilog Z80, with fidelity ranging from binary-compatible to cycle-accurate. Some CPU vendors have released HDL descriptions of their CPUs for open use.

Some commercial and research soft-processors include:

Open Commercial CPUs Sun Microsystems (now Oracle) released open-source versions of their server CPUs (OpenSPARC T1 and T2), as did Gaisler Research for their aerospace-oriented LEON CPU, also SPARC-based.

FPGA Vendor Soft-Processors These include Xilinx's 32-bit MicroBlaze and 8-bit PicoBlaze, Altera's 32-bit NiosII, and Lattice Semiconductor's LatticeMico32 and LatticeMico8. All of the FPGA vendor soft-processors implement a conventional MIPS-like architecture for easy software portability.

Multi-Processors As FPGAs grow in size, some research aimed to improve performance by using multiple soft-processors together. The PolyBlaze system enhances the MicroBlaze soft-processor to implement up to 8-way Symmetric Multi-Processing (SMP) Linux systems [86]. Similarly, the Multi-Level Computing Architecture (MLCA) coordinates up to 8 NiosII cores into a single Out-of-Order superscalar-like system [24]. Unnikrishnan *et al.* explore the synthesis and compilation to application-tuned many-cores and interconnect networks [119]. Similarly, MARC generates a many-core system from OpenCL kernels, with multi-threaded compute cores and a global shared memory [79]. Finally, Labrecque *et al.* compare the performance scaling of uni- and multi-processors on FPGAs, specifically their caches and memory latencies [73].

Multi-Threaded Some other research aims to improve the compute density (i.e.: work done per unit area) of soft-processors, usually through multi-threading. The CUSTARD system first explored the design space of multi-threaded soft-processors, comparing fine and coarse-grained threading, removal of result forwarding paths, and generation of custom instructions [41]. Fort *et al.* implement the Nios-compatible UTMP II multi-threaded processor to demonstrate a significant area reduction by allowing multiple programs to share the same hardware blocks [47]. Labrecque's exploration of multi-threaded processors also finds significant area savings [71], and dynamically schedules threads to reduce the overhead of locks on critical code sections [72]. Labrecque *et al.*'s previously cited work [73] also compares single and multi-core multi-threaded processors with uni-threaded processors.

FPGA-Centric Alternatively, some soft-processors start from the other end and tailor themselves to the FPGA first rather than the application, hoping to reduce area and/or improve performance. The Leros 16-bit microcontroller aims for minimal resource usage by using an accumulator-based architecture, with a performance comparable to Xilinx's 8-bit PicoBlaze [102]. Similarly, the iDEA processor makes efficient use of the complex DSP blocks on Xilinx FPGAs to achieve similar performance to a MicroBlaze, but at half the area and twice the clock speed [26,27]. Ehliar *et al.* design a single-threaded datapath with carefully designed ALU and result forwarding to reach a high F_{max} on Xilinx Virtex-4 FPGAs [43]. The ρ-VEX processor uses dynamic partial FPGA reconfiguration to act either as two 2-way VLIW processors or a single 4-way VLIW processor to favour either data parallelism or instruction parallelism [14].

Overlays: Vector Processors

Given the flexibility and ease-of-use of scalar soft-processors, extending them to process vectors of data provides a straightforward way to translate the spatial parallelism of FPGAs into a well-understood form of data parallelism. Soft vector processors (SVPs) increase parallelism without the logic complexity and heavily multi-ported memories (and thus lower net performance) of superscalar soft-processors [1, 2, 100].

All current SVPs share a similar architecture: a scalar soft-processor controls a number of vector lanes executing custom instructions on local vector memory. While the vector lanes process data, the scalar processor performs any scalar tasks, updates pointers and counters, and prepares the next block of vector processing. By effectively unrolling loops into vector operations, vector processors achieve significant speedups (up to 20x with a single vector lane [31]) by eliminating inner loop overheads such as counting, conditional branches, and loads/stores.

Yu and Lemieux make the initial argument for SVPs based on the abundance of memory and DSP blocks on FPGAs [134], later refined and named VIPERS [133]. Later teams, also at the University of British Columbia (UBC), greatly expand upon this work. The VESPA SVP (University of Toronto) further explores the design space for portability, customization to the application, and adapting to the underlying FPGA hardware [130–132]. VEGAS (UBC) improves on VIPERS by moving to a single vector scratchpad memory, rather than register files, fracturable ALUs for sub-word calculations, and address generation hardware, all lowering overhead and using on-chip memory more efficiently [31]. VENICE (UBC) further improves VEGAS by removing computation overhead, including: 2D and 3D vector address generation, operation on arbitrarily unaligned vectors, and per-byte conditional flags for conditional vector moves [106]. Finally, MXP (UBC, commercialized by VectorBlox) improves VENICE in many ways, including: scatter/gather with a coalescing cache [108], 2D and filtered DMA, custom vector instructions [107], and adding streaming vector pipelines for floating-point support [105]. We benchmark against MXP in Chapter 7.

Overlays: "Virtual" FPGAs and CGRAs

Finally, instead of a processor-based overlay, we can instead implement a "virtual" FPGA-on-an-FPGA which presents an easier (and non-proprietary!) FPGA fabric for

CAD tools, at some cost in area and speed. The ZUMA FPGA overlay uses LUTRAMs (i.e.: RAMs built from the LUTs in a Logic Cluster) to implement virtual LUTs and multiplexers, at a 40× area penalty over the underlying FPGA, but portably to both Xilinx and Altera devices [19]. Koch, Beckhoff, and Lemieux improved ZUMA by mapping the overlay's interconnect directly onto the FPGA switch fabric, reducing area overhead by 3.7×, and demonstrated using the overlay to implement custom instructions for a MIPS-like soft-processor [68].

Alternatively, we can map a program's data-flow graph (DFG) onto a Coarse-Grained Reconfigurable Array (CGRA) implemented on top of a conventional FPGA, enabling very rapid synthesis and P&R. However, this approach is in earlier research stages than processor-based overlays. Capalija and Abdelrahman's CGRA overlay contains integer or floating-point functional units connected via elastic pipelines [23]. Their overlay presents a simpler P&R problem than the underlying FPGA: fitting a DFG onto the overlay takes a few seconds at most. They maintain a high raw F_{max} via floorplanning, in contrast to our results in Chapter 4, which did not find an advantage to floorplanning.

2.3 Significant FPGA Features Affecting The Design Process

Given the previous overview of FPGA architecture, one could easily dismiss FPGAs as a larger, slower, more power-hungry substitute for ASICs [70,125,126]. However, FPGAs gain enormously in relative ease and flexibility of design, as they abstract away the worst of the difficulties of designing custom hardware. When aimed at specific tasks, rather than general-purpose computations, FPGAs can achieve power and performance figures closer to that of equivalent ASICs [52] (albeit with a larger area) than the best-case equivalent software running on a CPU [33,85].

When implementing a hardware design onto an FPGA, we must keep in mind some of the properties of the underlying FPGA hardware, and of its CAD tools, if we wish to reach the best possible performance:

Boolean Optimizations On the one hand, extensive and reliable logic synthesis and Boolean optimizations free us from having to implement logic directly in low-level, error-prone Boolean expressions. Instead, we can take advantage of behavioural descriptions such as if-statements and leave the translation to Boolean logic, and its optimization, to the CAD tool.

Furthermore, if we design a logic function to require 6 or fewer inputs, then we can reasonably assume it maps to a single LE per output bit. This assumption allows us to estimate, at design time, both the area and the speed of logic functions. For example, a single LE can encode up to 16 different Boolean functions of 2 inputs[2]. Thus an ordinary 32-bit Logic Unit would approximately use 32 LEs, plus some interconnect routing if each Logic Cluster contains fewer than 32 LEs. We design a similar Logic Unit in Chapter 3.9.

Logical Partitioning On the other hand, the CAD tool does not yet consider the location of the synthesized logic. If a certain sub-function appears repeatedly, the CAD tool may optimize it to a single instance feeding all its destinations. In large designs, these destinations may physically lie far from the single source instance, artificially adding routing delay and slowing down the final circuit, regardless of the area saved by logic optimization. We can impart this spatial knowledge to the CAD tool by placing repeated sub-functions into separate logical partitions to prevent excessive optimizations. We use such logical partitioning in Chapter 4 to maximize the speed of SIMD and multi-core soft-processors.

[2]Interestingly, this exactly implements the set of all possible 2-input Boolean functions [34].

The Cost of Multiplexers When implemented directly in silicon, multiplexers can cost as little as one pass-transistor per input, plus associated selector logic. However, FPGAs use LEs to implement general multiplexers. If the number of input and selector bits exceed the number of LUT inputs in the LE, then the CAD tool must split the implementation over multiple chained LEs, increasing area and lowering speed. Furthermore, due to routability limitations, the CAD tool may have to pack these multiple LEs into separate Logic Clusters, further worsening speed and area.

Thus, FPGAs discourage design implementations which make "large" decisions in a single step, such as Content-Addressed Memories (CAMs) and cross-bars[3]. If possible, we should break down large multiplexers into smaller units separated by pipeline registers, or alternately avoid multiplexers altogether.

Fortunately, we can easily do both: the 6-LUT inside an LE naturally fits a 4:1 multiplexer (4 data bits plus 2 selector bits), and can immediately register its output into the LE's flip-flop. Furthermore, if the input bits have "1-of-N" encoding (i.e.: only one bit set at any time), then we can use the 6-LUT as a 6-input `OR` gate to merge them using fewer LEs and no selector bits. We do exactly that in Chapter 6.5.2 to calculate multiple branches in parallel.

Interconnect Delay, Pipelining, and Retiming Interconnect delay forms the majority of the total delay on FPGAs [56], and has worsened across device generations [93, Fig. 1]. The interconnect channels have relatively long wires, with many configurable sources and sinks adding significant intrinsic capacitance. However, we can limit the length of the connections through the channels by pipelining logic functions, which also allows the CAD tool to retime logic across LEs and potentially further improve delay.

Simply implementing a logic function by itself on an FPGA does not provide a good delay estimate, since the CAD tool can place the Logic Clusters containing the function

[3]Though some recent research cleverly uses Block RAMs as time-multiplexed components of high-throughput switches [35]

as closely as possible without interference from any other circuitry. This approach only gives an approximate lower bound to the delay of the function on that FPGA. That said, we can leverage the CAD tool to more robustly determine the actual delay of a function, and also the required number of pipeline stages to reach a certain operating frequency.

Rather than implementing a function by itself, with dangling inputs and outputs, we instead loop the outputs of the logic function back to its inputs (taking care to avoid cancelling away logic), which forces the CAD tool to realistically use the Interconnect throughout that function's implementation. Placing some Logic Clusters in close proximity necessarily further separates other Clusters. We then add pipeline stages, sometimes manually retimed into the current critical path, until the logic function operates at the desired speed. The resulting implementation preserves its performance when integrated into a larger pipeline. We use this "self-loop characterization" technique in Chapter 3 to design a maximally fast soft-processor.

Place-And-Route Randomness Even with all the previous design techniques and precautions, we do not immediately get a reliable performance estimation from the CAD tool, especially for fast pipelines where a few percent difference add up to many MHz.

Each Place-And-Route (P&R) solution depends on pseudo-random processes, and may mislead us by returning fast or slow outlier results. For example, the calculated maximum clock speed (F_{max}) of the soft-processor shown in Figure 6.4 (pg. 113) varies over a 12% range, which for this design represents a difference of up to 63 MHz between any two P&R solutions.

When constructing high-performance designs, integrating a misleadingly fast sub-pipeline into a larger pipeline may make it difficult to meet performance targets in later P&R runs, with the true cause now obscured by the additional circuitry via retiming and generally greater complexity[4].

[4] Exploration with Logical Partitioning may isolate the true critical paths (Chapter 4), but we should avoid the problem in the first place.

To avoid outlier P&R results, we compute the average operating speed of a pipeline over a number of P&R runs (e.g.: 10, used throughout this work), each starting from a different initial random seed. The average gives a more reliable measure of achievable speed. Additionally, multiple P&R runs also reveal how many of all the solutions meet or exceed the performance target, giving a hint as to the robustness of the design performance. Similarly, when doing design space exploration, comparing average speeds better measures the impact of any design changes. Running multiple P&R runs also increases the likelihood of finding the best-possible solution. When benchmarking different systems, we want to fairly compare them at their peak possible performance, rather than the average. We use average F_{max} comparisons throughout this work, except for Chapter 7 where we benchmark at peak F_{max}.

2.4 The Potential for Higher-Performance Overlays

Given the trade-offs of HDLs, HLS, and Overlays, how do we improve the FPGA design process? We instead claim that overlays already provide an easier and faster development process (Section 2.2.3). The problem then reduces to: *How do we improve the performance of overlays?*

Existing overlays use conventional processor architectures to maintain compatibility with existing software and tools. Unfortunately, this compatibility means ignoring the underlying FPGA, treating it as a fungible, abstract substrate. Although some overlays achieve a net performance gain over others, their datapaths still use the FPGA hardware poorly in terms of area and clock speed versus more direct HDL implementations.

Instead, accounting for the architecture of the underlying FPGA might allow for an overlay which reaches some of the "sweet spots" of the hardware. By taking the generic FPGA architectural descriptions from Section 2.1 as a lens for a real device, we may find hints towards a more FPGA-centric computer architecture. The following descriptions, ratings, and results apply to an Altera Stratix IV device [5, 10, 81]:

Interconnect Structure The FPGA interconnect does not consist exclusively of long wire channels running alongside Logic Clusters (Figure 2.2(b), pg. 8), and does not connect equally in all directions. For example, each Stratix IV Logic Array Block (LAB, corresponding to a Logic Cluster in Figure 2.2(a)) consists of 10 Adaptive Logic Modules (ALMs, each approximately containing the equivalent of 2 Logic Elements from Figure 2.1). Each LAB can directly drive a total of 30 other ALMs, mostly in its left and right neighbouring LABs. Beyond this range, we must use sufficient pipelining and explicit logic duplication to limit the interconnect lengths and avoid critical paths. We address pipelining in Chapter 3 and logic duplication in Chapter 4. We do both at a higher level and with greater automation than previous high-performance designs [56], which manually placed and routed the logic.

Block RAMs Block RAMs have the highest operating speed of all Stratix IV hard blocks. Large designs tend to exhaust the supply of Block RAMs before any other hard block. Thus, to get the most work out of an FPGA, a design must maximize the work done by the Block RAMs. When connecting Block RAMs together, only two pipeline stages suffice to fully absorb the interconnect delay and grant a theoretical speed of 773 MHz. However, the minimum clock pulse width requirements limits the actual speed to 550 MHz. Furthermore, contrary to most commodity RAM which holds words of 32 or 64 bits, FPGA Block RAMs support a range of widths from eight to 72 bits, in multiples of eight or *nine*, meant to support error correction. We measure the necessary pipelining in Chapter 3, and take advantage of unusual word widths throughout this work.

Logic Synthesis The Stratix IV Adaptive Logic Module (ALM, corresponding to Logic Elements in Figure 2.1) contains a number of 3-LUTs and 4-LUTs, which we can approximate as a pair of 6-LUTs with some shared inputs. Some Boolean functions will map more effectively to these LUTs than others. Deliberately designing Boolean functions to limit themselves to 6 inputs guarantees a single logic-level implementation using 1 LUT per output bit, registered if needed. We explicitly follow this constraint when we design the Logic Unit in Chapter 3.9, and use it as a guideline when designing addressing and branching modules in Chapter 6.

Arithmetic In addition to Logic Elements, each Stratix IV LAB contains 20 stages of dedicated ripple-carry logic for adders and subtractors. The performance of ALUs also affects the total amount of work we can extract from an FPGA. Without pipelining, a 32-bit adder computes at 506 MHz: slower than the Block RAMs. Two pipeline stages suffice to exceed the 550 MHz limit of Block RAMs and avoid a performance bottleneck. We demonstrate such a fast adder/subtractor in Chapter 3.9. This same knowledge also tells us that smaller adders (e.g.: 10 bits, for address offsets) will not require pipelining. We take advantage of single-stage small adders in Chapter 6.

DSP Blocks The Stratix IV DSP blocks perform Multiply-Accumulate operations. Their fully-pipelined operating speed varies from 600 MHz at 18 bits down to 480 MHz at 36 bits. Using two DSP blocks clocked in counter-phase on a 300 MHz half-rate clock and fed a demultiplexed data stream allows full-width 36x36bit multiplies with 72-bit results at a rate of 600 MHz, thus also avoiding a bottleneck when coupled to 550 MHz Block RAMs. We integrate such a dual-pipeline multiplier into the ALU in Chapter 3.9.

Conclusion With some careful design, and by accepting some pipelining latency, we can maximize the amount of work done by the interconnect, memory, logic, and arithmetic components of an FPGA. With a suitable architecture and programming model, we could create an overlay with the highest operating frequency possible and which uses each cycle to its fullest.

2.5 Summary

Field-Programmable Gate Arrays (FPGAs) provide an easier and less expensive way to implement custom computing hardware than Application-Specific Integrated Circuits (ASICs). However, as FPGAs and the designs implemented upon them grow in size, current design processes face limitations: Hardware Description Languages (HDL) do not scale to large system designs and require hours of Place-And-Route (P&R) CAD time. Efforts to implement designs using High-Level Synthesis (HLS) do improve the design process, but still ultimately resort to conversion to HDLs, with the same drawbacks. Finally, Overlay architectures abstract away the FPGA hardware to provide a software-friendly target for system design, albeit at a greatly reduced performance and increased area relative to HDLs and HLS, usually by implementing one or more soft-processor(s). By accounting for the significant features of FPGAs which affect circuit design, primarily pipelining to control delay, we can potentially improve the performance of overlays.

Chapter 3

The Octavo Soft Processor

The FPGA substrate encourages soft-processors to have larger, low-associativity caches, deeper pipelines, and fewer bypass networks than similar hard processors.

Henry Wong, Vaughn Betz, and Jonathan Rose [125, 126]

Overlay processor architectures allow FPGAs to be programmed by non-experts using software, but prior designs replicate the architecture of their ASIC predecessors. This approach gains compatibility with established programming systems, but pays a heavy cost in performance since FPGAs do not provide the same building blocks as ASICs. In this chapter we develop a new processor architecture which, from the beginning, accounts for and exploits the predefined widths, depths, maximum operating frequencies, and other discretizations and limits of the underlying FPGA architecture.

From this design effort, we created Octavo, a ten-pipeline-stage eight-threaded processor that operates at the Block RAM maximum of 550MHz on a Stratix IV FPGA. A great number of parameters define Octavo's configuration, allowing us to explore trade-offs in datapath and memory width, memory depth, and number of thread contexts.

This work originally appeared at FPGA 2012, Monterey [77].

3.1 Design Goals

From our previous observations in Chapters 2.3 and 2.4 on how to maximize the work performed by individual FPGA structures, we ask: *How do we create an overlay architecture which maximizes the use of the underlying structures of an FPGA?* Such an ideal architecture would meet the following design goals:

Abundant Parallelism The overlay supports a variety of parallelism, both as homogeneous (SIMD) and heterogeneous (MIMD) tasks, to adapt to the target application and potentially make use of the entire FPGA device.

High Clock Frequency The overlay maintains a high operating frequency (F_{max}), at or close to the actual actual hardware maximum (550 MHz, limited by Block RAMs). Otherwise, even with seemingly full usage of the FPGA, the overlay hardware remains stable and idle for much of each clock cycle, wasting potential for work.

Low Architectural Overhead A high F_{max} does not suffice for high performance. The overlay also minimizes the number of instructions required to do useful work, and conversely, minimizes the number of cycles per instruction. Ideally, each primitive operation (e.g.: add, multiply, `XOR`, memory load) requires one, single-cycle instruction running at a maximal F_{max}. Multi-cycle operations hoist their overhead outside of loops, or get converted to dedicated hardware, to amortize their cost down towards a single cycle.

Few Stalls The overlay avoids delays from shared hardware contention, sequential data dependencies, and external memory latencies, aiming to have each cycle do useful work.

Simplicity And Minimalism The overlay bases itself on a simple architectural foundation which composes into new features or larger systems, without introducing the aforementioned architectural overhead. If the overlay instead begins with an existing architecture (hardware or software), the details of that architecture act as an "attractor" to other existing solutions, limiting exploration. Features of existing solutions may emerge in the overlay, but only by construction, not assumption.

Congruence With FPGA Structures The low-level features (e.g.: word widths, pipeline depths, address spaces, and primitive operations) of the overlay match those favoured by the underlying FPGA, even if they might not match those of conventional ASIC processors. Otherwise, the overlay introduces architectural overhead by emulating a feature, or leaves hardware unused.

3.1.1 A Multi-Threaded Overlay

Our overviews of FPGAs (Chapters 2.3 and 2.4) point to the inevitability of pipelining to get the most work out of FPGA structures. However, dependencies between pipeline stages often prevent full utilization, and mitigating the dependencies over a single thread of execution (e.g.: result forwarding, branch prediction, pipeline interlocks) reduces performance either by lowering operating frequency and/or increasing the average cycles-per-instruction count.

Therefore, to gain the benefits of pipelining, but avoid dependencies, we propose a fully pipelined and multi-threaded overlay architecture with the simplest multi-threading scheme: we apply the required number of pipeline stages to reach the maximum operating frequency (F_{max}), then multiplex the same number of hardware threads over the pipeline in a strict round-robin order (i.e.: we issue an instruction from the next thread in sequence each clock cycle, without variation).

A strict round-robin multi-threaded and fully-pipelined overlay architecture addresses the properties of an overlay which maximizes the use of the underlying FPGA. Independent threads provide initial parallelism, either SIMD, MIMD, or both. Full pipelining enables a high F_{max} to minimize hardware idle time. Each thread instruction appears to complete in a single-cycle, and can each complete a larger amount of work over the length of the pipeline. Strict round-robin thread scheduling eliminates stalls from the pipeline, and provides a deterministic relationship between threads to simplify programming. Finally, full pipelining also allows the overlay to match the necessary pipelining of FPGA structures, and does not impose other architectural constraints from above.

By disallowing the conventional practice where a stalled thread (e.g.: due to memory latency) relinquishes its slot to another waiting thread, we nearly eliminate data and structural hazards[1], greatly simplifying the control logic and any interactions between threads. We can then compose these highly-regular threads together to tackle larger programming problems, despite their limited individual performance.

3.2 Related Work

Many prior FPGA-based soft-processors designs have been proposed, although these have typically inherited the architectures of their ASIC predecessors, and none have approached the clock frequency of the underlying FPGA hardware. Examples include soft uniprocessors [3,128], multi-threaded soft-processors [32,41,47,71,90,91], soft VLIW processors [14,60,97,101], and soft vector processors [31,130,133]. Jan Gray has studied the optimization of processors specifically for FPGAs [49], where synthesis and technology mapping tricks are applied to all aspects of the design of a processor from the instruction set to the architecture.

Historically, several computers used multi-threaded pipelines: The CDC 6600 [115], and its descendant the CDC Cyber 170 [25], offloaded I/O tasks to 10 Peripheral Processors, implemented as a single processing unit multiplexed in a strict round-robin manner over 10 separate processor states and memories. Separating I/O from the main CPU allowed the designers to focus on its (superscalar) parallelism for scientific applications. The Denelcor HEP [39,61] maintained separate multi-threaded pipelines for instruction execution and memory accesses, moving threads between them as necessary. Each of up

[1]Not quite. Some branching and configuration single-cycle delay slots still exist, but cancelling branches (Chapter 6.5.2) and instruction re-ordering virtually eliminate them.

to 16 processors had an 8-deep pipeline but supported up to 128 threads. Also, each memory location had an associated empty/full bit to automatically synchronize dependent instructions, regardless of thread, forcing instructions to wait until their results were empty and their operands full. The Tera MTA [20, 111], the HEP's successor, pushed the same concepts further with larger numbers of processors (up to 256), a 3-way VLIW ISA, and allowed each thread to issue up to 8 outstanding memory references.

These past machines used multi-threading to absorb the access latency to main memory (or I/O), switching between threads while the pipelined load or store completed, rather than using a conventional cached memory hierarchy, which would perform poorly on large or irregular data sets. *In contrast, our proposed overlay architecture uses multi-threading to absorb the latency of the underlying FPGA structures*, switching between threads as each instruction moves one step down the pipeline, rather than using super-scalar, vector, or multi-core parallelism to complete more work per (slower) cycle.

3.3 Contributions

To begin,we focus on the architecture of a single soft-processor core and provide the following four contributions:

- we present the design process leading to an 8-thread multi-threaded soft-processor family that operates at up to 550MHz on a Stratix IV FPGA;
- we demonstrate the utility of *self-loop characterization* for reasoning about the pipelining requirements of processor components on FPGAs;
- we present a design for a fast multiplier, consisting of two half-pumped DSP blocks, which overcomes hardware timing and CAD limitations;
- we present the design space of soft-processor configurations of varying datapath and memory widths, memory depths, and number of pipeline stages.

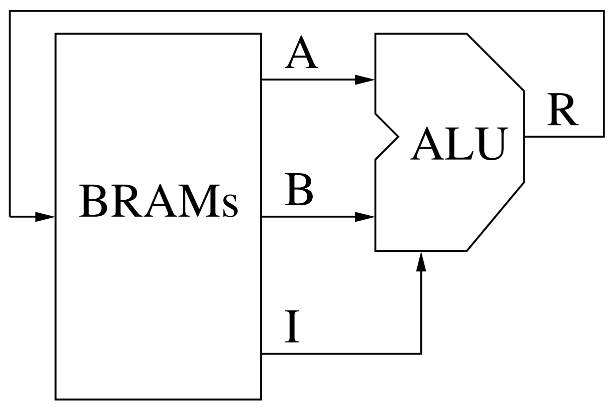


Figure 3.1: High-level overview of the architecture of Octavo.

3.4 Introducing Octavo

As a starting point, we show the simplest processor design we could imagine in Figure 3.1, which is composed of at least one multi-ported memory ($BRAMs$) connected to an ALU, supplying its operands (A and B) and instruction (I) and receiving its result (R). We argue that separate memory cache and register file storage is unnecessary: on an FPGA both are inevitably implemented using the same BRAMs. Thus, we unify memory and registers, reducing the data and instruction memories and the register file into a single entity directly addressed by the instruction operand fields. For this reason our final architecture is indeed not unlike the simple one pictured, having only a single logical storage component (similar to the scratchpad memory proposed by Chou *et al.* [31]). We demonstrate that this single logical memory eliminates the need for immediate operands and load/store operations, but for now requires writing to instruction operands to synthesize indirect memory accesses.

Via the technique of *self-loop characterization*, where we connect a component's outputs to its inputs to take into account the FPGA interconnect, we determine for memories and ALUs the pipelining required to achieve the highest possible operating frequency. This leads us to an overall eight-stage processor design that operates at up to 550MHz on a Stratix IV FPGA, limited by the maximum operating frequency of the BRAMs. To meet the goals of avoiding stalls and maximizing efficiency, we multi-thread the processor such that an instruction from a different thread resides in each pipeline stage [41,47,71,90,91], so that all stages are independent with no control or data hazards or result forwarding between them.

We name our processor architecture *Octavo*[2], for nominally having eight thread contexts. However, Octavo is really a processor *family* since it is highly parameterizable in terms of its datapath and memory width, memory depth, and number of supported thread contexts. This parameterization allows us to search for optimal configurations that maximize resource utilization and clock frequency.

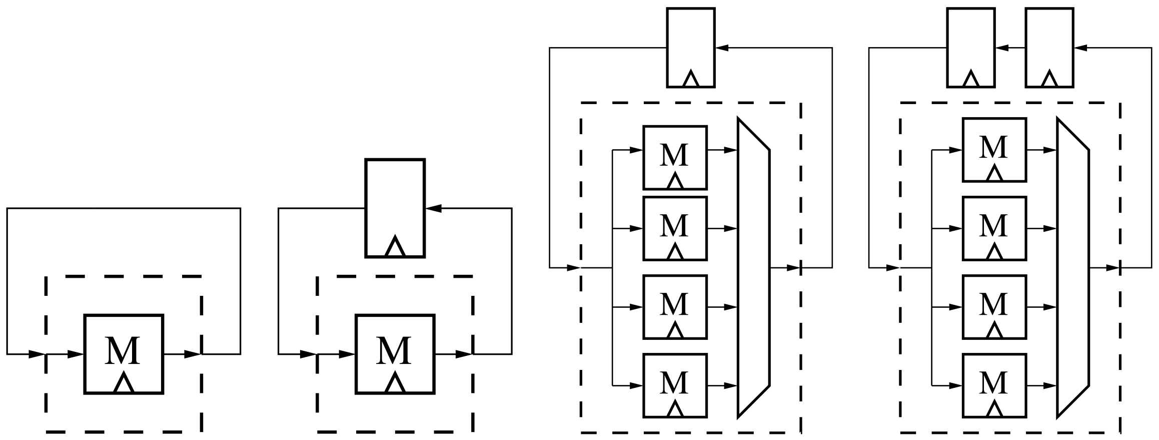
3.5 Experimental Framework

We evaluate Octavo and its components on an Altera Stratix IV FPGA of the highest speed grade, although we expect proportionate results on other FPGA devices given suitable tuning of the pipeline. We test our circuits inside a synthesis test harness to ensure an accurate timing analysis. We average the Quartus synthesis results over 10 random initial seeds, and tune Quartus to produce the highest-performing circuits possible. Appendices A and B describe the experimental framework and Quartus settings in detail.

3.6 Storage Architecture

We begin our exploration of FPGA-centric architecture by focusing on storage. Since modern mid/high-end FPGAs provide hard block RAMs (BRAMs) as part of the substrate, we assume that the storage system for our architecture will be composed of BRAMs. Since we are striving for a processor design of maximal frequency, we want

[2]An *octavo* is a booklet made from a printed page folded three times to produce eight leaves (16 pages).



(a) 398MHz (b) 656MHz (c) 531MHz (d) 710MHz

Figure 3.2: Self-loop characterization of memories to maximum unrestricted F_{max}

to know how the inclusion of BRAMs will impact the critical paths of our design. As already introduced, we use the method of *self-loop characterization*, where we simply connect the output of a component under study to its input, to isolate (i) operating frequency limitations and (ii) the impact of additional pipeline stages.

Figure 3.2 shows four 32-bit-wide memory configurations: 256-word memories using one BRAM with one (3.2(a)) and two (3.2(b)) pipeline stages, and 1024-word memories using four BRAMs with two (3.2(c)) and three (3.2(d)) pipeline stages. The result for a single BRAM (3.2(a)) is surprising: without additional pipelining, the F_{max} reaches only 398MHz out of a maximum of 550MHz (limited by the minimum-clock-pulse-width restrictions of the BRAM). This delay stems from a lack of direct connection between BRAMs and the surrounding logic fabric, forcing the use of global routing resources[3]. However, two pipeline stages (3.2(b), with register automatically retimed into BRAM output) increases F_{max} to 656MHz, and four pipeline stages (not shown) absorb nearly all delay and increase the achievable F_{max} up to 773MHz. Increasing the memory depth to 1024 words (3.2(c)) requires 4 BRAMs, additional routing, and some multiplexing logic—and reduces F_{max} to 531MHz. Adding a third pipeline stage (3.2(d)) absorbs the additional delay and increases F_{max} to 710MHz.

[3] In hindsight, a long t_{co} (Clock-to-Output time) delay may also contribute.

Table 3.1: Octavo's Instruction Word Format.

Size:	4 bits	a bits	a bits	a bits
Field:	Opcode (OP)	Destination (D)	Source (A)	Source (B)

These results indicate that pipelining provides significant timing slack for more complex memory designs. In Octavo, we exploit this slack to create a memory unit that collapses the usual register/cache/memory hierarchy into a single entity, maps all I/O as memory operations, and still operates at more than 550MHz. To avoid costly stalls on memory accesses, we organize on-chip memory as a single scratchpad [31] such that access to any external memory must be managed explicitly by software. Furthermore, since an FPGA-based processor typically implements both caches and register files out of BRAMs, we pursue the simplification of merging caches and register file into a single memory entity and address space. Hence Octavo can be viewed as either being (i) registerless, since there is only one memory entity for storage, or (ii) registers-only, since there are no load or store instructions, only operations that directly address the single operand storage.

3.7 Instruction Set Architecture

The single-storage-unit architecture decided in the previous section led to Octavo's instruction set architecture (ISA) having no loads or stores: each operand can address any location in the memory. Immediate values are implemented by placing them in memory and addressing them. Subroutine calls and indirect memory addressing are implemented by synthesizing code, explained in detail later in Section 3.12. Despite its frugality, we believe that the Octavo ISA can emulate the MIPS ISA[4].

Table 3.1 describes Octavo's instruction word format. The four most-significant bits hold the opcode, and the remaining bits encode two source operands (A and B) and a destination operand (D). The operands are all the same size (a address bits), and

[4]We emulate a MIPS-like ISA in Chapter 6, and also add hardware support for indirect memory.

Table 3.2: Octavo's Instruction Set and Opcode Encoding.

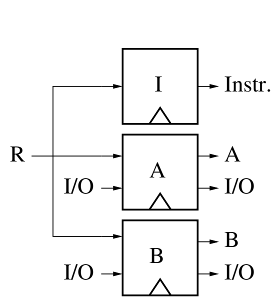
Mnemonic	Opcode	Action
	Logic Unit	
XOR	0000	$D \leftarrow A \oplus B$
AND	0001	$D \leftarrow A \wedge B$
OR	0010	$D \leftarrow A \vee B$
SUB	0011	$D \leftarrow A - B$
ADD	0100	$D \leftarrow A + B$
—	0101	*(Unused, for expansion)*
—	0110	*(Unused, for expansion)*
—	0111	*(Unused, for expansion)*
	Multiplier	
MHS	1000	$D \leftarrow A \cdot B$ (High Word Signed)
MLS	1001	$D \leftarrow A \cdot B$ (Low Word Signed)
MHU	1010	$D \leftarrow A \cdot B$ (High Word Unsigned)
	Controller	
JMP	1011	$PC \leftarrow D$
JZE	1100	if $(A = 0)$ $PC \leftarrow D$
JNZ	1101	if $(A \neq 0)$ $PC \leftarrow D$
JPO	1110	if $(A \geq 0)$ $PC \leftarrow D$
JNE	1111	if $(A < 0)$ $PC \leftarrow D$

the width of the operands dictates the amount of memory that Octavo can access. For example, a 36-bit instruction word has a 4-bit opcode, three 10-bit operand fields, and 2 bits unused—allowing for a memory space of 2^{10} (1024) words[5].

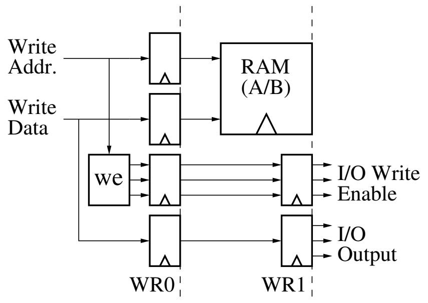
Table 3.2 shows Octavo's instruction set and opcode encoding, with eight opcodes allocated to Adder and Logic instructions, three for the Multiplier, and five allocated to control instructions, with the remaining three opcodes left for future expansion. The Logic Unit opcodes are chosen carefully so that they can be broken into sub-opcodes to minimize decoding in the ALU implementation. The Multiplier instructions return the High (MHS) and Low (MLS) word of the signed double-word product, and the unsigned High word (MHU) as the unsigned Low word computes identically to the signed version.

Note that later work in Chapter 6 will eliminate all control instructions, replacing them with more efficient programmable hardware modules, and freeing their five opcodes for future use.

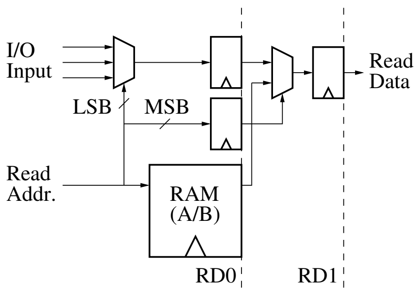
[5] We make use of the 2 unused bits in Chapter 5 to extend the Destination address space.



(a) All memories



(b) Writing A/B memories



(c) Reading A/B memories

Figure 3.3: Implementation of the A and B Memories with memory-mapped I/O ports.

3.8 Memory

Having decided the storage architecture and ISA for Octavo, we next describe the design and implementation of Octavo's memory unit. In particular, we describe the implementation of external I/O, and the composition of the different memory unit components.

I/O Support Having only a single memory/storage and no separate register file eliminates the notion of loads and stores, which normally implement memory-mapped I/O mechanisms. Since significant timing slack exists between the possible and actual F_{max} of FPGA BRAMs, we can use this slack to memory-map I/O mechanisms without impacting our high clock frequency. We map word-wide I/O lines to the uppermost memory locations (typically 2 to 8 locations), making them appear like ordinary memory and thus accessible like any operand. We interpose the I/O ports in front of the RAM read and write ports: the I/O read ports override the RAM read if the read address is in the I/O address range, while the I/O write ports pass through the write address and data to the RAM. This architecture provides interesting possibilities for future multicore arrangements of Octavo: any instruction can now perform up to two I/O reads and one I/O write simultaneously; also, an instruction can write its result directly to an I/O port and another instruction in another CPU can directly read it as an operand.

Implementation Figure 3.3 shows the connections of Octavo's memory units and details the construction of the A and B Memories. Each memory behaves as a simple dual-port (one read and one write) memory, receiving a common write value R (the ALU's result), but keeping separate read and I/O ports (Figure 3.3(a)). Thus, the I, A, and B Memories all hold identical contents[6]. The I Memory contains only BRAMs, while the A and B Memories additionally integrate a number of memory-mapped word-wide I/O ports (typically two or four). For the A and B memories, reads or writes take 2 cycles each but overlap for only 1 at $RD0/WR1$. A write (Figure 3.3(b)) spends its first cycle registering the address and data to RAM and activating one of the I/O write port write-enable lines based on the write address. The data write to the RAM and to all I/O write ports occurs during the second cycle.[7] A read (Figure 3.3(c)) sends its address to the RAM during the first cycle and simultaneously selects an I/O read port based on the Least-Significant Bits (LSB) of the address. Based on the remaining Most-Significant Bits (MSB) of the address, the second read cycle returns either the data from the RAM or from the selected I/O read port. Our experiments showed that we can add up to about eight I/O ports per RAM read/write port pair before the average operating speed drops below 550MHz.

[6] We eliminate this wasteful data duplication in Chapter 5.

[7] We implemented the RAM using Quartus' auto-generated BRAM write-forwarding circuitry, which immediately forwards the write data to the read port if the addresses match. This configuration yields a higher F_{max} since there is a frequency cost to the Stratix IV implementation of BRAMs set to return old data during simultaneous read/write of the same location [10]. However, since pipelining delays the write to a BRAM by one cycle, a coincident read will return the data currently contained in the BRAM instead of the data being written.

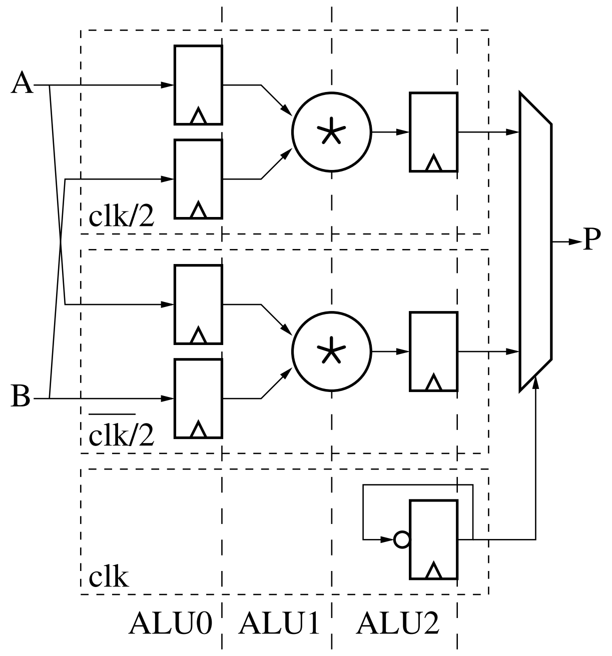


Figure 3.4: Detailed view of the Multiplier unit

3.9 ALU

In this section we describe the development and design of Octavo's ALU components, including the Multiplier, the Adder/Subtractor, the Logic Unit, and their combination to form the ALU.

Multiplier Unit To support multiplication for a high-performance soft-processor it is necessary to target the available DSP block multipliers. Although Stratix IV DSP blocks have a sufficiently-low propagation delay to meet our 550MHz target frequency, they have a minimum-clock-pulse-width limitation (similar to BRAMs) restricting their operating frequency to 480MHz for word-widths beyond 18 bits [8].

Figure 3.4 shows the internal structure of Octavo's Multiplier and our solution to the

[8]For widths ≤18 bits, it might be possible to implement the multiplier with a single DSP block, but current CAD issues prevent getting results consistent with the published specifications [10] for high-frequency implementations.

clocking limitation: we use two word-wide DSP block multipliers[9] in alternation on a PLL-generated synchronous half-rate clock ($clk/2$), such that we can perform two independent word-wide multiplications, staggered but in parallel, and produce one double-word product every cycle. In detail, the operands A and B are de-multiplexed into the two half-rate datapaths on alternate edges of the half-rate clock. A single state bit driven by the system clock (clk) selects the correct double-word product (P) at each cycle.

Adder/Subtractor We also carefully and thoroughly studied adder/subtractors and logic units while building Octavo, again using the method of self-loop characterization described in Section 3.6. We experimentally found that an unpipelined 32-bit ripple-carry adder/subtractor can reach 506MHz, and that adding 4 pipeline stages increases F_{max} up to 730MHz. An unpipelined carry-select implementation only reaches 509MHz due to the additional multiplexing delay, but requires only two stages to reach 766MHz. Due to the 550MHz limitation imposed by BRAMs, a simple two-stage ripple-carry adder reaching 600MHz is sufficient.

Logic Unit The Logic Unit performs bit-wise `XOR`, `AND`, and `OR` operations (Table 3.2). It also acts as a pass-through for the result of the Adder/Subtractor, which avoids an explicit multiplexer and allows us to separate and control the implementation of the Adder/Subtractor from that of the Logic Unit. The Logic Unit efficiently maps to a single ALUT (Adaptive Look-Up Table) per word bit: 3 bits for the opcode, plus one bit from the Adder/Subtractor result, and 2 bits for the A and B operands of the bit-wise operations, totalling 6 bits and naturally mapping to a single Stratix IV 6-LUT per output bit.

[9] We implement each multiplier using an LPM instance generated by the Quartus MegaWizard utility. Although the Altera DSP blocks have input, intermediate, and output registers, a designer can only specify the desired number of pipeline stages that begin at the input to the DSP block—hence we cannot specify to use only the input and output registers to absorb the delay of the entire DSP block. We bypass this limitation by instantiating a one-stage-pipelined multiplier and feeding its output into external registers. Later register-retiming optimizations eventually place these external registers into the built-in output registers of the DSP block, yielding a two-stage pipelined multiplier with only input and output registers.

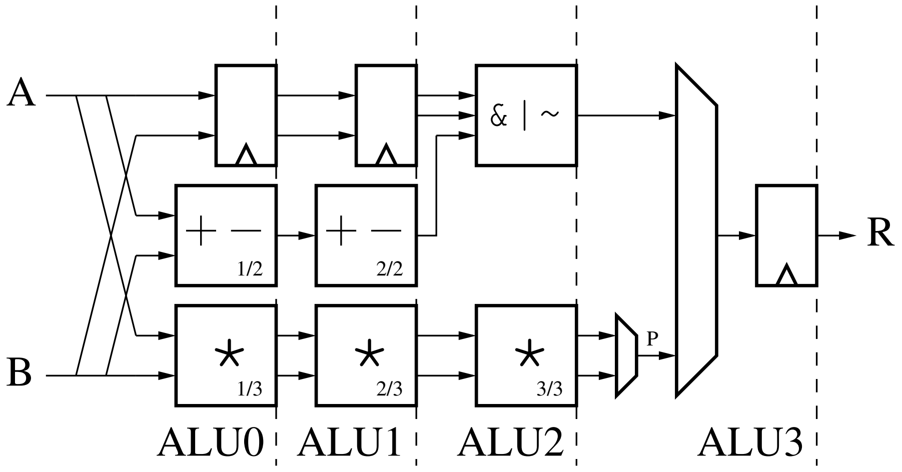


Figure 3.5: Organization of Octavo's ALU

Combined ALU Design Figure 3.5 shows the block-level structure of the entire ALU, which combines the Multiplier ($*$), Adder/Subtractor (`+-`), and Logic Unit (`&|~`). All operations occur simultaneously during each cycle, with the correct result selected by the output multiplexer after four cycles of latency. We optimized each sub-component for speed, then added extra pipeline registers to balance the path lengths. We use the Logic Unit as a pipeline stage and multiplexer to reduce the delay and width of the final ALU result multiplexer. The combined ALU runs at an average of 595MHz for a width of 36 bits.

3.10 Controller

Figure 3.6 shows the design of the Octavo Controller. The Controller provides the current Program Counter (*PC*) value for each thread of execution and implements flow-control. A Program Counter Memory (*PCM*) holds the next value of the PC for each thread of execution. We implement the *PCM* using one MLAB[10] instead of a BRAM, given a typically narrow PC (< 20 bits) and a relatively small number of threads (nominally 8, but up to 16)—this also helps improve the resource-diversity of Octavo and will ease its

[10]Memory Logic Array Blocks (MLABs) are small (e.g., 32 words deep by 20 bits wide) memories found in Altera FPGAs.

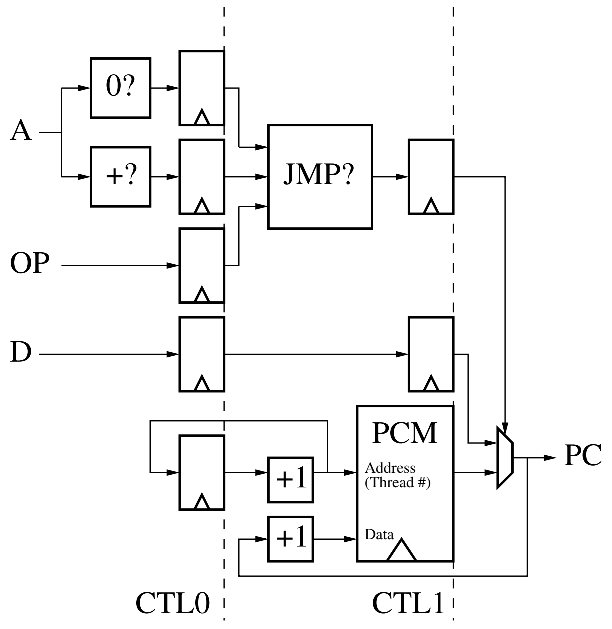


Figure 3.6: The multi-threaded Controller, which implements flow-control.

replication in future multicore designs. A simple incrementer and register pair perform round-robin reads of the *PCM*, selecting each thread in turn. At each cycle, the current *PC* of a thread is incremented by one and stored back into the *PCM*. The current *PC* is either the next consecutive value from the *PCM*, or a new jump destination address from the *D* instruction operand.

The decision to output a new *PC* in the case of a jump instruction is based on the instruction opcode *OP* and the fetched value of operand *A*. A two-cycle pipeline determines if the value of *A* is zero (0?) or positive (+?), and based on the opcode *OP* decides whether a jump in flow-control happens (*JMP*?)—i.e., outputs the new value of the PC from *D*, instead of the next consecutive value from the *PCM*. A Controller supporting 10-bit PCs for 8 threads can reach an average speed of 618MHz, though the MLAB implementing the PCM limits F_{max} to 600MHz.

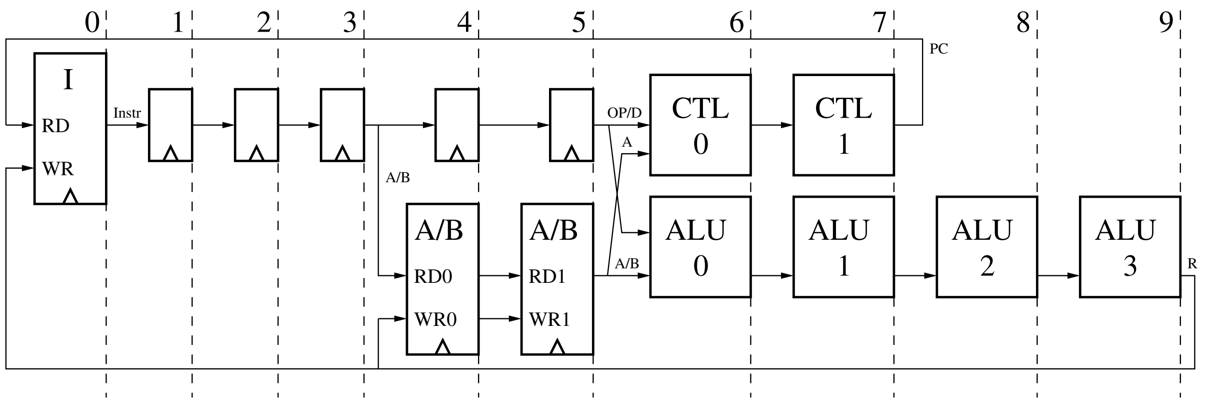


Figure 3.7: The complete multi-threaded Octavo system

3.11 Complete Octavo Hardware

In this section, we combine the units described in the previous three sections to build the complete Octavo datapath shown in Figure 3.7, composed of an instruction Memory (I), two data Memories (A and B), an ALU, and a Controller (CTL).

We begin by describing the Octavo pipeline from left to right. In Stage 0, Memory I is indexed by the current PC and provides the current instruction containing operand addresses D, A, and B, and the opcode OP. Stages 1-3 contain only registers and perform no computation. Their purpose is to separate the BRAMs of Memory I from those of Memories A/B by a suitable number of stages to maintain a 550MHz clock: as shown by the self-loop characterization in Section 3.6, we must separate groups of BRAMs with at least two stages—having only a single extra stage between the I and A/B memories would yield an F_{max} of only 495MHz for a 36-bit, 1024-word Octavo instance.

We insert three stages to avoid having an odd total number of stages. Across stages 4 and 5, the A and B memories provide the source operands (of the same name). The ALU spans stages 6-9 and provides the result R which is written back at address D to both Memories A and B (across stages 4 and 5 again), as well as Memory I (at stage 0). The Controller (CTL) spans stages 6 and 7 and writes the new PC back to Memory I in stage 0. The controller contains the PC memory for all threads, and for each thread decides whether (i) to continue with the next consecutive PC value, or (ii) to branch to the new target address D.

There are three main hazards/loops in the Octavo pipeline. The first hazard exists in the control loop that spans stages 0-7 through the controller (CTL)—hence Octavo requires a minimum of eight independent threads to hide this dependence. The second hazard is the potential eight-cycle Read-After-Write (RAW) data hazard between consecutive instructions from the same thread: from operand reads in stages 4-5, through the ALU stages 6-9, and the write-back of the result R through stages 4-5 again (recall that writing memories A/B also takes two stages)—this dependence is also hidden by eight threads. The third hazard also begins at the operand reads in stages 4-5 and goes through the ALU in stages 6-9, but writes-back the result R to Memory I for the purpose of the instruction synthesis introduced in Section 3.7 and described in detail in the next section. This loop spans ten stages and is thus not covered by only eight threads. Rather than increase thread contexts beyond eight to tolerate this loop, we instead require a *delay slot* instruction between the synthesis of an instruction and its use.

3.12 Octavo Software

As described in Section 3.7, the Octavo ISA supports only register-direct addressing, since all operands are simple memory addresses—hence the implementation of displacement, indirect, or indexed addressing requires two instructions: a first instruction reads the memory location containing the indirect address or the displacement/index, and stores it into the source or destination operand of a second instruction that performs the actual memory access using the modified operand address. The remainder of this section provides examples of indirection implemented using the Octavo ISA, including pointer dereference, arrays, and subroutine calls.

```
int a = 42;
int c = 88;
int *b = &c;
...
a = *b;
```

(a) C code

```
a: 42
c: 88
b: c
   ...
la  r1, a
la  r2, c
   ...
lw   r3, 0(r2)
sw   r3, 0(r1)
```

(b) Optimized MIPS code

```
Z: 0
a: 42
c: 88
b: c
...
   or  T, T, b
   nop
T: add a, Z, 0
```

(c) Octavo code (pre-execution)

```
Z: 0
a: 88
b: c
c: 88
...
   or  T, T, b
   nop
T: add a, Z, c
```

(d) Octavo code (post-execution)

Figure 3.8: Code Synthesis: Pointer dereference example.

Later, in Chapter 6, we convert these techniques for memory indirection and array traversal into hardware operating in parallel with the software. We leave hardware subroutine calls as Further Work in Chapter 8.

3.12.1 Pointer Dereference

The C code in Figure 3.8(a) performs an indirect memory access by dereferencing the pointer *b* and storing the final value into location *a*. In the MIPS ISA (Figure 3.8(b)), this code translates into a pair of address loads (we use the common 'la' assembler macro for brevity) followed by a displacement addressing load/store pair. Since the value of *b* is known at compile time, we assume that the compiler optimizes-away the dereference and uses the address of *c* directly.

In the Octavo ISA we synthesize indirect addressing at run-time by placing the address stored in *b* into a source operand of a later instruction that stores into *a* the *contents* of the address taken from b. Without load/store operations, we instead use an `ADD` with “register zero” as one of the operands. Figure 3.8(c) shows the initial conditions of the Octavo code and begins with a memory location defined as “register zero” (Z) and others containing the same initialized variables (a, b, and c) as the C code. Line 6 contains an instruction that `OR`’s a target instruction T (line 8) with the contents of b (line 3)—note that T’s second source operand initially contains zero. A `NOP` or other independent instruction must exist between the generating instruction and its target due to the 1-cycle RAW hazard when writing to Memory I (Section 3.11) if executing less than 10 threads. Figure 3.8(d) shows the result of executing from line 6 onwards, that replaces the zero source operand in T with the contents of b, and later executes T with the modified operand, storing the contents of c into a. If the compiler knows the value of the pointer b, it can perform these steps at compile-time and synthesize the final instruction—avoiding the run-time overhead. To traverse a linked list or any other pointer-based structure, the target instruction T instead can update the pointer b itself.

3.12.2 Iterating over Arrays

Despite the apparent inefficiency of needing to synthesize code to perform indirect memory accesses, manipulating the operands of an instruction can also have advantages. For example, the C code in Figure 3.9(a) describes the core of a loop summing two arrays. Figure 3.9(b) shows a straightforward translation to MIPS assembly: the same letters as in the `C` code denote consecutive array locations. After a 3-instruction preamble to load the array addresses into registers $r1$, $r2$, and $r3$, the next four instructions (lines 12-15) load the B and C array element values, sum them, and store them back into the corresponding A element. The last three instructions increment the array pointers.

The equivalent Octavo assembly code in Figure 3.9(c) works in the same way, but using synthesized code: after directly performing the array element sum at T on line 9, we add 1 to each address operand using a word-wide value I on line 7. This increment

```
int A[] = { 42, ...};
int B[] = { 23, ...};
int C[] = { 88, ...};
...
*A = *B + *C;
A++;
B++;
C++;
```

(a) C code

```
A : 42
A': ...
B : 23
B': ...
C : 88
C': ...
    ...
la  r1, A
la  r2, B
la  r3, C
    ...
lw   r5, 0(r2)
lw   r6, 0(r3)
add  r4, r5, r6
sw   r4, 0(r1)
addi r1, r1, 1
addi r2, r2, 1
addi r3, r3, 1
```

(b) MIPS code

```
A : 42
A': ...
B : 23
B': ...
C : 88
C': ...
I :    0 1, 1, 1
    ...
T : add A, B, C
    add T, T, I
```

(c) Octavo code (pre-execution)

```
A : 111
A': ...
B : 23
B': ...
C : 88
C': ...
I :    0 1, 1, 1
    ...
T : add A', B', C'
    add T, T, I
```

(d) Octavo code (post-execution)

Figure 3.9: Code Synthesis: Array access example.

value I contains the increment of each array pointer, each shifted to align with the corresponding address field, and a zero value aligned with the opcode field. Adding I to T yields the updated code for the next loop iteration in Figure 3.9(d). Compared to the MIPS code in Figure 3.9(b), Octavo requires only two instructions instead of seven to compute the same loop body.

```
 1 Z:  0
 2 RET1:  jmp X,  0,  0
 3 RET2:  jmp R,  0,  0
 4
 5 sub:
 6        ...
 7 E:     jmp X,  0,  0
 8
 9 caller:
10        ...
11        add E,  Z,  RET2
12        jmp sub,  0,  0
13 R:     ...
```

(a) pre-execution

```
 1 Z:  0
 2 RET1:  jmp X,  0,  0
 3 RET2:  jmp R,  0,  0
 4
 5 sub:
 6        ...
 7 E:     jmp R,  0,  0
 8
 9 caller:
10        ...
11        add E,  Z,  RET2
12        jmp sub,  0,  0
13 R:     ...
```

(b) post-execution

Figure 3.10: Code Synthesis: Call/return example.

Synthesized code does however increase the size of loop preambles. Octavo's loop preamble overhead could become significant with many short nested loops, but compiler optimizations such as loop coalescing would reduce it. Similarly, induction variable elimination would reduce the amount of synthesized code required for more complex array access patterns.

3.12.3 Synthesizing Subroutine Calls

Without call stack hardware support, Octavo must synthesize code to implement subroutine linkage using a method previously described by Knuth [67]. While somewhat awkward, having to synthesize CALL and RET instructions saves two scarce opcodes for other uses and enables conditional calls and returns at no extra cost.

Figure 3.10(a) shows a synthesized CALL and RET pair example. Lines 2 and 3 contain return jumps $RET1$ and $RET2$ that act as the "RET" for specific "CALLs" to *sub* (lines 5-7). These return jumps get placed by callers at the exit point E of *sub*, that currently contains a copy of $RET1$ placed there by a previous caller. Before jumping to *sub* at line 12, the *caller* will change *sub*'s return jump target from X to R, the return point in the *caller* at line 13. Figure 3.10(b) shows the updated code after line 11 executes, with the exit point E updated to return to R. Using JNZ, JZE, JPO, or JNE instead of JMP at line 3 implements a conditional subroutine return. Doing the same at line 12 implements a conditional subroutine call.

Because this linkage scheme stores the return jump target address within the subroutine code, it does not allow a second call to a subroutine before the first call returns. Within a single thread, recursive subroutines must be converted to iterative ones. Across different threads, we must create thread-private copies of the subroutine. We outline a basic solution for true subroutine calls as Further Work in Chapter 8.

3.13 Programming Model

To create programs that use Octavo most effectively, we must take into account Octavo's pipelined and multi-threaded nature. Although Octavo's strict round-robin multithreading (i.e.: without variation in thread order) may initially appear rigid, it allows us to easily compose smaller threads together against larger problems. Similarly, the absence of stalls in the pipeline and the minimal RAW hazards allow us to easily keep the pipeline fully utilized. We outline here some of the major features of Octavo's pipelined and multi-threaded programming model, and will update them as this work progresses in later chapters.

3.13.1 Unrolling Code

Within a thread, consecutive instructions usually suffer no RAW hazards. Each thread instruction can immediately use the result of the previous one. Exceptions arise when a thread instruction writes outside of the ALU datapath to the Instruction memory [11]. These writes take effect after one thread cycle. Reordering instructions usually fills the

[11] And in Chapter 6, to the AOM and BTM.

resulting delay slot, and we can unroll sequences of instructions with such RAW hazards to keep the pipeline busy. For example, Listing 8.1 (pg. 161) presents a simple interpreter which unrolls its writes to Instruction memory to avoid having to fill the delay slot with a no-op. When programming an Octavo thread, we should unroll loops containing RAW hazards as necessary to avoid no-ops, especially since Octavo favours small, tight loops with little instruction re-ordering possible.

3.13.2 Pipelined Data-Parallel Multi-Threading

If an application has any regular data parallelism (e.g.: arrays), we can divide the work across multiple Octavo threads. We should first solve the problem as if using a single thread. We can then calculate the appropriate addresses in multiple copies of the single thread code to divide the data into multiple pieces, one per thread, and run the customized single-threaded solutions in each thread concurrently, filling Octavo's pipeline. Although this programming model resembles current multi-threaded and multi-core GPU (Graphics Processing Unit) programming (e.g.: OpenCL), Octavo's threads operate independently, without penalty for diverging control flow, as would occur on a GPU with code containing conditional expressions.

However, this amplification by multi-threading also multiplies the cost of branching and addressing. When all threads operate in a pseudo-SIMD manner, by executing identical code in lock-step but on different data, then each branch also gets repeated by the number of threads, effectively causing a large bubble in the pipeline. The cost of addressing similarly increases: each thread requires a private copy of the code in Instruction memory, and must each repeat the address calculations within each copy. Nonetheless, we manage to eliminate these overheads in Chapter 6 with some hardware improvements.

3.13.3 Converting ILP to Multi-Threading

At the limit, if we cannot express a solution directly as multi-threaded parallelism, we can schedule independent instructions across multiple Octavo threads to emulate superscalar execution and increase the usage of the pipeline beyond a single Octavo thread. For example, if we can apply loop transformations to enable loop unrolling and increase Instruction-Level Parallelism (ILP), we can create as many independent "threads" of execution as we need. Briefly put, we then schedule the new independent instructions into consecutive Octavo threads. If an instruction depends on a previous one, we must schedule it either as the next instruction in the same thread, or at the same point in a following thread, to ensure the previous instruction has completed. Once we have scheduled all the possible ILP, we can then software-pipeline where applicable to fill empty slots in the used Octavo threads, or to replicate the process across more Octavo threads (e.g.: 2 groups of 2 "superscalar" Octavo threads).

3.13.4 Difficulties: Synchronization

While threads may diverge in control-flow without problem, we cannot perform a subsequent "*rendez-vous*" across threads since the memory reads and writes of thread instructions are not atomic across threads. Eight clock cycles pass between a thread instructions' source operand reads and its destination operand write, during which other threads may read or write the same memory locations. Thus, a thread cannot implement an atomic read-modify-write action upon which we could build some synchronization or exclusion primitives such as `test-and-set` or `compare-and-swap`. We will need to use a hardware I/O handshaking mechanism, introduced in Chapter 5, to construct atomic read and/or write primitives (Section 5.3).

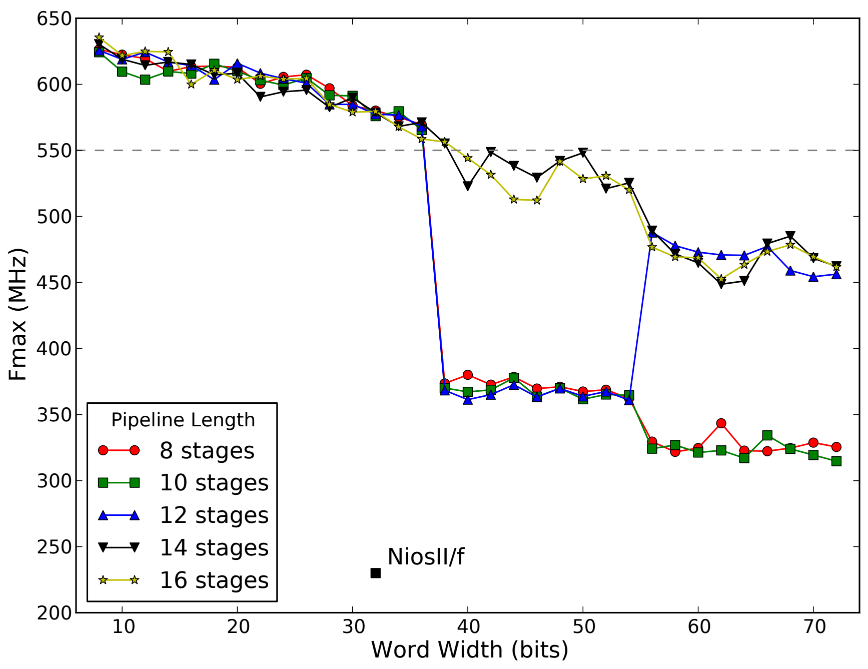


Figure 3.11: Octavo maximum operating frequency vs. word width

3.14 Speed and Area

In this section, we examine many varying instances of Octavo as instantiated on a Stratix IV `EP4SE230F29C2` device. In particular we measure maximum operating frequency (F_{max}), area usage, and area density over a range of configurations, varying word width, memory depth, and number of pipeline stages. We perform these experiments to confirm that Octavo achieves our stated goals for a processor design (Section 3.1) over a wide range of configurations.

3.14.1 Maximum Operating Frequency

Our first experiments address whether Octavo's high F_{max} will hold for non-trivial and unconventional word widths and increasing memory depths. We find that, over a range of word widths from 8 to 72 bits, F_{max} remains high and degrades smoothly.

Figure 3.11 shows the maximum operating frequency F_{max} of Octavo for word widths ranging from 8 to 72 bits, and for Octavo instances with 8 to 16 pipeline stages. The

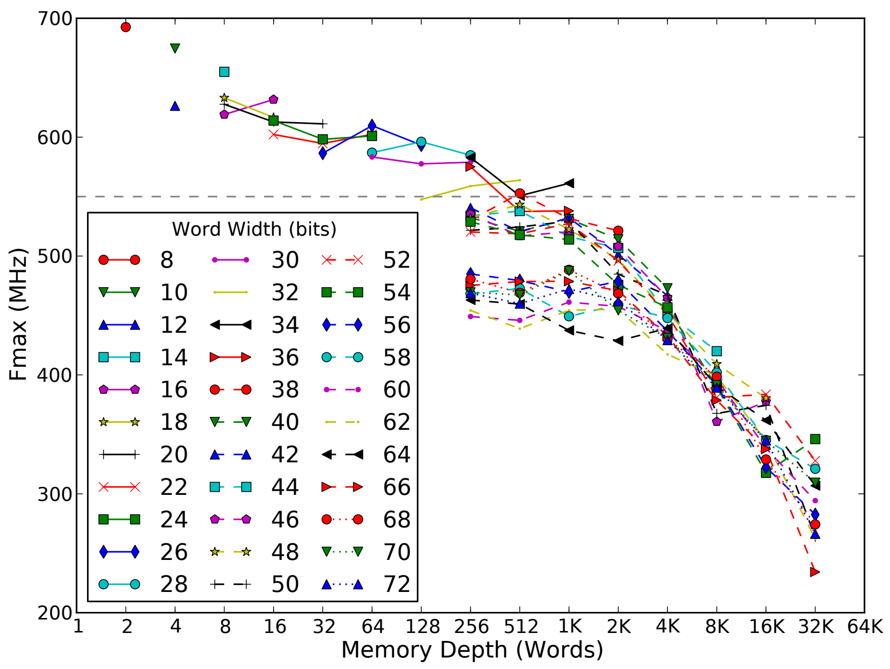


Figure 3.12: Octavo maximum operating frequency vs. memory depths and word widths

dashed line indicates the 550MHz F_{max} upper limit imposed by the BRAMs. As a rough comparison we plot the 32-bit NiosII/f soft-processor, reported to be 230MHz for our target FPGA [11]. For this experiment we limited memory depth to a maximum of 256 words so that each memory fits into a single BRAM, avoiding any effect on F_{max} from memory size and layout.

For all pipeline depths, F_{max} degrades slowly from about 625MHz down to 565MHz when varying word width from 8 to 36 bits. For 12 to 16 pipeline stages F_{max} decreases only 28% over a 9x increase in width from 8 to 72 bits, and still reaches just over 450MHz at 72 bits width. Word widths beyond 36 bits exceed the native capacity of the DSP blocks, requiring additional adders (implemented with ALUTs) to tie together multiple DSP blocks into wider multipliers. Adding more pipeline stages to the Multiplier absorbs the delay of these extra adders but increases total pipeline depth. Increasing pipeline depth by 4 stages up to 12 absorbs the delay of these extra adders.

Unfortunately a CAD anomaly occurs for widths between 38 and 54 bits (inclusive), where Quartus 10.1 cannot fully map the Multiplier onto the DSP blocks, forcing the use of yet more adders implemented in FPGA logic. Increasing the pipelining to 14 stages, again by adding stages in the Multiplier, overcomes the CAD anomaly. Increasing the pipelining to 16 stages has no further effect on Octavo, whose critical path lies inside the Multiplier. The CAD anomaly affects Octavo in two ways: the affected word-widths must pipeline the Multiplier further than normally necessary to overcome the extra adder delay, and also show a discontinuously higher F_{max} than the wider, unaffected word-widths (56 to 72 bits), regardless of the number of pipeline stages. Unfortunately this CAD anomaly hides the actual behaviour of Octavo at the interesting transition point at widths of 36 to 38 bits, where the native width of both BRAMs and DSP blocks is exceeded.

Figure 3.12 shows the maximum operating frequency (F_{max}) for a *16-stage* Octavo design over addressable memory depths ranging from 2 to 32,768 words and plotted for word widths from 8 to 72 bits. We also mark the 550MHz actual F_{max} upper limit imposed by the BRAMs. We use 16 stages instead of 8 to avoid the drop in performance caused by the CAD anomaly.

The previously observed discontinuous F_{max} drop in Figure 3.11 for Octavo instances with widths of 56 to 72 bits is visible here in the cluster of dashed and dotted lines lying below 500MHz for depths of 256 to 4096 words. Similarly, the cluster of dashed lines above 500MHz spanning 256 to 4096 words depth contains the word widths (38 to 54 bits) affected by the CAD anomaly.

A memory requires twice as many BRAMs to implement widths exceeding the native BRAM maximum width of 36 bits. Unfortunately, the CAD anomaly masks the initial effect on F_{max} of doubling the number of BRAMs for the same depth when exceeding a word width of 36 bits.

For depths up to 256 words, which all fit in a single BRAM, and widths below where the CAD anomaly manifests (8 to 36 bits), F_{max} decreases from 692MHz down to 575MHz, a 16.9% decrease over a 4.5x increase in word width and 128x increase in memory depth (2 to 256 words). For depths greater than 256 words, if we take as example the narrowest width (50 bits) which can address up to 32,768 words, F_{max} decreases

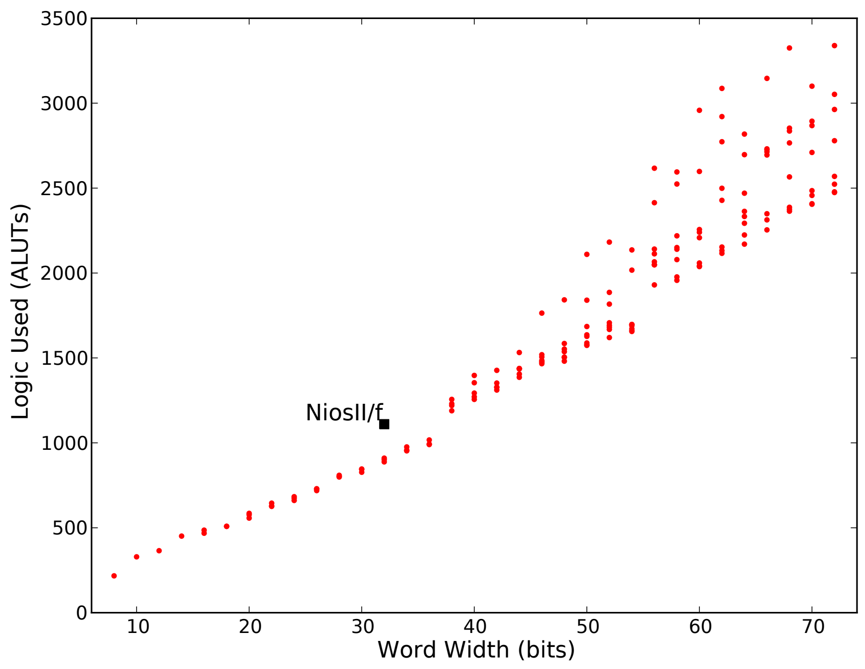


Figure 3.13: Octavo area vs. memory depths and word widths

49.8% over a 64x increase in depth (512 to 32,768 words). The decrease changes little as width increases: 42.1% at 72 bits width over the same memory depths. Overall, an increase in memory depth affects F_{max} much more than an increase in width, with the effect becoming noticeable past 1024 words of depth.

Summary We summarize with two main observations: (i) widths $>$ 36 bits require additional logic and pipelining, and (ii) a CAD anomaly forces longer pipelines and hides the actual curves for less than 14 pipeline stages. We also found that at least 12 pipeline stages are necessary for widths greater than 56 bits, modulo the CAD anomaly, and that memory depth has a greater effect on F_{max} than word width, becoming significant beyond 1024 words.

3.14.2 Area Usage

Our next experiments tests if Octavo's area scales practically as word width and memory depth increase. Figure 3.13 shows the area used in ALUTs (Adaptive Look-Up Tables), excluding BRAMs and DSP blocks, over word widths ranging from 8 to 72 bits, for an *8-stage* Octavo design. Where possible, for each width, we plot multiple points, each representing an addressable power-of-2 memory depth ranging from 2 to 32,768 words. We also mark the reported 1,110 ALUT area usage of the 32-bit NiosII/f soft-processor on the same FPGA family [11].

For small memories having less than 256 words, the area used varies roughly linearly, increasing 11.4x in area over a 9x increase in width. The CAD anomaly causes two small discontinuous increases in the ALUT usage: +24.2% while increasing from 36 to 38 bits width, and +16.5% from 54 to 56 bits, both cases for a memory depth of 256 words. Increasing memory depth has little effect on the amount of logic used: at a width of 72 bits, the area increases from 2478 to 3339 ALUTs (+37.5%) when increasing the memory depth from 256 to 32,768 (128x).

Summary We found that area varies roughly linearly with word width, varies little with memory depth, and is also affected by the CAD anomaly.

3.14.3 Density

Our final experiments seek to find if some Octavo configurations are "denser" than others, leaving fewer ALUTs (Adaptive Look-Up Tables), BRAMs, or DSP blocks unused within their rectangular area (a *LogicLock* floorplan automatically determined by Quartus). Figure 3.14 shows the density, measured as the percentage of ALUTs in actual use within the rectangular area containing an *8-stage* Octavo instance, over word widths ranging from 8 to 72 bits and plotted for each addressable memory depth ranging from 2 to 32,768 words. BRAMs and DSP block do not count towards ALUT count. Word width has no clear effect, but density drops sharply for depths exceeding 1024 words due to the BRAM columns needing a larger rectangular area to contain them than would compactly contain the processor logic implemented using ALUTs.

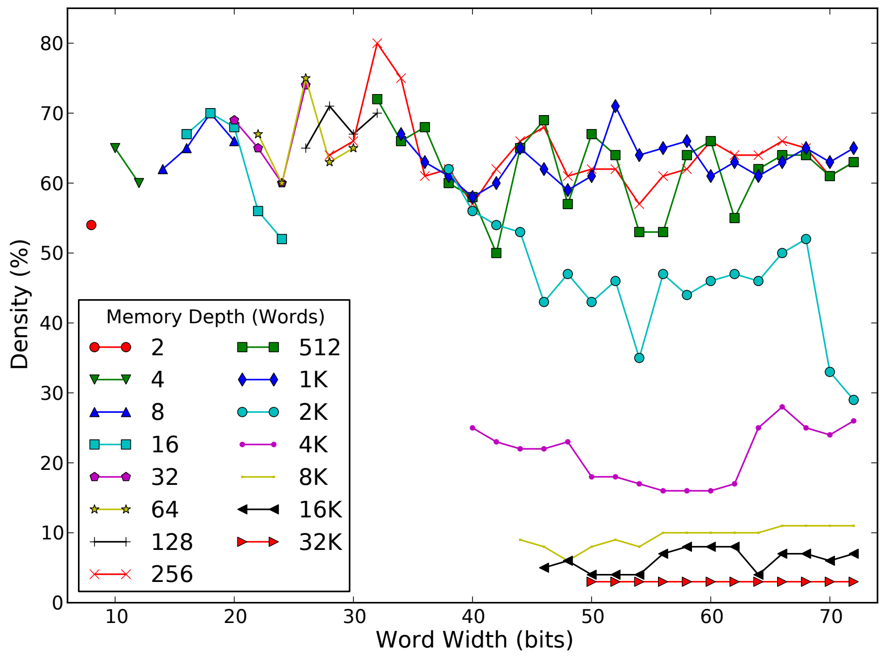
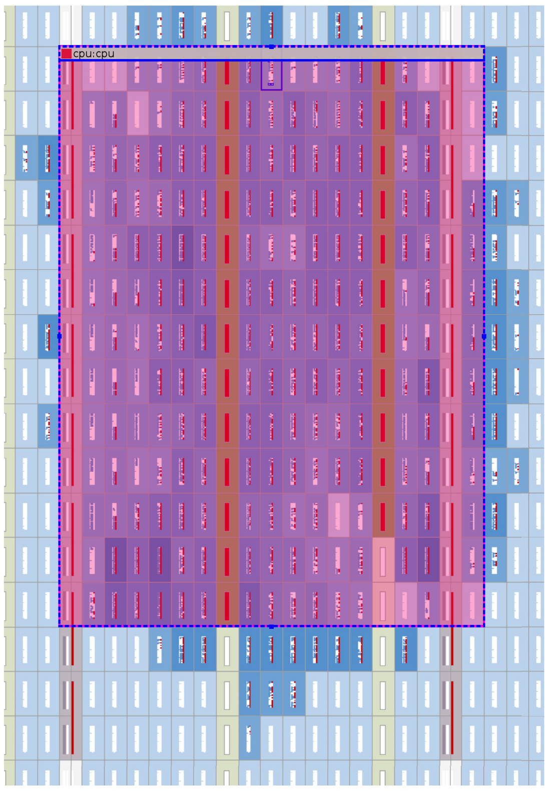


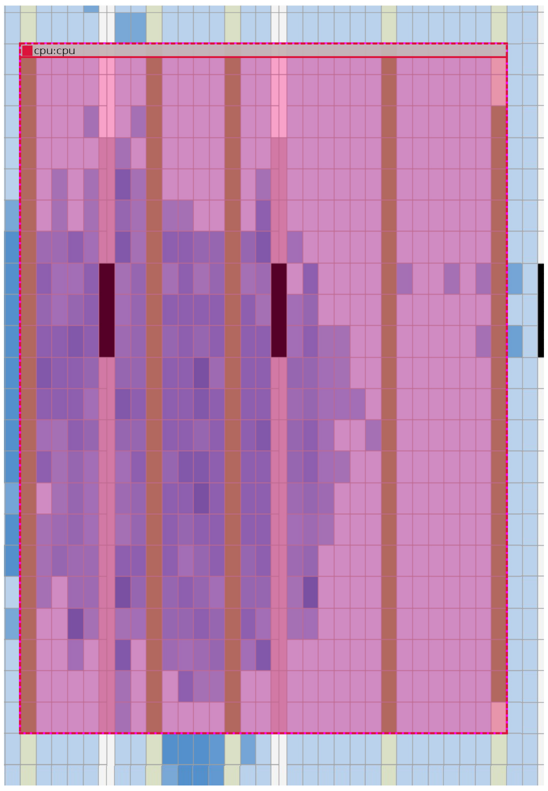
Figure 3.14: Octavo density vs. memory depths and word widths

Figures 3.15(a) and 3.15(b) illustrate the effect of the layout of BRAMs on the density. Each show an 8-stage, 72-bit wide Octavo instance with a memory of 1024 and 4096 words respectively. The large shaded rectangular area contains only the ALUTs used by Octavo: any outside ALUTs belong to a test harness and do not count; the darker columns contain the BRAMs implementing the Memory; the pale columns contain DSP blocks implementing the Multiplier and are part of Octavo despite protruding below the rectangular area in one instance; the remaining small blocks denote groups of ALUTs, with shade indicating the relative number of ALUTs used in each group. When increasing from a 1024 to 4096 word memory, the number of ALUTs used to implement Octavo increases only 15.3%, but the density drops from 65% to 26% due to the unused ALUTs enclosed by the required number of BRAMs.

For memories deeper than 1024 words, we could recover the wasted ALUTs by allowing non-Octavo circuitry to be placed within its enclosing rectangular area, but this choice may negatively affect F_{max} due to increased routing congestion. Further work may



(a) 1024 Words



(b) 4096 Words

Figure 3.15: Physical layout of an 8-stage, 72-bit wide Octavo instance with (a) 1024 and (b) 4096 memory words.

lead us to create vector/SIMD versions of Octavo to reclaim unused resources[12].

Summary Our experiments confirm our original intuition that there exists a "sweet spot"—where the number of BRAMs used fits most effectively within the area of the CPU—at approximately 1024 words of memory depth, regardless of word width.

[12] We create SIMD and many-core Octavo systems in Chapter 4.

3.15 Conclusions

In this chapter we presented initial work to answer the question "How do FPGAs want to compute?", resulting in the Octavo FPGA-centric soft-processor architecture family. Octavo is a ten-pipeline-stage, eight-threaded processor that operates at the BRAM maximum of 550MHz on a Stratix IV FPGA, is highly parameterizable, and behaves well under a wide range of datapath and memory width, memory depth, and number of supported thread contexts:

- F_{max} decreases only 28% (625 to 450MHz) over a 9x increase in word width (8 to 72 bits);
- F_{max} decreases 49.8% over a 64x increase in memory depth (512 to 32k words), and almost independently of word width;
- the amount of logic used is almost unaffected by memory depth: at a width of 72 bits, the usage increases from 2478 to 3339 ALUTs (+37.5%) when increasing the memory depth from 256 to 32,768 (128x);
- the amount of logic used varies roughly linearly with word width, increasing 11.4x over a 9x increase in width (8 to 72 bits);
- and the area density is unaffected by word width, but drops sharply for memory depths exceeding 1024 words due to the BRAM columns needing a larger containing rectangular area than that required for the processor logic.

Chapter 4

Tiling Overlay Architectures

Note that for these ultrahigh-speed designs, properly positioning the data takes more CLBs than performing the computation. This observation suggests that wiring speed has more importance than LUT speed for ultrahigh-performance designs.

Brian Von Herzen [56]

Common practice for large FPGA design projects divides sub-projects into separate synthesis partitions to allow incremental recompilation as each sub-project evolves. In contrast, smaller design projects avoid partitioning to give the CAD tool the freedom to perform as many global optimizations as possible, knowing that the optimizations normally improve performance and possibly area.

In this chapter, we show that for high-speed tiled designs composed of duplicated components and hence having *multi-localities* (multiple instances of equivalent logic), a designer can use partitioning to preserve multi-locality and improve performance. In particular, we focus on the lanes of SIMD soft-processors and multicore meshes composed of them, as compiled by Quartus 12.1 targeting a Stratix IV `EP4SE230F29C2` device.

We demonstrate that, with negligible impact on compile time (less than ±10%):

- we can use partitioning to provide high-level information to the CAD tool about preserving multi-localities in a design, without low-level micro-managing of the design description or CAD tool settings;

- by preserving multi-localities within SIMD soft-processors, we can increase both frequency (by up to 31%) and compute density (by up to 15%);
- partitioning improves the density and speed (by up to 51 and 54%) of a mesh of soft-processors, across many building block configurations and mesh geometries;
- the improvements from partitioning increase as the number of tiled computing elements (SIMD lanes or mesh nodes) increases.

As an example of the benefits of partitioning, a mesh of 102 scalar soft-processors improves its average operating frequency from 284 up to 437 MHz, its performance from 28,968 up to 44,574 MIPS, while increasing its logic area by only 0.85%.

This work originally appeared at ICFPT 2013, Kyoto [78].

4.1 Introduction

Soft overlay architectures can ease design challenges on FPGAs by making them more software-programmable. Examples of such systems include VESPA [130], VEGAS [31], VENICE [106], iDEA [26, 27], Octavo [77], and others [14, 73, 119, 135, 137]. In general, overlays provide parallelism through "tiling" (duplicating in two dimensions) computing elements such as datapaths and soft-processors. Our goals for this work are to discover the best practices for achieving (i) maximum operating frequency, and (ii) maximum compute density (the amount of possible work per unit area) for tiled soft overlays. Notably we explore tiled designs that operate at much higher than usual clock frequencies (400 to 500 MHz), so we must carefully control their critical paths.

4.1.1 Terminology

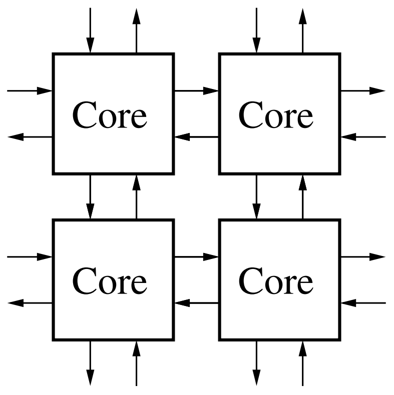
Multi-Local Logic Tiled designs necessarily contain logically equivalent, duplicated circuitry across each tiled element. We refer to this circuitry as "multi-local" since it operates locally and identically in multiple instances across a design. Normally, to reduce logic usage, a CAD tool performs redundancy elimination to optimize multi-localities down to a single instance ("deduplication") and then fans-out its output to all the original locations. While generally beneficial on a small scale, deduplication may introduce new critical paths on a larger scale or in higher-speed circuits, for only a modest reduction in logic usage. Current methods to control deduplication involve manually micro-managing the design description or the CAD tool settings on a per-logic-node basis [104].

Connected multi-local logic has a common point of origin, feeding multiple identical circuits producing identical outputs and/or entering the same state (e.g.: address and instruction decoders in SIMD lanes). We typically observe connected multi-local logic when tiling datapaths.

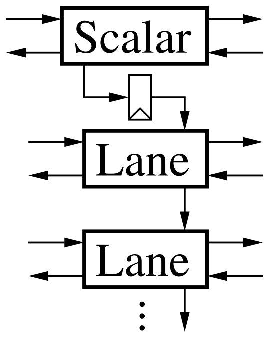
Unconnected multi-local logic shares no inputs or outputs, but produces identical outputs and remains in identical states (e.g.: identical counters initialized at reset). We typically observe unconnected multi-local logic when tiling entire soft-processors.

Partitioning Partitioning refers to the *logical* division of a design into one or more subsections, usually at module boundaries, which then synthesize as separate netlists. The total design remains the same except for certain optimizations such as register retiming or (de)duplication, and Boolean simplifications, which do not cross partition boundaries. In this study we find that partitioning provides a simple and effective method to provide high-level information to the CAD tool to preserve multi-localities during synthesis and thus avoid deleterious deduplication. *Merely partitioning the major datapaths of a tiled system suffices to prevent harmful global deduplication, while preserving beneficial local optimizations, for little cost in area.* Furthermore, this use of partitioning causes no significant changes in total CAD time, granting improved performance "for free" without increasing the design cycle time.

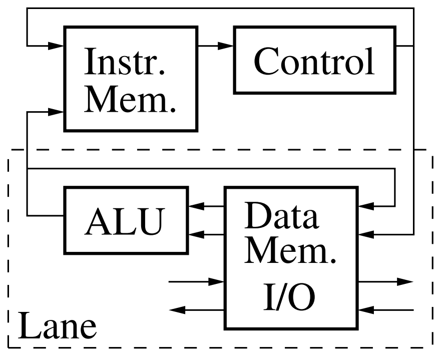
Compute Density Rather than focus entirely on maximum operating frequency, we also consider the overall efficiency of the resulting designs in terms of computation per



(a) Mesh



(b) Core



(c) Scalar

Figure 4.1: Block diagrams of Mesh, (SIMD) Core, and Scalar modules.

unit FPGA area per cycle—in other words, the *compute density*. A denser implementation is more desirable to replicate and tile into multicores. We demonstrate that partitioning improves density by increasing speed more than area.

Floorplanning Floorplanning describes the process of *spatially* pre-allocating areas on the FPGA device to sub-modules of a larger project, allowing them to evolve independently as sub-projects without encroaching on the placement (and thus, the timing) of other sub-modules. We find that floorplanning has no clear or predictable benefit to compute density, and always lowers compute density relative to the same design without a floorplan. *However, these results also show that a designer can tile an overlay without concern for the placement, proximity, or shape of each tile, letting the CAD tool find a good solution itself.*

Designers do currently use floorplanning and partitioning as project management techniques to enable incremental re-compilation in larger, multi-part projects [4, 136], but solely to preserve past CAD results, without any focus on improving performance directly. Similarly, designers leave smaller projects unpartitioned and free-form to give the CAD tool the freedom to perform as many global optimizations as possible, knowing that those optimizations normally improve performance and area.

4.1.2 Design Space

In this chapter, we focus on tiled designs composed of SIMD lanes of soft-processors, and multicore meshes built up from these. We construct these designs as extensions to the openly available Octavo[1] soft-processor [77], described in Chapter 3, whose operating frequency can reach up to the BRAM limit of 550MHz on Stratix IV FPGAs, and thus presents a good candidate for exploring the effects of tiled scaling and partitioning.

Figure 4.1 outlines the modules and tilings used throughout: Meshes (Figure 4.1(a)) directly connect the I/O of Cores in a bidirectional North/South/East/West manner. Each Core (Figure 4.1(b)) contains a Scalar processor with optional SIMD Lanes, all pipelined one instruction behind the Scalar processor for best performance[2]. Each Scalar (Figure 4.1(c)) processor contains a simple Control path and Data path with I/O memory-mapped into the Data Memory. Each SIMD Lane contains a single Data path.

We explore the speed, area, and density of SIMD processors with 0 to 32 lanes, with and without partitioning, and find that partitioning preserves multi-locality, increasing speed and density significantly, and increasing area moderately. We try to extend these SIMD results by adding instruction pipeline stages *between* each SIMD lane to force a sequential dependency between each lane to prevent deduplication. Pipelining does provide some benefits, but can only crudely approximate the effects of partitioning and staggers the execution across SIMD lanes. Finally, we create rectangular meshes (with North, South, East, and West point-to-point bidirectional links) of soft-processors, with and without SIMD lanes. We find that constructing high-speed meshes exposes different *unconnected* multi-localities, preserved with different partitioning, despite having the same underlying hardware as in the SIMD study. Partitioning meshes also increases their speed and density, with little area increase.

[1] Available on GitHub at `https://github.com/laforest/Octavo`

[2] We remove that pipeline register prior to benchmarking in Chapter 7, paying a 2–3% drop in average F_{max} in exchange for simpler programming by keeping all datapaths in lockstep.

4.1.3 Contributions

Our main contribution is demonstrating that partitioning can preserve the multi-locality of tiled designs, without requiring detailed per-node micro-management, and without interfering with other beneficial optimizations or the place-and-route process. Partitioning also gives better results than having the CAD tool indiscriminately either globally remove duplicate logic or not remove any. In addition, we demonstrate that: floorplanning provides no predictable benefits to compute density, and generally lowers it; partitioning the major datapaths of SIMD and Mesh tiled soft-processors improves speed and compute density, with the improvement increasing with the number of tiles and *without increasing the total CAD time*; and that we can force the CAD tool to preserve multi-locality by pipelining the paths between *connected* multiply-local logic instances, albeit also at the cost of staggered execution, and with no improvement to *unconnected* multi-localities.

4.2 Experimental Framework

Our experiments target an Altera Stratix IV FPGA of the highest speed grade. We test our circuits inside a synthesis test harness to ensure an accurate timing analysis. We average the Quartus synthesis results over 10 random initial seeds, and tune Quartus to produce the highest-performing circuits possible. Appendices A and B describe the experimental framework and Quartus settings in detail.

To explore the effects of duplicate register removal, we sometimes disable this specific optimization, denoted throughout as “nrdr” (No Removal of Duplicate Registers). We calculate compute density as total Peak MIPS (over all datapaths) per 100 eALMs (equivalent ALMs, see Appendix A). By itself, a Stratix IV ALM roughly contains two 6-LUTs, two flip-flops, and two full-adders with carry-chain logic.

Our experiments and results only apply against Altera’s Quartus CAD tool suite. However, based on past experience when porting designs from Altera to Xilinx platforms [75],

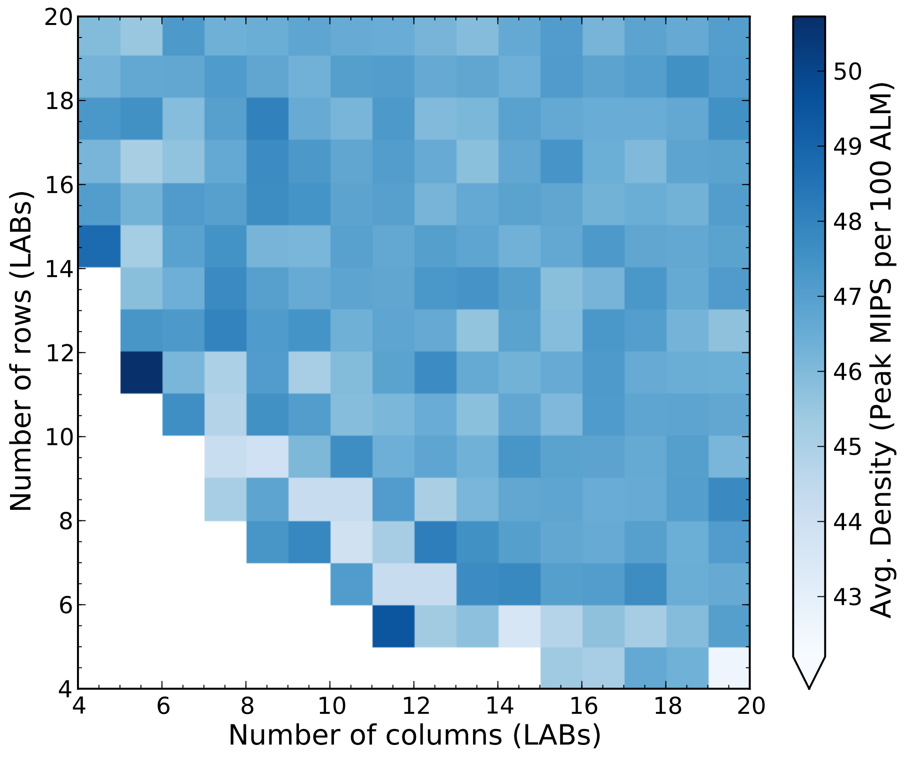


Figure 4.2: Average density heat map of a floorplanned Scalar Octavo Core

we believe that our partitioning techniques would produce similar results on Xilinx's ISE and Vivado CAD tools, once we accounted for their particular logic optimization behaviours.

4.3 Floorplanning a Scalar Core

We initially explored simply floorplanning multiple tiles into separate and adjacent floorplans (of sufficient area) to preserve their multi-localities, contain their critical paths, and thus improve performance. However, this approach gives the CAD tool a harder place-and-route problem to solve for an otherwise identical design, always produces worse results than non-floorplanned versions, and thus we abandoned this approach.

Knowing this limitation, our new goal was to find a floorplan layout that would act as a hint to the CAD tool to enable the logic of a single tile to fit better into the structure of the FPGA hardware and improve performance, prior to any tiling. The

Stratix FPGA architecture [82] places each logic resource type (ALMs, BRAMs, DSPs, etc...) into columns which span the height of the device and contain only one resource type each. Each cell in these columns has a relatively tall, narrow rectangular shape, and thus has more horizontal routing passing over it than vertical. We hypothesized that some floorplan shapes might take advantage of the cell aspect ratio and routing directional bias to improve logic usage and performance. We could then preserve the post-placement layout inside the floorplan, and simply “rubberstamp” the optimized tile across the FPGA.

Figure 4.2 shows a heatmap of the density of a Scalar Octavo Core when floorplanned into rectangular areas ranging from 4x4 to 20x20 LABs (Logic Array Blocks), with DSP and BRAM block excluded: The CAD tool may use DSP and BRAMs from outside the floorplan if necessary, and may float the floorplan anywhere on the FPGA to get good access to resources. A LAB contains a column of 10 ALMs with local interconnect. A Scalar Core requires over 64 LABs of logic area within the floorplan to successfully fit. The coordinates of a point on the heatmap denote a floorplan with a congruent rectangular shape, with the origin as the opposite corner. White squares indicate where the design did not fit in the floorplan.

As expected, forcing a tight fit (e.g.: less than 10x10) generally reduces F_{max}, and thus density, due to difficult routing. Any looser fit results in a middling density since the floorplan poses *fewer* constraints on placement and routing. Thus, even having a mostly full floorplan (or entire FPGA device) will not negatively affect density, regardless of geometry, but will also not improve it. A “sweet spot” at the 6x12 and 12x6 points, where the density jumps up by about 10% from about 47 to almost 52 Peak MIPS per 100 eALMs, suggests that an optimum floorplan shape does exist: the 8x9 and 9x8 points enclose the same area, but have lower density.

However, if we compare the floorplanned density results with the density of the same unfloorplanned Scalar Core in the next section, we see that even an ample floorplan has a noticeable negative impact on density: with no floorplan constraint, the Scalar Core reaches a density of 54 Peak MIPS per 100 eALMs: 4% higher than the 6x12 and 12x6 sweet spots, and about 14% higher than most other floorplan geometries.[3]

4.3.1 Summary

Overall, floorplanning generally reduces density, regardless of shape and size, and provides no predictable benefit. Thus, barring project management concerns, a designer should tile an overlay without floorplanning, letting the CAD tool find a good placement solution itself.

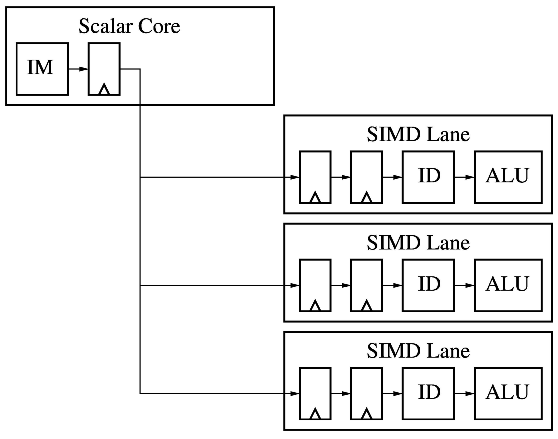
4.4 Partitioning Scheme Definitions

Throughout the remainder of this chapter, we refer to various partitioning schemes with shorthand labels. In increasingly fine-grained order: "Flat", which places the entire design inside one partition; "nrdr", identical to "Flat" except with **n**o **r**emoval of **d**uplicate **r**egisters; "Per-Core", which places each soft-processor into separate partitions; and "Per-Lane", which places each scalar processor and each of its SIMD lanes into their own separate partitions. We implement these schemes by declaring one or more modules as separate design partitions in the CAD tool. We do not consider schemes combining "nrdr" with "Per-Lane" or "Per-Core" as they always yielded worse results since not only would we prevent inter-module optimizations, we would also prevent any internal and usually beneficial optimizations.

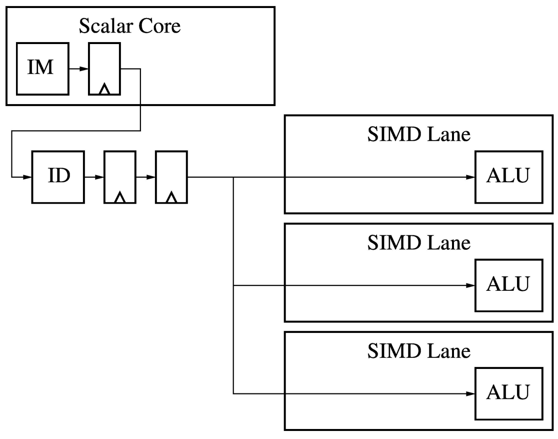
4.5 Partitioning a SIMD Core

Many overlays increase parallelism by tiling datapaths in a SIMD manner. We extended the Octavo soft-processor [77] to support SIMD processing and observed the same logic in

[3] After publication, private conversations with people inside Altera revealed that, because we let Quartus float the location of the floorplan as required, some internal placement optimizations got disabled.



(a) Before Deduplication



(b) After Deduplication

Figure 4.3: Deduplication of Instruction Decoding Pipeline in SIMD Lanes

each SIMD lane consistently appearing in the worst critical paths, with their propagation delay increasing with the number of SIMD lanes. This multiply-local logic included the instruction distribution pipelines within each SIMD lane, as well as any common instruction decoding logic, such as the I/O address decoders. The CAD tool deduplicated all instances across all SIMD lanes, worsening the fanout distance of the remaining instance as the number of lanes increased. Using partitioning to preserve the multi-local logic avoids these artificial critical paths.

Figure 4.3(a) outlines how our Scalar Core supports SIMD operations by duplicating its datapath once for each SIMD Lane. We feed instructions from the Instruction Memory (IM) to all Lanes in lock-step, but lagging one instruction behind the Scalar Core (Figure 4.1(b)). Using fewer pipeline stages lowers F_{max} by 2–3%, while more pipeline stages do not significantly improve it. Each SIMD Lane contains its own copy of the instruction pipeline registers, feeding an Instruction Decoder (ID), to scale them with the number of SIMD Lanes. We must explicitly duplicate the registers in the Verilog source itself since the CAD tool will not automatically replicate registers to control longer paths caused by increased fanout [104][4].

[4]Specifying a maximum fanout for the registers does not reliably prevent deduplication, and generally fails silently!

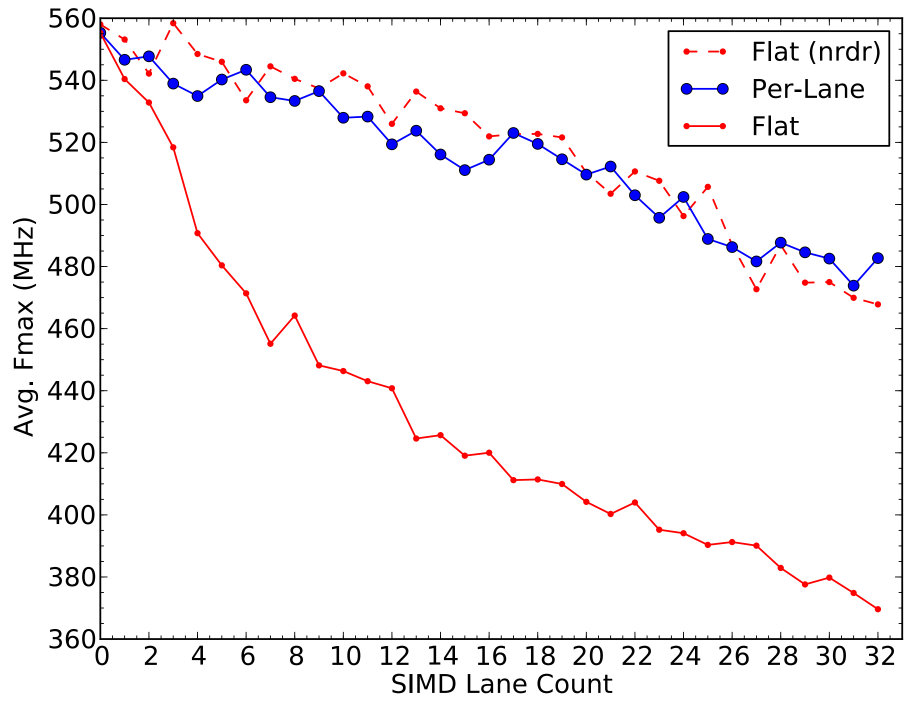


Figure 4.4: Average F_{max} of partitioned SIMD Cores

Unfortunately, without partitioning, the CAD tool deduplicates the multi-local SIMD instruction pipelines, resulting in the implementation shown in Figure 4.3(b). All SIMD pipeline registers and ID instances reduce to a single instance, which then feed the SIMD ALUs. Even if the CAD tool retimes the singular ID instance along the pipeline, the last pipeline register must still fanout to many physically distant ALUs. This fanout introduces artificial critical paths which worsen as the number of SIMD Lanes increases. The CAD tool does not deduplicate the SIMD ALUs since independent, unpredictable data memories also feed them, breaking multi-locality.

Figure 4.4 shows the average maximum operating frequency (F_{max}) of a soft-processor Core with 0 to 32 SIMD lanes when partitioned as a whole unit ("Flat"), or same with no removal of duplicate registers ("nrdr"), and with the Scalar processor and each SIMD Lane placed in their own partitions ("Per-Lane"). Under "Flat", F_{max} decreases from 555 to 372 MHz due to excessive deduplication. *Both the "nrdr" and "Per-Lane" partitioning schemes preserve the multiply-local logic in each Lane, thus limiting fanout distance and*

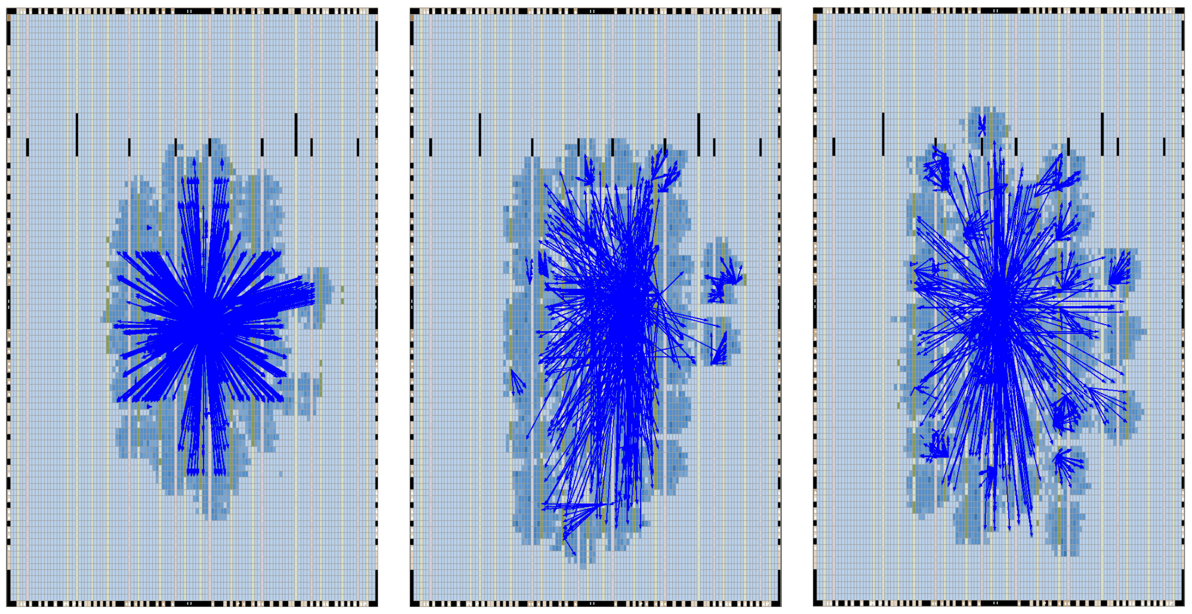

(a) Flat (373 MHz) (b) nrdr (456 MHz) (c) Per-Lane (489 MHz)

Figure 4.5: Fanout of the source nodes of the top 100 critical paths for a “Flat”, “nrdr”, and “Per-Lane” partitioned 32-lane SIMD Core.

preserving F_{max} to up to 482 MHz, a 30% gain over “Flat”.

4.5.1 Preserving Multi-Locality

Figure 4.5 illustrates the preservation of multi-locality when we partition SIMD Lanes, or simply disable duplicate register removal, by spatially plotting on the FPGA device the fanout of each source node in the top 100 critical paths of a 32-lane SIMD Core. The “Flat” critical paths (4.5(a)) reach 373 MHz and originate from 21 centrally placed nodes resulting from a deduplicated instruction pipeline fanning out over the entire design. The “nrdr” critical paths (4.5(b)) originate from 43 instruction pipeline nodes, distributed (and duplicated) more evenly over the design, and reach 456 MHz. Finally, the “Per-Lane” critical paths (4.5(c)) originate from 64 nodes (mostly I/O port address decoders fed by instruction operands) further spread out over the design, reaching 489 MHz because *we both preserved multi-locality across partitions and allowed optimizations (including deduplication) within each partition.*

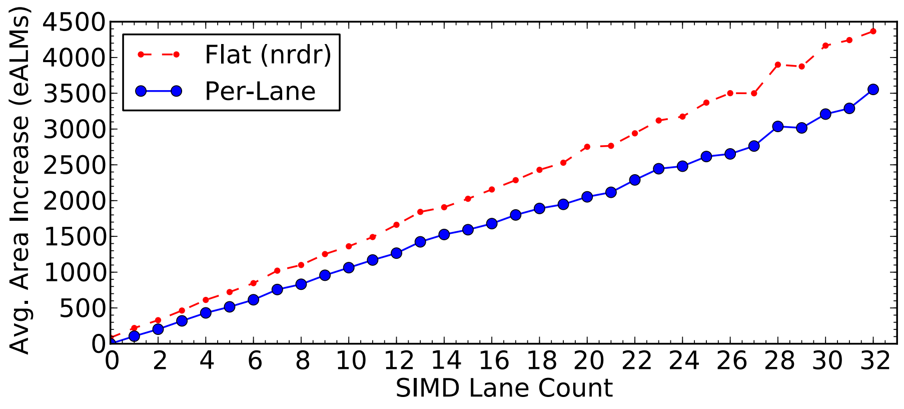


Figure 4.6: Average area increase of partitioned SIMD Cores, relative to "Flat"

4.5.2 Area Impact

Figure 4.6 shows the increase of the average area (in eALMs) of Cores with 0 to 32 SIMD Lanes, under the "nrdr" and "Per-Lane" partitioning schemes, relative to the "Flat" area. While "nrdr" and "Per-Lane" use more area than "Flat" since they prevents logic deduplication, *the "nrdr" scheme does so indiscriminately, preventing optimizations which would normally occur inside "Per-Lane" partitions, resulting in a consistently larger total area.* For reference, the "Flat" area ranges linearly from 1,027 to 25,779 eALMs, over 0 to 32 SIMD Lanes.

4.5.3 Compute Density

Figure 4.7 combines the results of Figures 4.4 and 4.6 into a chart of the compute density, measured in Peak MIPS per 100 eALMs, of the "Per-Lane", "nrdr", and "Flat" partitioning schemes for Cores with 0 to 32 SIMD Lanes. Partitioning "Per-Lane" improves density once a design grows to the point of introducing new critical paths if multiply-local logic gets globally optimized into a single instance. Thus, the "Flat" scheme provides the best density with 4 SIMD Lanes or fewer (almost 63 at 3 Lanes, a roughly 5% increase over "Per-Lane" and "nrdr"), *while the multi-locality preservation of "Per-Lane" improves density for larger number of SIMD Lanes (approx. 54.5 at 32 Lanes, a roughly*

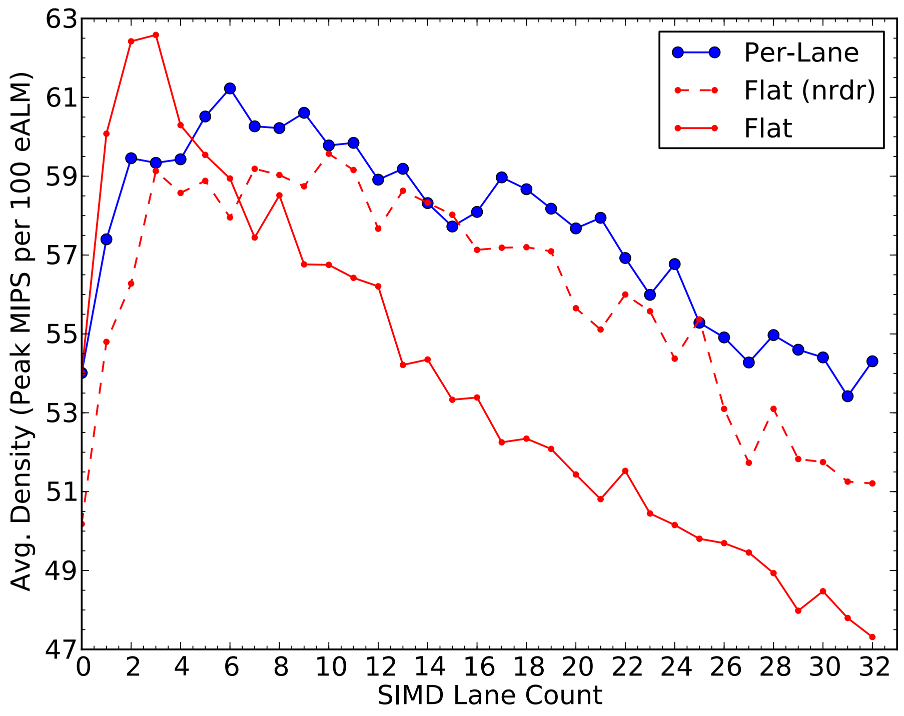


Figure 4.7: Average compute density for partitioned SIMD Cores

14% increase over "Flat"). The "nrdr" scheme's consistently larger area reduces its density to below that of "Per-Lane", despite having similar F_{max}.

4.5.4 Comparing Partitioning Schemes

Figure 4.8 compares the results from Figures 4.4, 4.6, and 4.7 as the average percent differences in F_{max}, area (eALMs), and compute density (Peak MIPS per 100 eALMs) of Cores with 0 to 32 SIMD Lanes, under "Per-Lane" and "nrdr" partitioning schemes, relative to "Flat" partitioning. The solid lines compare "Per-Lane" over "Flat", while the dashed lines compare "nrdr" over "Flat". A greater positive difference denotes a greater increase relative to "Flat". "Per-Lane" and "nrdr" always increase F_{max}, by as much as 31%, with a corresponding 15% increase in compute density. *However, the consistently 5% larger area of "nrdr" lowers its density to below that of "Per-Lane", thus we do not consider "nrdr" in later experiments.* Finally, for 4 SIMD Lanes or fewer, "Flat" has 5 to 10% better density due to its greater logic optimizations and smaller fanout distances.

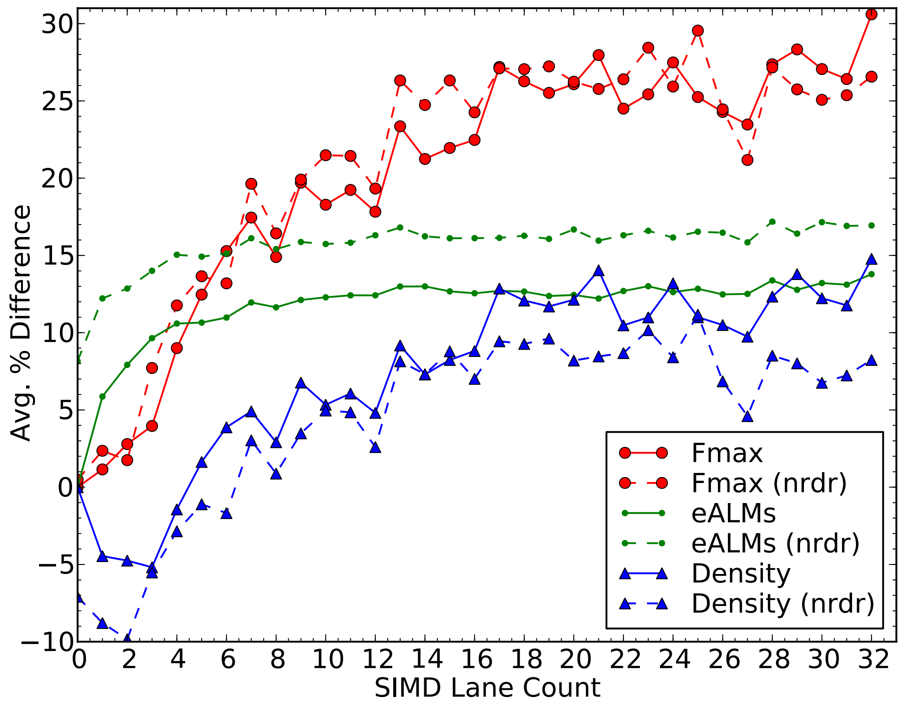


Figure 4.8: Average percent differences in F_{max}, area, and compute density of partitioned SIMD Cores, relative to "Flat"

4.5.5 Summary

For SIMD Cores, partitioning "Per-lane" results in the greatest increase in F_{max} and density compared to keeping the entire Core within a single partition ("Flat"), even when also disabling removal of duplicate registers ("nrdr"). "Per-Lane" partitioning preserves the multi-locality within each SIMD Lane, avoiding the excessive fanout of instruction pipelines and I/O port address decoders otherwise optimized to a single instance by the CAD tool, while allowing beneficial local optimizations within each partition. The improvement from partitioning increases with the number of SIMD Lanes, reflecting the increasing multi-locality.

4.6 Layering a SIMD Core

From our exploration of SIMD Lane partitioning in Section 4.5, we know that a SIMD Core implementation with fewer SIMD Lanes reaches a higher F_{max} (Figure 4.4) due to

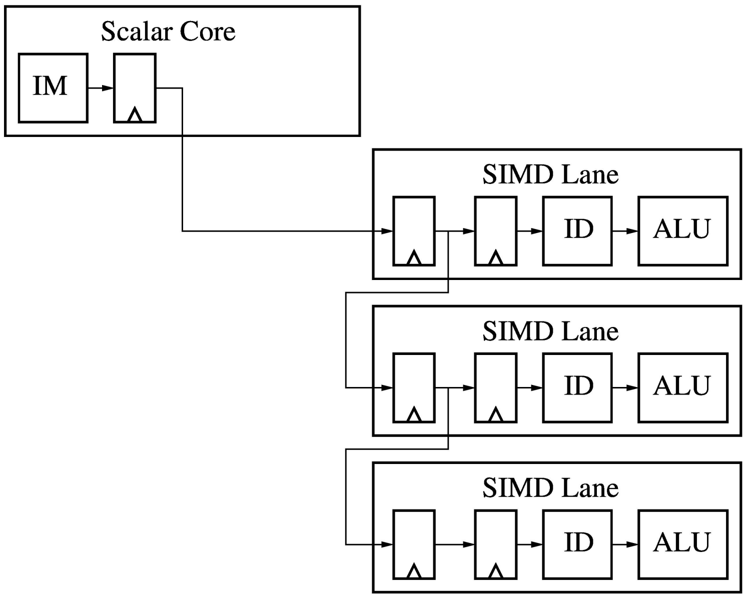


Figure 4.9: Layered SIMD Lanes: 3 Lanes with 1 Lane Per Layer

lower instruction fanout.

Figure 4.9 illustrates how we could replicate the conditions of fewer SIMD Lanes in a design with more SIMD Lanes. We divide N SIMD Lanes into M "layers" each containing N/M SIMD Lanes, with one of these Lanes also acting as an instruction distribution pipeline stage to the next layer. Because of this layering of SIMD lanes, each successive layer lags the previous one by one instruction, which staggers their execution and would complicate programming. However, as the number of layers increases, we effectively re-introduce the instruction distribution pipeline registers that the "Flat" partitioning optimizes away. These new registers sequentially depend on each other, since they hold different instructions each, thus the CAD tool cannot deduplicate them and increase their fanout. We can compare the impact of Layering with that of "Per-Lane" partitioning to see if Layering can replace it.

Figure 4.10 shows the impact on the average F_{max} (MHz) when layering combinations of soft-processor configurations with $N = [1, 2, 4, 8, 16, 32]$ SIMD lanes placed in each case of $M = [1, 2, 4, 8, 16, 32]$ layers (where possible), up to N layers containing 1 SIMD Lane each. We show both "Flat" and "Per-Lane" partitioning of the layered SIMD Cores

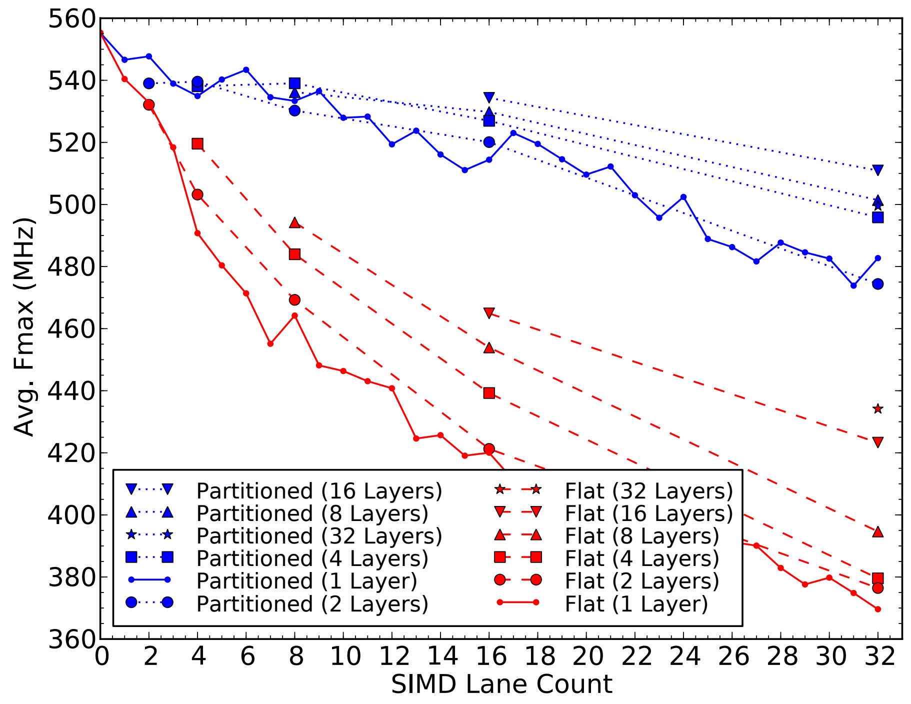


Figure 4.10: Average F_{max} of "Per-Lane" and "Flat" layered SIMD Cores

where, in the "Per-Lane" case, each Layer gets placed in its own partition. The base case consists of N Lanes each in 1 Layer (or, M Layers each containing 1 Lane), represented by the solid lines, and identical to the results from Figure 4.4. *The greatest F_{max} increases happen when each layer contains a single SIMD Lane, regardless of total Lane count.* For 32 lanes, the F_{max} of "Per-Lane" increases by only 30 MHz (+6%) relative to to the base case, while the F_{max} of "Flat" increases by 65 MHz (+17%), which also increases more throughout all Layer/Lane combinations. Even with the speed gains of Layering, we see that "Per-Lane" partitioning of a single-layer SIMD Core (i.e.: solid line) always performs better than any layered "Flat" version, and without staggering execution across layers. Layering SIMD Lanes has little effect on area (< 2% increase at most).

4.6.1 Summary

Sequentially pipelining SIMD Lanes into layers preserves their multi-locality in a way the CAD tool cannot optimize away, coarsely reproducing the effects of "Per-Lane" partitioning of each SIMD Lane. If a designer cannot use partitioning, then pipelining provides the next best alternative, albeit with staggered execution between Layers. However, partitioning by itself avoids the programming complications of staggered execution and will always reach a higher F_{max}.

4.7 Partitioning Meshes

To contrast with SIMD parallelism, we tile a number of Scalar and SIMD Octavo Cores into rectangular Meshes (Figure 4.1(a)), with simple horizontal and vertical point-to-point bidirectional links connecting read and write I/O ports, and compare them to their building block Core in isolation. Tables 4.1 and 4.2 compare the F_{max} (MHz), area (eALMs), and density (Peak MIPS per 100 eALMs) of various Scalar and SIMD Cores tiled into rectangular Meshes of various shapes and sizes, under three increasingly fine-grained partitioning schemes: “Flat”, which keeps the entire mesh in a single partition; “Per-Core”, which places each Core (including its SIMD lanes, if any) into a partition; and “Per-Lane”, which places each Scalar Core, and each SIMD Lane, into their own separate partitions. *For single-Core Meshes (1x1), “Per-Core” and “Flat” are equivalent, so we show “Flat” only.*

4.7.1 Meshes of SIMD Cores

Table 4.1 compares Meshes with a total of 32 datapaths (number of datapaths in all SIMD Lanes and Scalar Cores), along with their respective Core used as a building block, ranging from a 4x8 32-node Mesh of Scalar Cores (1 datapath each), down to a 1x1 Mesh composed of a single 31-Lane SIMD Core (31 datapaths in SIMD Lanes, plus 1 datapath in Scalar Core).

Under “Flat” partitioning, the dominating critical paths originate in the deduplication of a 3-bit free-running thread counter, found in the Control logic of each Core (see Figure 4.1), which does not depend on the contents of any memory or the output of any

Table 4.1: Speed/Area/Density of Partitioned Meshes with 32 Datapaths

Mesh (WxH)	SIMD Lanes	Datapaths (Total)	Scheme	Fmax (MHz)	Area (eALMs)	Density
Scalar Core						
1x1	0	1	Flat	555	1,027	54.0
4x8	0	32	Flat	362	36,379	31.8
4x8	0	32	Per-Core	481	36,651	42.0
1-Lane SIMD Core						
1x1	1	2	Flat	540	1,799	60.0
1x1	1	2	Per-Lane	547	1,904	57.5
4x4	1	32	Flat	380	32,226	37.7
4x4	1	32	Per-Core	438	32,718	42.8
4x4	1	32	Per-Lane	468	34,348	43.6
3-Lane SIMD Core						
1x1	3	4	Flat	518	3,313	62.5
1x1	3	4	Per-Lane	539	3,632	59.4
2x4	3	32	Flat	385	29,865	41.3
2x4	3	32	Per-Core	412	30,313	43.5
2x4	3	32	Per-Lane	461	32,321	45.6
31-Lane SIMD Core						
1x1	31	32	Flat	374	25,096	47.7
1x1	31	32	Per-Lane	473	28,385	53.3

logic. Thus, the CAD tool considers them logically equivalent and optimizes them down to a single instance. The fanout from this single counter instance causes a drop in F_{max} proportional to the number of Mesh nodes. *We can demonstrate this drop by simply separating the Cores into their own partitions ("Per-Core") to preserve the multi-locality of the thread counter, with F_{max} improving in proportion to the number of Mesh nodes:* a 4x8 Mesh of Scalar Cores improves its F_{max} by 33%, while the equivalent but half-size 4x4 Mesh of 1-Lane SIMD Cores see a proportional improvement of 15%, and the quarter-size equivalent 2x4 Mesh of 3-Lane SIMD Cores see a 7% improvement. Note that all these Meshes have similar total areas, so the gain from "Per-Core" partitioning only depends on the number of Cores, not their individual area.

Even after "Per-Core" partitioning, Mesh nodes with SIMD Lanes still have to preserve the multi-locality of their instruction pipeline and I/O port address decoders. Par-

Table 4.2: Speed/Area/Density of Partitioned Meshes with 102 Datapaths

Mesh (WxH)	SIMD Lanes	Datapaths (Total)	Scheme	Fmax (MHz)	Area (eALMs)	Density
1x1	0	1	Flat	555	1,027	54.0
17x6	0	102	Flat	287	115,655	25.3
17x6	0	102	Per-Core	434	115,245	38.4
6x17	0	102	Flat	284	114,803	25.2
6x17	0	102	Per-Core	437	115,775	38.5

titioning "Per-Lane" suffices, since this scheme places each Scalar component of each SIMD Core into its own partition, and thus also preserves the multi-locality of the thread counter. In each case shown in Table 4.1, when going from "Per-Core" to "Per-Lane" partitioning, the F_{max} increases with the number of datapaths in each Core: the 4x4 Mesh of 1-Lane Cores improves by 7% (over "Per-Core"), the 2x4 Mesh of 3-Lane Cores improves by 12%, and the 1x1 Mesh of a single 31-Lane SIMD Core improves by 26%. *Once we preserve all the multi-localities, the final F_{max} of all the different 32-datapath Meshes lie within a 4% range.*

4.7.2 Meshes of Scalar Cores

Table 4.2 shows two shapes (17x6 and 6x17) of a Mesh of 102 Scalar Cores, the largest number possible before running out of M9K BRAMs[5]. Spanning the entire FPGA device (see Figure 4.11), this 102-node Mesh uses 67% of all logic, broken down by Quartus as: 76% of all memory bits, 74% of all flip-flops, and 24% of all ALUTs. The aspect ratio of the 102-core Mesh has negligible effect on speed and area, and partitioning "Per-Core" improves F_{max} by 51 to 54%, again showing the benefits of preserving the multi-locality of the thread counter and that our gains do not depend on being able to spatially separate each datapath, as Figure 4.5 might imply.

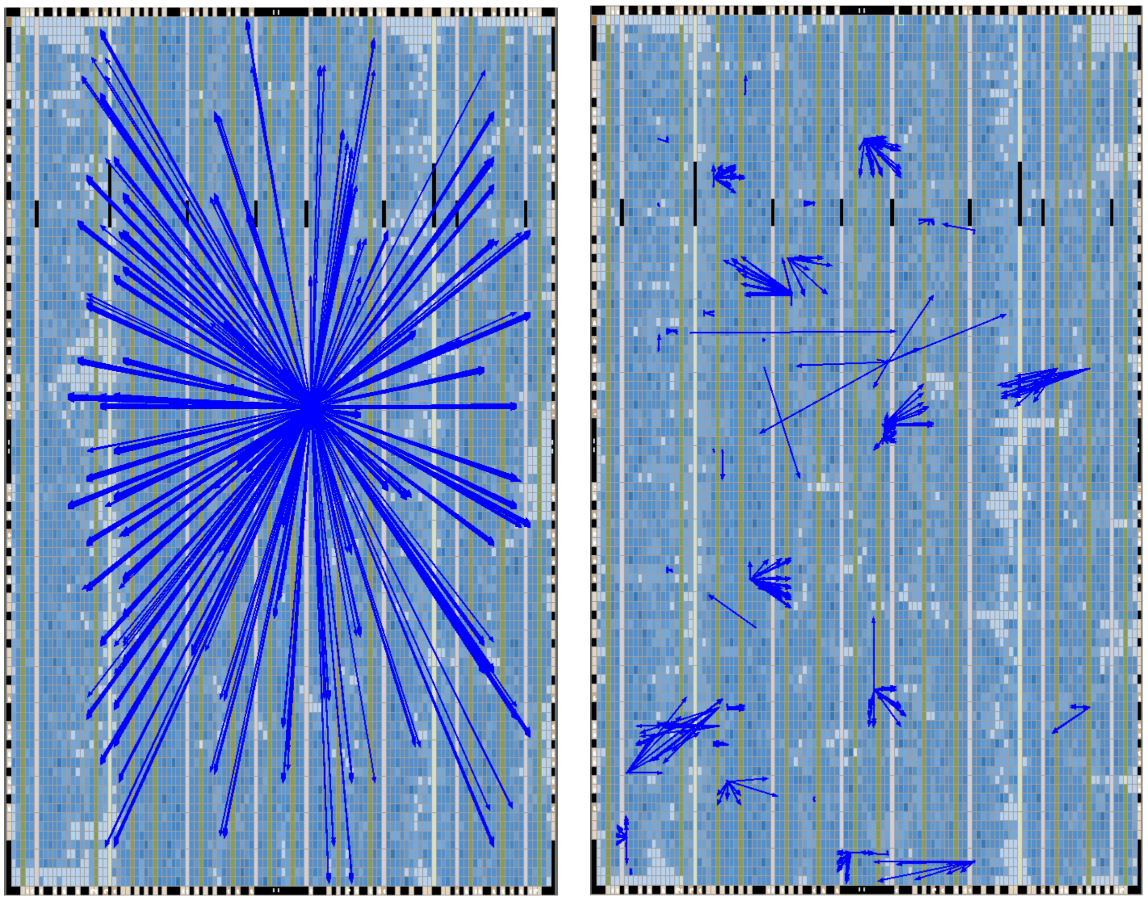

(a) Flat (331 MHz) (b) Per-Lane (489 MHz)

Figure 4.11: Fanout of the source nodes of the top 100 critical paths for "Flat" and "Per-Lane" partitions of a 102-core 17x6 tiled mesh of Scalar Cores.

4.7.3 Preserving Multi-Locality

Figure 4.11 illustrates the fanout of the source nodes of the top 100 critical paths for "Flat" and "Per-Lane" partitions of a 102-core 17x6 tiled mesh of Scalar Cores. The "Flat" critical paths reach 331 MHz, and despite having no shared instruction pipelines or address decoders, originate from only **2** centrally placed nodes resulting from the deduplication of free-running thread counters common to the Control logic in all Cores (see Figure 4.1), now fanning out over the entire design. In contrast, the "Per-Lane" critical paths originate from 61 unrelated and dispersed nodes, and reach an F_{max} of 489 MHz.

[5]Each Scalar Core uses 12 M9K BRAMs, leaving 11 unused out of 1235.

4.7.4 Area, Speed, Density, and CAD Time

Since partitioning prevents deduplication of multi-local logic, it increases the area of a tiled design. *However, the area increases only proportionally to the size of the preserved multi-local logic, not from partitioning itself.* For example, the area of the 102-core 6x17 Mesh increases by less than 1% under "Per-Lane" partitioning, reflecting the tiny area of the preserved multi-local 3-bit counters. At the other extreme, the area of the 1x1 Mesh with 32 datapaths increases by 13% due to the larger area of multi-local SIMD instruction pipelines and address decoders. In the middle, the area of the 4x8, 4x4, and 2x4 Meshes (also with 32 datapaths) increases by 8% or less since the multi-localities include both the smaller counters and the larger instruction pipelines and address decoders. *In all cases, no relation exists between the increase in F_{max} and the increase in area.*

The changes in F_{max} and area as we scale-up Meshes suggest that: (i) we may be getting near-optimal speed preservation from the CAD tool when partitioning; (ii) the CAD tool introduces some unknown area overhead, causing a drop in density, even though we added no extra hardware when tiling. Under "Per-Lane" partitioning, tiling a Scalar Core 32 times in the 4x8 Mesh decreases F_{max} only 13% (with similar decreases for the 4x4 and 2x4 Meshes). Increasing the tiling 3.18x further to 102 cores decreases F_{max} by only an additional 9%, down to 78% of the original speed of a single Scalar Core. *A 22% drop in speed over 102x scaling supports our original intuition that multiple independent tiles should run at similar speeds to a single tile.*

Intuitively, we expect that tiling without adding any other hardware should scale the total area by the number of tiles. However, when tiling a Scalar Core 32 times (4x8 Mesh) the area increases 35.7x (Table 4.1), and 112.2x when tiling it 102 times (Table 4.2): a 10 to 11% overhead. This area increase occurs regardless of the partitioning scheme used, suggesting that it might not originate in new logic duplication or deduplication caused by the tiling, especially with only nearest-neighbour point-to-point connections. If the area increased as expected, and assuming the same final F_{max}, the density of the 102-core mesh would reach 42.3, placing it within 8% of the densest 2x4 Mesh (or within 11% of a similarly speculative 4x8 Mesh). *Thus, we speculate that some area overhead introduced by the CAD tool, rather than the decrease in F_{max}, primarily causes the drop in density as Meshes scale up.*

Partitioning has no significant effect on CAD time. For example, for the 17x6 102-core mesh, the total CAD time goes from 2h:28m when "Flat", to 2h:17m (-7%) when partitioned "Per-Lane". Synthesis and Place-and-Route consume most of the total time. The synthesis time went from 0h:18m to 0h:12m (-33%), since the CAD tool can synthesize each partition in parallel, but still represents only 12 to 8% of the total time. The Place-and-Route time went from 2h:05m to 1h:55m (-8%), but still represents 84% of the total, with or without partitioning. Furthermore, once partitioned for performance, nothing prevents a designer from using partitioning (and optionally, floorplanning) to enable incremental re-compilation and reduce future CAD time by containing the effect of design changes [4, 29, 136].

4.7.5 Summary

Different kinds of tiling (SIMD Lanes vs. Mesh) expose different multi-localities in the same tile hardware, which we preserve with different partitioning schemes to recover lost performance. Partitioning itself has no area impact: any increase in area purely depends on the area of the preserved multi-local logic, with no correlation to the amount of lost/regained performance. The CAD tool preserves the operating frequency of partitioned tiles extremely well as their number increases, but introduces an unexplained proportional area overhead, independent of partitioning, which may explain the decreasing compute density with scaling. Lastly, partitioning has no significant effect on total CAD time, even without incremental recompilation.

4.8 Related Work

Past work on partitioning and FPGAs address a different, but related problem: how to partition the netlist of an ASIC project across multiple FPGAs for simulation [17]. Fang and Wu [44, 45] found that using the design hierarchy to guide partitioning would lead to higher logic block utilization and lower I/O pin utilization, which commonly formed the bottleneck when partitioning over multiple FPGAs, and resembles our avoidance of high-fanout paths. Some early work by Vahid, Frank, Le, and Hsu [120, 121] pointed to the advantages of functional partitioning, the kind we do in this chapter, over structural partitioning, where partitioning occurs after the synthesis of a final, flattened netlist. They observed a similar control of the critical paths (and area increase) by avoiding sharing logic between functions (i.e.: "multi-local" logic).

4.9 Programming Model: SIMD

The addition of SIMD Lanes naturally extends the pipelined multi-threading programming model we describe in Section 3.13.2: each thread executes a private copy of the same code with different address calculations to divide the data amongst the threads. Each SIMD Lane effectively executes a copy of these original Scalar Octavo threads, but does so on private data in the SIMD Lane's A and B memories.

Difficulties: Data-Dependent Control-Flow Unfortunately, these SIMD copies of the original Scalar Octavo threads must follow the branching of their controlling Scalar thread. Thus, if the thread code contains data-dependent branches, each SIMD copy may end up diverging, which we do not support. Instead, we have to execute *both* sides of a data-dependent branch identically in each SIMD thread copy, then select the desired result via a conditional move (`CMOV`) operation which does not itself require branching. We can currently simulate `CMOV` using Boolean operations and bit-masks, and we outline potential implementations in Section 8.6.1.

Data Pre-Copy Across SIMD Lanes Since the SIMD Lanes cannot communicate as-is[6], we must pre-copy some data, normally shared across threads in the same datapath, into each SIMD Lane at load time. For example, in Figure 7.6 (pg. 139), we show how to convert a FIR filter to multi-threaded (and thus also, SIMD) operation by adding data headers pre-copied from neighbouring data subsets.

4.10 Conclusions

Through our study of high-speed tiled FPGA overlays, we found the notion of "multi-local" (repeated and logically-equivalent across tiles) logic useful to describe the origin of the worst critical paths which emerge when tiling datapaths or even entire processors: By default, the CAD tool performs redundancy elimination ("deduplication") on all multi-local logic, reducing it to a single instance which then fans out to all tiles, introducing large and unnecessary critical paths for little area savings.

We found that partitioning the major datapaths of a high-speed tiled system into separate netlists suffices to prevent harmful global deduplication of multi-local logic, while preserving beneficial local optimizations. Partitioning provides:

- a simpler approach than the existing per-node micro-management of a design description or of the CAD tool settings;
- a lower area increase than disabling duplicate register removal in the CAD tool, with the area increase only depending on the area of the preserved multi-local logic;
- greater performance benefits than forcibly introducing sequential dependencies via pipelining;
- a performance increase that scales with the number of tiles;
- improved performance with no significant change in total CAD time.

[6]See Section 7.2.3 for an example of hardware-assisted SIMD cross-lane communications.

Although our results specifically apply to Altera's Quartus FPGA CAD tool suite we believe, based on past experience when porting designs from Altera to Xilinx platforms [75], that the same partitioning principles should hold for Xilinx's ISE and Vivado CAD tools, once we account for their particular logic optimization behaviours.

Chapter 5

Planning for Larger Systems

The first weakness of minicomputers was their limited addressing capability. The biggest (and most common) mistake that can be made in a computer design is that of not providing enough address bits for memory addressing and management. The PDP-11 followed this hallowed tradition of skimping on address bits, but it was saved by the principle that a good design can evolve through at least one major change.

C. Gordon Bell [18]

While Octavo's architecture achieves a high F_{max} (Chapter 3 and scales effectively to an entire FPGA (Chapter 4), the ALU writes its results to all memories, uselessly duplicating data in the A/B memories and eating up precious space in the I memory.

We can add any desired hardware extension via the I/O ports, but F_{max} suffers as their numbers increase. Also, the I/O ports are "bare", without any handshaking to signal the sender/receiver of consumed or available data, requiring inefficient software busy-wait interfaces to variable latency devices such as DRAM controllers.

Thus, to set the stage for later architectural improvements (Chapter 6) and performance benchmarking (Chapter 7), we must remove the duplication of ALU writes, provide a separate address space for additional hardware, and enable the I/O ports to autonomously synchronize data transfers without involving software.

Table 5.1: Octavo's Instruction Word Format with Extended Write Address Space

Size:	4 bits	12 bits	10 bits	10 bits
Field:	Opcode (OP)	Destination (D)	Source (A)	Source (B)

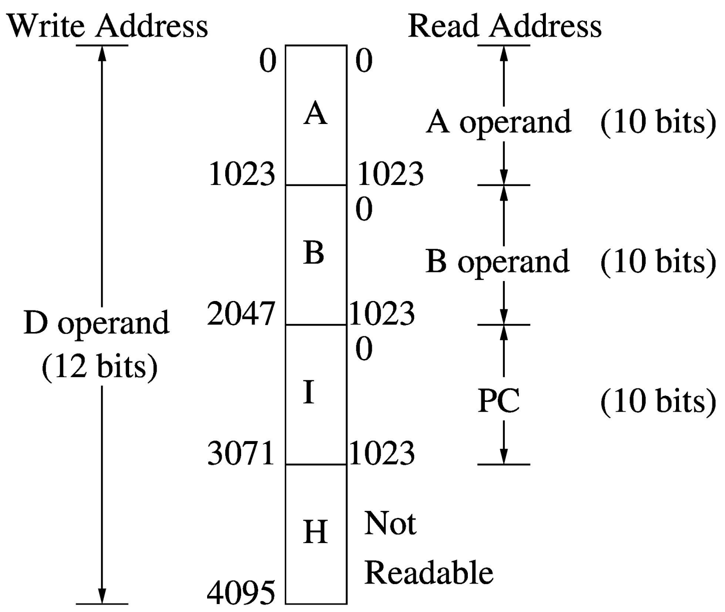


Figure 5.1: New Octavo memory map with 4 kilo-word memory write space.

5.1 Extending the Write Address Space

Octavo's original instruction format (Table 3.1, pg. 34) uses 34 bits out of a 36-bit word: 4 bits for the opcode, and 10 bits each for both source operands and for the destination operand, leaving 2 bits unused. We can append those 2 spare bits to the destination operand, expanding it to 12 bits (Table 5.1).

Figure 5.1 illustrates the new memory map, treating the resulting 4 kilo-word write address space as four 1 kilo-word pages: one for each of the A, B, and I memories, and a new "High" memory area within which we can map new hardware extensions without requiring I/O ports, but at the price of being "write-only".

The memory reads execute as before, with the A/B operands indexing into their corresponding A/B data memories, and the PC indexing into I memory (not readable by A/B). However, we now decode writes from the ALU to only write into one of these memories at a time, over a single contiguous write address space. This change avoids code

Listing 5.1: Memory Write Address Offset Example

```
// A[1] and B[1] both contain 1
// A[100], B[100], and I[100] contain 1
// Instruction format: OP  D, A, B

// A[100] now contains 2
ADD 100, 100, 1

// B[100] now contains 2
ADD 1124, 1, 100

// I[100] now contains 3 (A[1] + B[100])
ADD 2148, 1, 100
```

and data duplication across memories, doubling the amount of available A/B memory, and removes data from I memory and code from A/B memory.

5.1.1 Impact On Software

Because we re-map the write address space of the A/B/I memories to possibly non-zero origins, the read and write addresses of a given memory location no longer necessarily match. We must add the appropriate offset to a read address to get the write address of the same location before we can use it as a destination operand address. Respectively, to write to the same location in the A, B, and I memories, we add offsets of 0, 1024, and 2048 to the read address. Listing 5.1 illustrates the process in assembly.

We must currently add the offset at assemble/compile time, but we replace it with a hardware solution in Chapter 6 where we can pre-set the offset for each memory.

5.2 Adding I/O Handshaking

Currently, the Octavo I/O read and write ports assert a read enable (rden) or write enable (wren) signal when reading/writing, indicating received or newly written data. However, no mechanism exists to determine if the other end has sent or can receive new

Listing 5.2: Read Busy-Wait Loop Example

```
1 // WREN_INPUT: read port with wren bits from sending ports
2 // WREN_MASK:  bit mask for wren bits of interest
3 // STATUS:     status of the sending write ports
4
5 WAIT:
6 AND STATUS, WREN_INPUT, WREN_MASK // Mask out other ports
7 XOR STATUS, STATUS,     WREN_MASK // Toggles STATUS bits
8 JNZ WAIT,   STATUS,     0         // Wait while STATUS != 0
9 <read data here>
```

data, short of using a software loop to monitor another I/O port signalling readiness. Complicating this limitation, any Octavo instruction may access up to 3 I/O ports (two reads, one write). We must ensure that if *any* of the I/O ports are not ready, then *none* of the reads and/or writes proceed, else we may lose data. We must somehow suspend an instruction accessing un-ready I/O ports until all ports are ready.

Listing 5.2 outlines an example busy-wait loop which we would place before any instruction which accesses an I/O read port (or two). We use an additional read port (`WREN_INPUT`) to receive the active-high wren bits of the sending write ports (e.g.: from another Octavo core). Line 6 masks out the wren bits of the port(s) we want to receive data from. Any remaining high bits indicate a ready sending port. Line 7 uses the same mask to toggle the high bits back to low, and any low bits to high, hopefully generating an all-zero `STATUS` indicating that *all* ports of interest are ready. Finally, line 8 jumps on non-zero to restart the busy-wait process if *any* port(s) remain un-ready.

This example busy-wait loops selects from up to 36 ports in parallel, and could also process write ports by connecting the receiving rden signals to `WREN_INPUT`. However, each instruction spent by a thread on a busy-wait loop adds 8 clock cycles of latency to any transaction on the I/O port, greatly reducing the throughput of the fine-grained interactions we expect within an Octavo overlay. We must also precede any instruction accessing I/O with such a busy-wait loop, leaving less room in the limited instruction memory for code performing actual work. Thus, we need a more efficient hardware

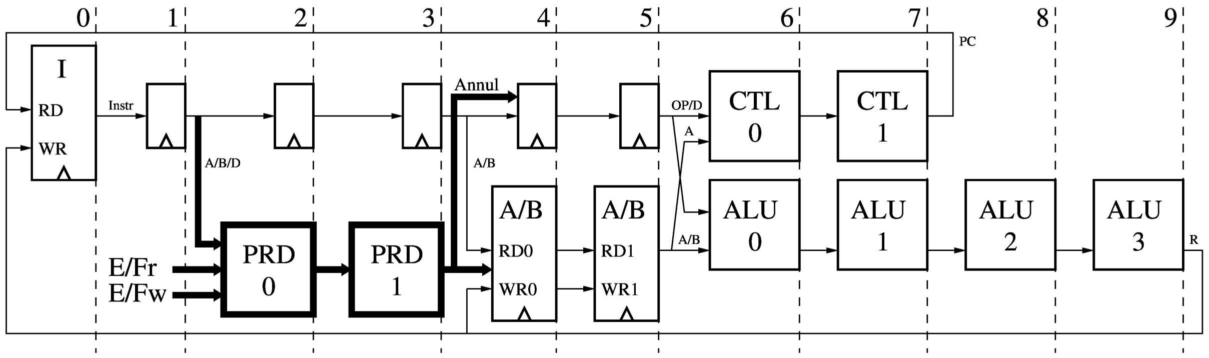


Figure 5.2: New Octavo block-level diagram, with I/O predication

mechanism to handle unpredictable I/O latencies.

5.2.1 Instruction I/O Predication

Ideally, a thread instruction which accesses an unready I/O port should have no effect and re-issue the next time around, eventually executing when all ports are ready. Thus, a thread will automatically hang, with no additional overhead, until the instruction can complete. All other threads remain unaffected. We can speak of this instruction as *predicated* on the status of the I/O ports it accesses.

Figure 5.2 illustrates the high-level location of the I/O predication mechanism, in pipeline stages 2 and 3, which inspects the A/B/D instruction operands and raises an "Annul" signal (stage 4, top-centre) if the instruction accesses any I/O ports not able to send/receive data, with the following effects:

- zeroes-out the instruction (converting it into a NOP) before it reaches the ALU and the Controller,
- inhibits the wren and rden signals of all accessed I/O ports, preventing loss of data,
- signals the Controller to re-issue the PC value of the instruction, so it can try to execute again.

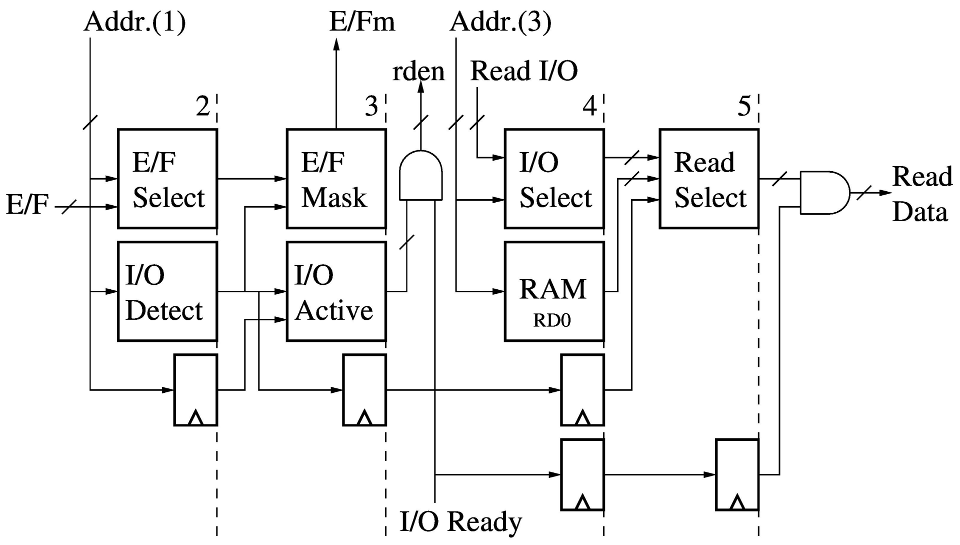


Figure 5.3: Block diagram of the I/O Read predication module.

The I/O Predication mechanism contains three major components: the I/O Read and I/O Write predication modules, and the All I/O Ready module, which coordinates their operation and ultimately generates the Annul signal. Enabling instruction re-issue also requires modifications to the Controller.

5.2.2 I/O Read Predication

Figure 5.3 shows the block diagram of the I/O Read predication module, with each pipeline stage labelled with its Octavo pipeline stage number. We require two I/O Read modules, one for each of the A and B memories, each addressed by their corresponding A/B instruction source operand.

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 memory (see Figure 3.3(c), pg. 36). 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 I/O ports are ready, will disable the rden signal if not asserted, preventing the completion of the read transaction.

Stages 4 and 5 proceed as originally designed, using the read address out of stage 3 to both select a Read I/O port 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.

Finally, if two threads share an I/O read port connected to a variable latency sender, the data will indeterminately divide between the two threads, possibly starving one thread. In the worst case, one thread will always empty the read port before the other, which will then perpetually hang waiting for read data. Thus, either the various latencies must be fixed, or each thread must have exclusive use of a port, or we must multiplex a read port's data and handshaking state across all accessing threads.

5.2.3 I/O Write Predication

Figure 5.4 shows the block diagram of the I/O Write predication module, with each pipeline stage labelled with its Octavo pipeline stage number. 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.

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 (see Figure 3.3(b), pg. 36). 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.

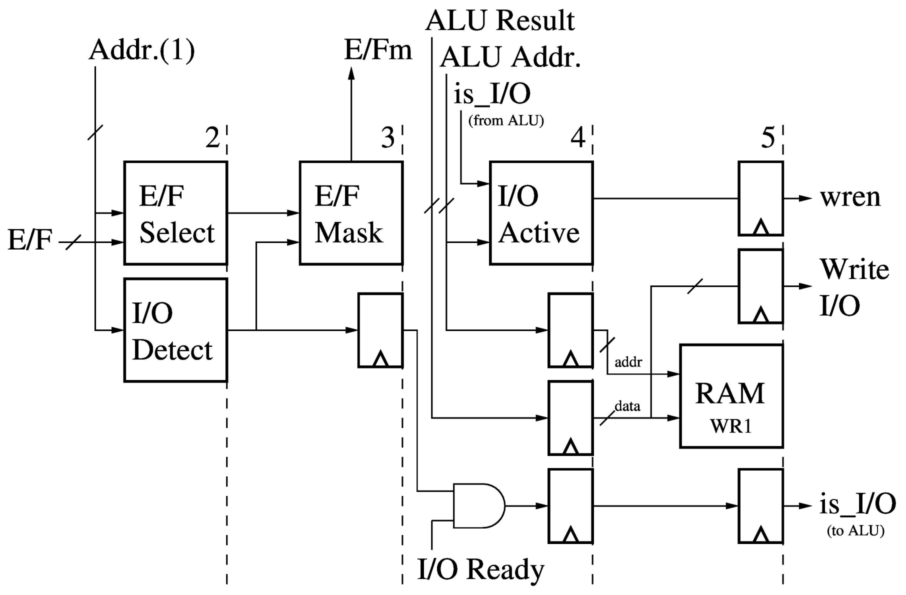


Figure 5.4: Block diagram of the I/O Write predication module.

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. In the worst case, we annul an instruction due to a full I/O write port just before the port empties, costing 8 clock cycles which would have been saved by checking later in the pipeline, at the time of the write.

However, if we predicated writes at the point where they occur in the pipeline, 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 8 threads. Also, the next thread instruction will have already issued by then, and so we must annul it until the previous I/O write completes, requiring potentially slow immediate 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 and the propagated is_I/O bit to enable the appropriate write

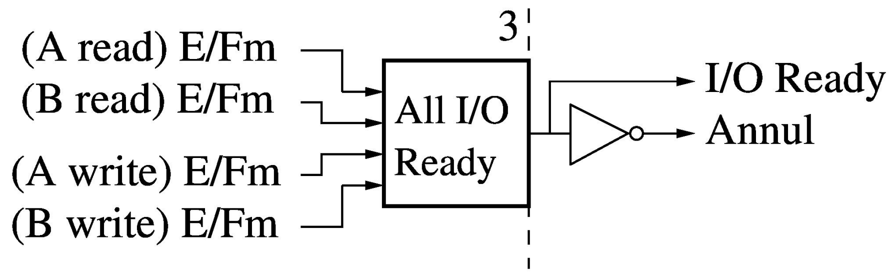


Figure 5.5: Block diagram of the All I/O Ready module.

enable (wren) line, while the ALU Result proceeds to both RAM and Write I/O port.

Finally, without special care, two threads should not share a common write port. Two instructions from different threads can check the I/O write port and find it empty. Later in the pipeline, the earlier instruction then fills the write port, and the later instruction then attempts to write to a still-full write port, losing the data. Thus, either the filling and emptying rates of the port must be fixed and matching, each thread must have exclusive use of a write port, or we must multiplex the data and handshaking state of the write port across all accessing threads.

5.2.4 All I/O Ready Coordination

Figure 5.5 shows the origin of the crucial I/O Ready signal. The All I/O Ready block receives the selected and masked Empty/Full bit (E/Fm) from each I/O Read and I/O Write 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 treat the Annul bit as the inverse of I/O Ready for convenience.

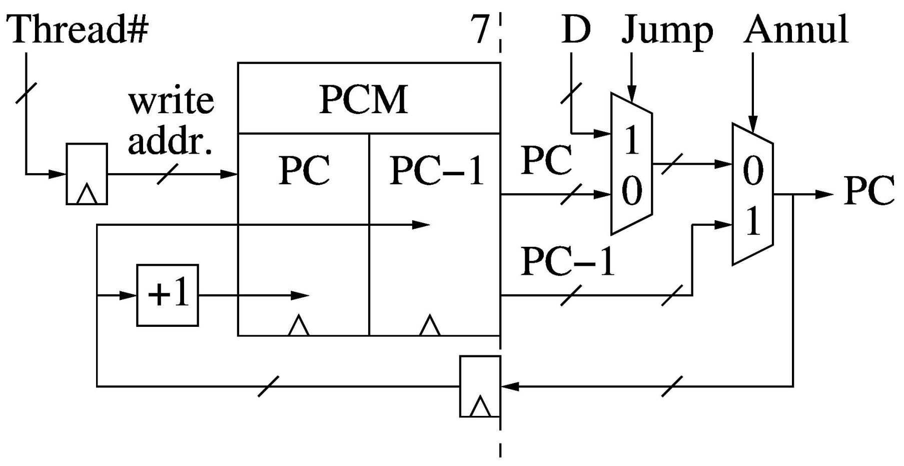


Figure 5.6: Modified Controller PC Memory, with instruction re-issue

5.2.5 Instruction Annulment and Re-Issue

Figure 5.6 shows the changes to the Controller (Figure 3.6), specifically the Program Counter Memory (PCM), required to re-issue an annulled instruction. A small thread number counter (not shown) indexes the PCM to read the Program Counter (PC) of the next thread instruction, as well as that of the current thread instruction (PC-1). As usual, if asserted, the Jump signal replaces the PC with the branch target address D. 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, thus we must also delay the value of the thread counter by one cycle to write back to the correct PCM entry.

5.3 Programming Model: Synchronization

Synchronization and Mutual Exclusion I/O handshaking allows us to use I/O ports to implement the inter-thread synchronization primitives that we found missing in Section 3.13.4.

For example, we can implement producer-consumer synchronization by directly connecting one I/O write port to an I/O read port: the producer thread writes data (or a token representing access to a buffer of data) to the I/O write port, and the consumer thread reads from the connected I/O read port. If one thread tries to run ahead of the other, it will "hang" until the other thread has caught up.

Alternatively, if the *same* thread both reads and writes a token via the same looped I/O ports, we can implement mutual exclusion: the first thread to read the port gets the token. Any other threads trying to read the same port will hang. When the first thread has completed its critical section, it writes the token back, and the first hung thread encountered will then gain access.

We can easily extend these basic mechanisms with counters, FIFOs, multiplexers, etc..., placed between the read and write I/O ports to implement work queues, multiple reader/writer locks, barriers, and so on.

SIMD Since each SIMD Lane (Chapter 4) has its own handshaking I/O ports, we can also create simple hardware mechanisms to handle divergent behaviour between Lanes. For example, we could contrive to automatically stall all other SIMD Lanes (and the controlling Scalar Octavo instance) if one Lane experiences a delay from external I/O, without involving the software.

Multi-Core Finally, we can connect Octavo cores in a mesh (Chapter 4) using handshaking I/O ports to transparently synchronize transfers between them, without having to explicitly manage any latency from external sources, or from different cores executing different code paths.

5.4 Results

Extending the write address space and implementing I/O handshaking reduced Octavo's average scalar F_{max} (see Chapters 3 and 4) by 4.5%, from 555 MHz to 530 MHz, with the peak F_{max} still reaching 550 MHz. The equivalent ALMs (eALMs) area, which accounts for RAM and DSP blocks (Chapter 4), rose by 5.5%, from 1027 eALMs to 1083 eALMs.

In exchange for the small loss in speed and increase in area:

- we eliminated useless code and data duplications across the A, B, and I memories, increasing the scope of programs on Octavo,
- we created 1024 words of "High" memory to contain hardware control registers, freeing the I/O ports,
- we eliminated the need for I/O busy wait loops, reducing the cycle penalty of testing for un-ready I/O ports from a minimum of 24 clock cycles down to 0,
- we created the basic framework for instruction annulment and re-issue, which sets the stage for implementing cancelling branches in Chapter 6.

Chapter 6

Approaching Overhead-Free Execution

In the von Neumann model, the control flow and dataflow instructions are embedded in the same program and executed sequentially. In the hardware implementation, the two "mechanisms" are separate. Instructions such as branches and jumps are not implemented on the FPGA. Instead, all branches of a control path are implemented and the correct outcome selected (i.e., "if conversion").

Zhi Guo *et al.* [51]

Implementing systems on FPGA soft-processors, rather than as custom HDL hardware, eases and accelerates the development process, but at the cost of a great reduction in performance. Orthogonal to limitations in parallelism or clock frequency, this reduction in performance primarily originates in the intrinsic addressing and flow-control overheads of scalar microprocessors, which expend a considerable number of cycles interleaving address calculations and branch decisions within the actual useful work.

In this chapter, we present an improved Octavo soft-processor which statically overlaps "overhead" computations and executes them in parallel with the "useful" computations, significantly reducing the number of processor cycles needed to execute sequential programs. The architectural changes reduce the average clock frequency to 0.939x of its original value, but also reduce the cycle count of all benchmarks, granting speedups from 1.07x for control-heavy code, up to 1.92x for looping code. The optimized benchmarks never perform worse than the original sequential code, and always better than a totally unrolled loop. This improved execution efficiency increases the range of FPGA designs amenable to soft-processors rather than custom hardware.

This work originally appeared at ICFPT 2014, Shanghai [74].

6.1 Introduction

Implementing computations in hardware on FPGAs can offer a significant speedup relative to computations in software running on a standard processor. Speedups of an order of magnitude have been reported (e.g.: [33, 85]), despite the FPGA hardware running at a considerably lower clock speed (F_{max}) than the processor. Much of this speedup arises from exploiting spatial parallelism in the FPGA hardware, but a significant portion comes from removing addressing and flow-control overhead ("support instructions" [38, 51, 52, 55, 110]). Hardware implementations remove such overhead by computing it in parallel with the actual work using application-specific Finite-State Machines (FSM) to accept/reject input, count loops, compute memory addresses, and perform the equivalent of flow-control by driving multiplexers.

However, implementing a given algorithm in hardware remains a difficult and laborious process, and hardware skills are comparatively rare vs. software skills [118]. Hardware design traditionally involves the use of hardware description languages (HDLs), which require specification at the bit level and explicit coordination of computation and communication, making design and debug challenging. Even higher-level HDLs such as BlueSpec SV, or compiling from OpenCL to hardware kernels, ultimately generate a low-level circuit description, which must then be synthesized, placed, and routed, taking hours or days for the largest designs.

As an alternative process, a designer can implement a “soft-processor” on the FPGA, benefiting from the flexibility of the FPGA, while retaining the programmability of software (Chapter 2). Unfortunately, soft-processors pay a large performance penalty relative to what the underlying FPGA hardware can achieve, and also relative to conventional “hard” CPUs.

More importantly, soft-processors carry the same addressing and flow-control overheads as any other processor. In most cases, unrolling loops and vectorizing code can eliminate addressing and flow-control overheads, but unrolling bloats code and increases the required instruction memory bandwidth, while vectorizing [105, 132] poses programming challenges to use effectively and cannot improve control code. Pipelining and multithreading [71, 77] will increase raw speed, but still do not eliminate overhead.

Rather than extending FPGA soft-processors with application-specific custom instructions or accelerators to avoid overhead, again involving more difficult hardware design, this chapter proposes an alternative solution: a soft-processor architecture which enables elimination of control-flow and addressing overheads, yet retains the ease-of-use of software. The key contribution of our work is to extend the Octavo soft-processor [77] (Chapter 3) to remove addressing and flow-control overheads:

- We locate the addressing and flow-control overheads in sub-graphs interleaved in a sequential program, which we can extract and execute in parallel.
- We introduce the Branch Trigger Module (BTM) as a means of executing *multiple* branches in parallel with an ALU instruction, based off arbitrary conditions. The BTM eliminates all the control instructions described in Chapter 3.7, freeing their opcodes for future use.
- We introduce the Address Offset Module (AOM), which enables per-thread private data for shared code, implements indirect addressing, and optionally automatically post-increments indirect addresses after use.

- We show, using several micro-benchmarks, that the BTM and AOM *always reduce cycle count*, without significantly affecting cycle time, and always more so than loop unrolling.
- We also show an increase in the ratio of useful work done, *often approaching the maximum provided by total loop unrolling*, without having to unroll loops.

6.2 Related Work

Previous branch folding approaches dynamically combined together an instruction and a following branch in the processor's instruction cache [42, 80] and speculatively executed conditional branches, flushing the pipeline on a misprediction. In contrast, Octavo's multi-threaded design never stalls or flushes its pipeline, avoiding the need to speculate on branches. Also, the BTM allows us to hoist the definition of a branch outside of loops – a critical feature in a system with an un-cached, tightly-coupled instruction memory without time for pre-fetching both branch paths [48, 66, 122]. Davidson and Whalley [36] describe a sophisticated system of branch registers similar to the BTM. However, their approach does not support folding, multi-way, or cancelling branches as the BTM does.

For vector processors, decoupling the scalar and vector components can provide the *appearance* of zero-overhead loops [129], but only if the scalar processor has time to execute loop control operations, while the vector processor operates. Truly eliminating loop overhead still requires loop counting and operand addressing hardware [31, 106, 107].

DSPs support zero-overhead loops with a variety of mechanisms: Analog Devices' TigerSharc [12] has a pair of loop counters and matching branch conditions, while their Blackfin [13] extends this approach with two sets of Loop Top and Loop Bottom registers, but only for simple counted loops. Finally, Texas Instruments' C64x+ [114] uses a loop buffer and some counter registers to execute compact software-pipelined loops. In contrast, the BTM provides a simpler and more general mechanism (though we have not yet implemented loop counters), and its support for cancelling branches enables useful branch folding in any flow-control code, not just loop branches.

Prior work on multi-way branches focused on improving ILP in VLIW processors [28,88]. Our work does not require the same complex CDFG analysis to merge branches, nor the generation of duplicate code. We use multi-way branches to reduce the number of consecutive tests on the same data by using multiple fast compares [62,87] in parallel.

6.3 Motivation and Overview

While our work improves the Octavo soft-processor [77] (Chapter 3), it also alters its programming model sufficiently that we can no longer compare code on both versions. For example, one of our improvements effectively implements memory indirection and eliminates the need for self-modifying code to implement pointers and array indexing (Section 3.12), a peculiarity of Octavo not normally present in microprocessors. We want to compare our architectural improvements against other architectures, not software emulations thereof. Thus, we must compare our work against an idealized hardware model which supersedes the original Octavo.

6.3.1 Baseline: A “Perfect” MIPS-like CPU

We can use the multi-threaded nature of Octavo to create a cycle-accurate emulation of an ideal “perfect” MIPS-like processor on our improved Octavo processor and then implement a micro-benchmark on both this emulated ideal model and natively on the improved Octavo. We do not need to compare against the original Octavo since the ideal model will always perform better than any actual scalar processor.

We can thus focus on the intrinsic overheads found in a general-purpose scalar processor, separate from implementation issues such as pipelining and architectural issues such as hazards, and show that, despite the absence of stalls and hazards in the ideal case, significant control-flow and addressing overheads remain relative to the actual desired computation. We then show how to extract these overheads as separate *parallel* sub-programs, and overlap their execution with that of the actual work-producing code on our improved Octavo processor.

Our baseline ideal MIPS-like system has the following properties:

- No memory access latency.
- Single-cycle instruction and data memory access.
- No load or branch delay slots.
- Branch conditions known at the start of pipeline.
- Result forwarding across pipeline stages.
- No structural hazards across instructions.
- Single-cycle instruction execution.

While we cannot build such an ideal CPU with high performance, each individual Octavo thread closely approaches this ideal, allowing us to execute a cycle-accurate emulation of the ideal model within a thread.

6.3.2 Benchmark: Hailstone Numbers

To overview our improvements, we calculate hailstone numbers as a simple, manually tractable example which nonetheless exhibits flow-control and addressing overheads. The hailstone benchmark iteratively computes a series from a positive seed number (if n is even: $n = n/2$, else $n = 3n + 1$), presenting many basic forms of computation (addition, shifting, multiplication, bit-masking, branching, looping). We apply this calculation to an array of 100 positive integers, terminated by -1, to introduce addressing overhead and average the time spent in the even/odd cases. We also output each new value as we compute it. Listing 6.1 shows the pseudo-code.

We directly translate this pseudo-code into MIPS-like assembly in Listing 6.2, with the instruction format convention of `OP dest, src1, src2`. This code, running on the

Listing 6.1: Hailstone Pseudo-Code

```
outer: seed_ptr = ptr_init
inner: temp = MEM[seed_ptr]
       if (temp < 0):
           goto outer
       temp2 = temp & 1
       if (temp2 == 1):            // Odd
           temp = temp * 3
           temp = temp + 1
       else:
           temp = temp / 2         // Even
       MEM[seed_ptr] = temp
       seed_ptr += 1
       OUTPUT = temp
       goto inner
```

Listing 6.2: Hailstone MIPS-like Assembly Code

```
outer:  ADD  seed_ptr, ptr_init, 0
inner:  LW   temp,  seed_ptr
        BLTZ outer, temp
        AND  temp2, temp,  1
        BEQZ even,  temp2
        MUL  temp,  temp,  3       // Odd
        ADD  temp,  temp,  1
        JMP  output
even:   SRA  temp,  temp,  1       // Even
output: SW   temp,  seed_ptr
        ADD  seed_ptr, seed_ptr, 1
        SW   temp,  OUTPUT
        JMP  inner
```

emulated ideal MIPS-like processor, averages 970 cycles per pass over 100 seed values with an *execution efficiency* (see below) of 0.655. Unrolling the same code results in an average of 769 cycles per pass and an efficiency of 0.824, a 1.26x improvement in both cycle count and efficiency.

We define *execution efficiency*, the ratio of "useful" to "not useful" instructions executed, as follows: Useful instructions are those which use the ALU and remain after total loop unrolling. Thus, loads, stores, and ALU operations count as useful. Pointer initialization and incrementing also count as useful if loop unrolling cannot eliminate them.

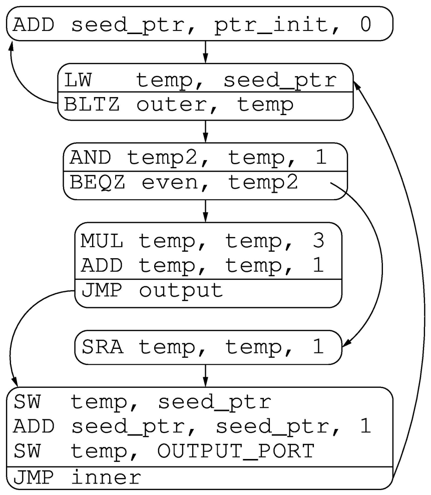
Listing 6.3: Hailstone Octavo Assembly Code

```
outer:  ADD seed_ptr, ptr_init, 0
inner:  LW  temp, seed_ptr
        MUL temp, temp, 3 ; BEVNn even ; BLTZn outer // Odd
        ADD temp, temp, 1 ; JMP output
even:   SRA temp, temp, 1                            // Even
output: SW  temp, seed_ptr
        SW  temp, OUTPUT  ; JMP inner
```

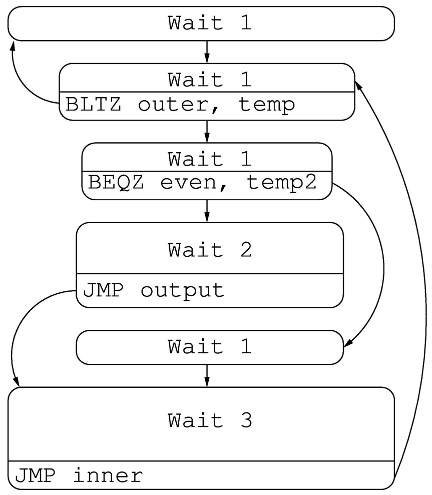
An unrolled loop will always approach an efficiency of 1.00, minus obligatory non-loop branches. Branches never count as useful by themselves since the ALU does no work during their execution.

We can then use our Octavo improvements to optimize the Listing 6.2 code into Listing 6.3, which shows the resulting optimized Octavo assembly code with the folded branches placed next to their concurrent ALU instruction. First, we add support for post-incrementing `seed_ptr` (eliminating line 11 of the MIPS code), add a "fast compare" [62, 87] to the result of the previous instruction to create a Branch on Even (`BEVN`, eliminating the Boolean masking on line 4), fold together both branches to `outer:` and `even:` (lines 3 and 5) into the start of the odd-number case (line 6) while also setting a "Predict Not Taken" bit (denoted by a suffix `n` on the branch instruction) to implement a multi-way cancelling branch which cancels the `MUL` instruction if we do not fall through into the odd-number case, and finally, we fold the unconditional `JMP`s into their preceding instructions (eliminating lines 8 and 13).

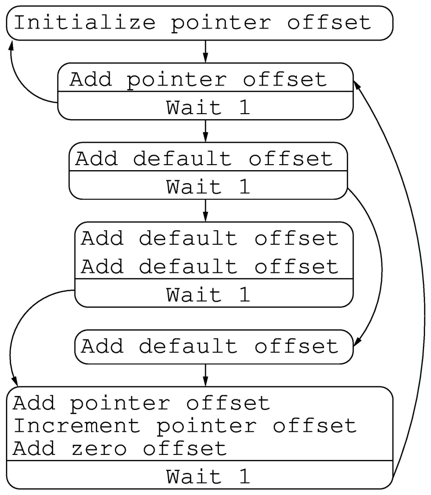
We later show in the results section that this optimized code, running on the improved Octavo processor, averages 504 cycles per pass over 100 seed values with an execution efficiency of 0.863, giving a 1.92x cycle-count speedup and 1.32x execution efficiency increase over the ideal MIPS-like CPU, exceeding the benefits of loop unrolling, while only reducing Octavo's clock frequency to 0.939x of its original value.



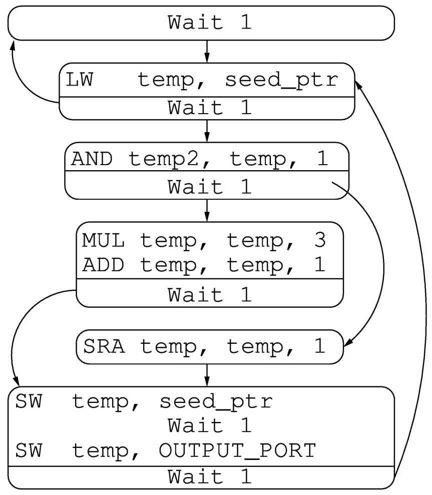
(a) Control-Data Flow Graph



(b) Flow-Control Sub-Graph



(c) Addressing Sub-Graph



(d) Useful Work Sub-Graph

Figure 6.1: Control-Data Flow Graph (CDFG) of the Hailstone benchmark, along with Flow-Control, Addressing, and Useful Work sub-graphs.

6.4 Extracting Control and Data Flow Sub-Graphs

Within a single sequential program, we can think of there being three separate "sub-programs", each of which is responsible for different tasks:

- The actual computational work.
- The flow-control to realize repetition and decision.
- Computing memory addresses.

We cannot practically eliminate the flow-control and addressing sub-programs, or else we would have to scale the actual work program along with its data, explicitly encoding each memory location into each instruction and duplicating the code for every repeated or conditional computation, at enormous cost in performance and code size [99]. Interleaving these three sub-programs imposes a sequential ordering to their operations. The core concept underlying our architectural enhancements to Octavo is to recognize that portions of these sub-programs are independent from one another and can therefore be executed in *parallel* with each other.

Fig. 6.1 illustrates how we can express all three sub-programs as sub-graphs of the original Control-Data Flow Diagram (CDFG) of the Hailstone benchmark. Figure 6.1(a) shows the original MIPS-like code broken into basic blocks. If we keep only the branch and jump instructions and replace the other instructions with a `Wait` instruction having an argument to represent the length of the body of the basic block, we end up with Figure 6.1(b), which describes the flow-control sub-program. Similarly, keeping only instructions which relate to addressing gives us the addressing sub-program in Figure 6.1(c), where we either add a fixed offset to each regular memory address (one per thread), a zero offset to any shared absolutely-addressed memory (e.g.: memory-mapped hardware), or a pointer offset to any memory location used as a pointer. We also keep instructions which alter the state of the addressing sub-program by initializing or incrementing offsets. Finally, removing any flow-control or state-altering addressing instructions from the initial CDFG leaves us with the actual useful work program in Figure 6.1(d). We can now visually find opportunities for executing flow-control and addressing in parallel with useful work, manifesting as instructions horizontally overlapping with `Wait` statements across sub-graphs, so long as no sequential dependency exists with the previous instruction.

Several approaches exist to execute parallel sub-programs such as superscalar processing, Very-Long Instruction Word (VLIW) computers, sub-units executing horizontal microcode, or multiple processors executing separate threads. However, these approaches require complex instruction scheduling hardware, larger instruction words and memory bandwidth, or require synchronization of multiple threads of execution.

Instead, we observe that the Program Counter (PC) suffices to represent the present location within *any* of the three sub-graphs, and that the flow-control and addressing sub-programs contain very little information: the flow-control sub-program spends most of its time waiting for a branching point, while the addressing sub-program simply adds a constant offset unless a pointer or I/O access happens. Viewed another way: even multiply nested loops have only a handful of repeatedly reached branches at any one time, and only the most memory-bound code would perform loads and stores more often than internal computations.

Furthermore, from the PC value, it is apparent at any point in the overall program which branching and addressing operation comes next. It suffices to provide this information to some machinery ahead of time (and ideally outside of busy loops), and let the PC and instruction operands indicate when special flow-control or addressing operations must happen.

Motivated by the discussion above, in the proposed processor, we reduce the execution of the flow-control and addressing sub-programs to pattern-matching on the set of conditions needed to generate the branches and address offsets required at the current point in the actual work program. This is achieved by using a small memory to encode the patterns to look for, along with associated matching logic.

Pattern-matching allows the designer to vary the number of entries to easily trade off area and speed against the need to reload pattern-matching entries as the program executes. At the limit case of a single entry, the performance simply returns the original case of sequentially interleaved sub-programs: each cycle saved by performing a flow-control or addressing operation in parallel with an ALU instruction returns as a cycle spent loading the entry for the next case. Ideally, the designer only needs to include enough entries to fully parallelize the branching and addressing operations inside the most critical sections of the program. Even matching a partial set of these operations will still improve performance some amount.

6.5 Implementation

Rather than executing the flow-control and addressing sub-programs on their own sub-processors, or as Instruction-Level Parallelism (ILP) in a superscalar, VLIW, or micro-coded processor (any of which would represent a significant departure from Octavo's scalar, multi-threaded architecture), we use the FPGA's capacity for fine-grained parallelism to check all sub-program conditions concurrently with the main instruction fetch. We use our knowledge of the current PC, the addresses in the instruction operands, and the result of the previous instruction, to select the desired effect on the flow-control and addressing of the current instruction as it flows through the pipeline.

6.5.1 Address Offset Module

The Address Offset Module (AOM) executes the addressing sub-program. We need three instances of the AOM, one for each instruction operand, as any one may access a pointer or other special memory at any time. The AOM adds a selected offset to the address contained in its associated instruction operand. The offset added may be 1) constant, 2) changing dynamically as the program executes (e.g. automatically incremented), or 3) 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. Dynamically changing offsets are useful for operations such as walking through an array. Zero offsets are useful for addresses that are shared across all threads, for example addresses of memory-mapped hardware in the system.

Fig. 6.2 shows the block diagram of the main AOM hardware. The numbers at the top denote the Octavo pipeline stage numbers. The instruction memory read happens at stage 0, and the data memory read happens at stage 4. Thus the AOM must do all its

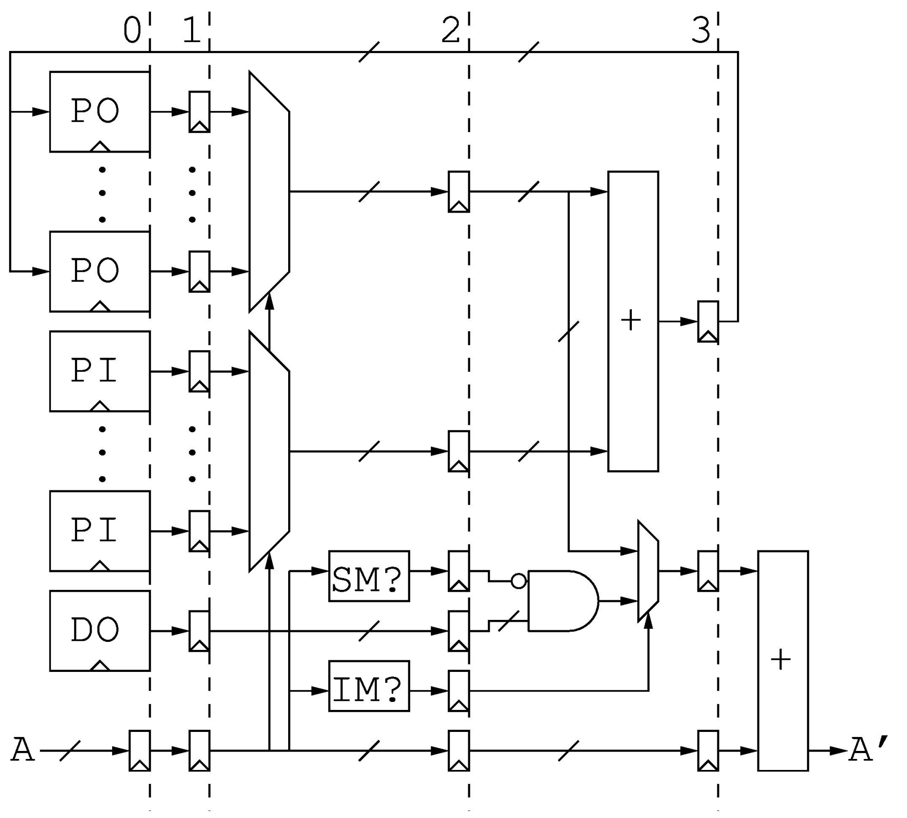


Figure 6.2: Address Offset Module implementation.

work in stages 0 through 4. Each thread running on the processor sees its own private AOM instances; specifically, the AOM memories are multiplexed using the thread index (0-7) (not shown).

In stage 0, the AOM reads a number of Programmed Offset (`PO`) and Programmed Increment (`PI`) memories, as well as a single Default Offset (`DO`) memory, all implemented using MLAB Block RAMs. Stage 1 serves to pipeline away the latency of the MLABs, and to receive the address (`A`) from the instruction operand. We cannot directly use the address (`A`) to select from the AOM memories as it is not available yet, and moving the memories forward in the pipeline leaves too few stages to maintain a high F_{max}.

In stage 2, we use the least-significant bits of `A` to select one of the `PO` and `PI` values. We also combinationally decode `A` to determine if it refers to a Shared Memory location (`SM?`) and/or an Indirect Memory location (`IM?`). The `SM?` and `IM?` logic modules simply decode part of the processor's memory map, set at design time, defining fixed ranges of memory locations which either act as absolutely-addressed shared resources (i.e.: I/O ports) or which act as pointers. Since pointers exist at fixed addresses, simply adding a Programmed Offset 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 stage 3, if `A` refers to a Shared Memory location, the AOM zeroes-out the `DO`. However, if `A` refers to an Indirect Memory location, the AOM discards the `DO` and instead selects the previously selected `PO`. In parallel, the AOM increments the `PO` with the `PI` and stores it back into its memory[1]. Finally, at the beginning of stage 4, just before the read from data memory, we add the final offset to `A`, yielding the final address `A'`. Programmed and Default Offsets have the same width as the instruction operand they modify: 10 or 12 bits. Programmed Increments currently use only 1 bit, for values of +1 or 0.

To use the AOM, the program must load `DO` with the offset for its own thread's memory region, and the `PO` and `PI` entries with the offset and post-increment values for the memory locations (set at design-time by the `IM?` decoder) which act as pointers.

6.5.2 Branch Trigger Module

The Branch Trigger Module (BTM) executes the flow-control sub-program. We need one BTM instance for each branch we wish to execute in parallel. The BTM monitors the Program Counter (PC) and some flags based on the result of the previous instruction: if the PC matches the location of a branch, and the flags match the branch condition, the BTM generates a destination address and signals the Controller (not depicted) to replace the current thread's next PC value with the destination address, performing a jump. The BTM optionally also outputs a signal to cancel the current instruction if the branch does not go as statically predicted, which allows us to always place a useful instruction in

[1]Note that we disable the write to `PO` if the instruction was anulled (Chapter 5). Also see Appendix E for a similar concern with BTMs.

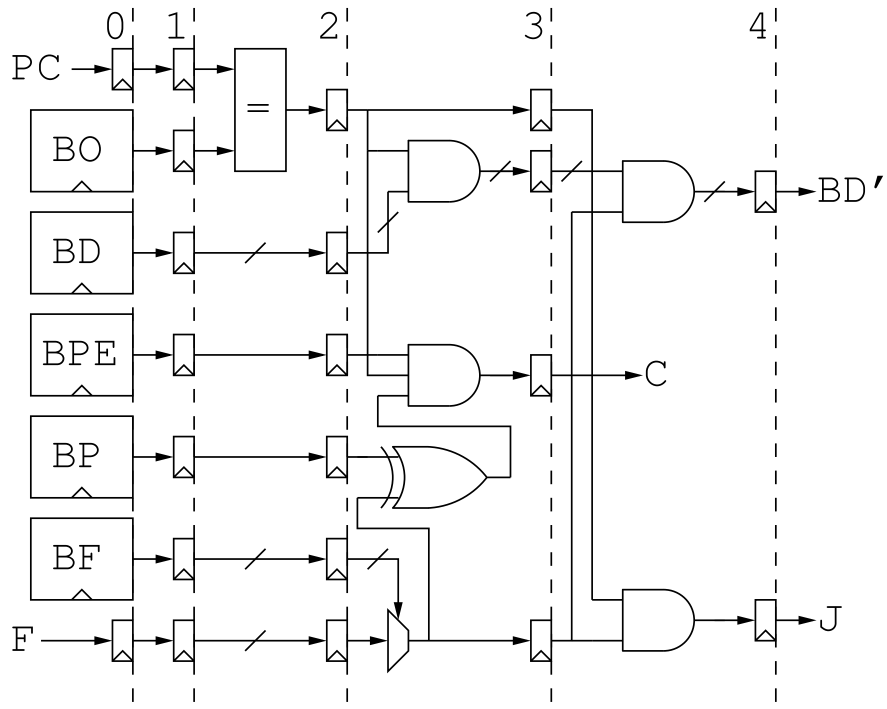


Figure 6.3: Branch Trigger Module implementation.

parallel with the branch. Also, if either the PC or the flags do not match, the BTM outputs all zeroes, which allows us to simply take the bit-wise OR of the output of multiple BTMs, rather than using multiplexers. Finally, having multiple BTMs operate in parallel incidentally enables multi-way branches for free.

Fig. 6.3 shows the block diagram of the main BTM hardware. The numbers at the top denote the Octavo pipeline stage numbers. The instruction memory read happens at stage 0, and any instruction annulling happens between stage 3 and 4, so the BTM must do its work between stages 0 and 4. However, merging the outputs of all the BTM instances into a single branching decision can happen later in the pipeline. Each thread running on the processor sees its own private BTM instances; specifically, the BTM memories are multiplexed using the thread index (0-7) (not shown).

In stage 0, the BTM reads a number of memories describing a branch operation: the Branch Origin (`BO`) contains the memory address of the branch, the Branch Destination (`BD`) contains the branch target address, the Branch Predict Enable (`BPE`) bit controls whether static branch prediction occurs or if the parallel instruction always executes, the Branch Predict (`BP`) bit selects between "Predict Taken" and "Predict Not Taken" instruction cancelling behaviour, and the Branch Flag (`BF`) selects one of the flags (`F`) derived from the result of the previous instruction as the condition for the branch. Stage 1 serves to pipeline away the latency of these MLABs Block RAMs.

In stage 2, the BTM compares the PC of the current instruction with `BO` and generates a match signal. In stage 3, we use this match signal to mask `BD`. We also select one of the branch flags (`F`) and compare it to `BP`[2].

Additionally, also in stage 3, if we enabled branch prediction for this branch (`BPE` set), and the selected flag and the branch prediction disagree, and the branch origin matches, we generate a signal (`C`) to cancel the current instruction in parallel with the branch. A cancelled instruction converts to a no-op but, unlike an annulled instruction, does not re-issue later.

Finally, in stage 4, we use the selected flag to mask `BD` again, and the `BO` match signal to mask the selected flag, resulting in the final branch destination `BD'` and jump signal `J`. If either the branch flag or the branch origin do not match, both `BD'` and `J` are zero.

Branch Origin and Destination memories have the same width as the Program Counter (10 bits), while the Branch Predict Enable, Branch Predict, and Branch Flag memories have 1, 1, and 3 bits respectively.

To use the BTM, the program must load all the memories with the values describing an upcoming branch. We automatically compute the branch flags (`F`) from the result of the previous instruction, using separate dedicated logic instead of the main ALU, in the manner of Katevenis' "fast compare" [62, 87]. Currently, we support 8 flags total: Negative, Positive, Zero, Non-Zero, Always (for JMPs), and Even, with 2 flags still left unused. Since all BTM instances function concurrently and eventually merge their decision output, we can easily support multi-way branches so long as all branches with matching origins have mutually exclusive branch conditions.

[2] We omit details related to I/O Predication from Chapter 5. See Appendix E.

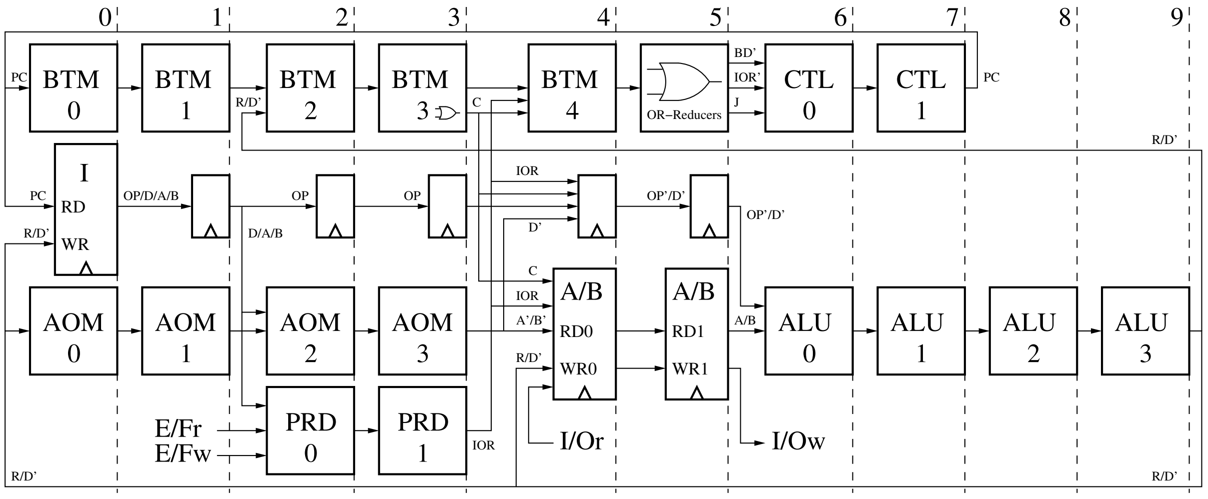


Figure 6.4: Octavo Block Diagram with I/O Predication, AOM, and BTM

We can support as many branches in a region of code as there are BTM instances. For example, two BTM instances would support two simple nested loops without the need to reload the BTM memories during their execution. Any additional branches (e.g.: conditionals) would require corresponding BTM instances. We can manage a limited number of BTM instances in a manner similar to register allocation, keeping only the "hottest" branches in the BTMs and re-using one BTM for the less-frequently encountered branches.

6.6 Improved Octavo Processor Configuration

Figure 6.4 shows the block diagram of the improved Octavo soft-processor which integrates the AOM and BTM. Starting from the original design in Figure 3.7 (pg. 42), we added I/O Predication (`PRD`) from Figure 5.2 (pg. 89) (present, but not used in the following benchmarks), plus the new Branch Trigger and Address Offset Modules (`BTM` and `AOM`) from Figures 6.2 (pg. 109) and 6.3 (pg. 111).

6.7 Experimental Methodology

Our experiments target an Altera Stratix IV FPGA of the highest speed grade. We test our circuits inside a synthesis test harness to ensure an accurate timing analysis. We average the Quartus synthesis results over 10 random initial seeds, and tune Quartus to produce the highest-performing circuits possible. Appendices A and B describe the experimental framework and Quartus settings in detail.

To measure the number of cycles spent in each benchmark, we place our design in a simple testbench, which provides a clock and required I/O signals, and execute each benchmark on a delay-free HDL simulation of the Octavo processor using Modelsim 10.1d. Since we already know the achievable F_{max} of the Octavo processors' HDL description, the simulation cycle-count accurately reflects the benchmark wall-clock time. Each simulation runs for 200,000 cycles total, with one Octavo thread running the benchmark, and the other seven stalled in an infinite loop without any effect on the system.

6.8 Evaluation, Benchmarks, and Results

Octavo has, as one of its major features, the capacity to approach the maximum possible clock frequency supported by the underlying FPGA. Thus, we evaluate how many Address Offset Module (AOM) entries and Branch Trigger Modules (BTM) instances we can include, and in what ratios, before the raw clock frequency begins to suffer too much. We consider an F_{max} of 500 MHz as a realistic goal, still at 0.909x of the limiting 550 MHz absolute maximum rating for simple dual port M9K Block RAMs in Stratix IV devices of the highest speed grade [10], and more representative of the actual rated limit of most such devices.

Our implementation aimed for flexibility and generality for design space exploration, describing each individual memory in the AOM and BTM as separate MLABs, regardless of depth of width. We have not yet analyzed the synthesis results to determine if Quartus

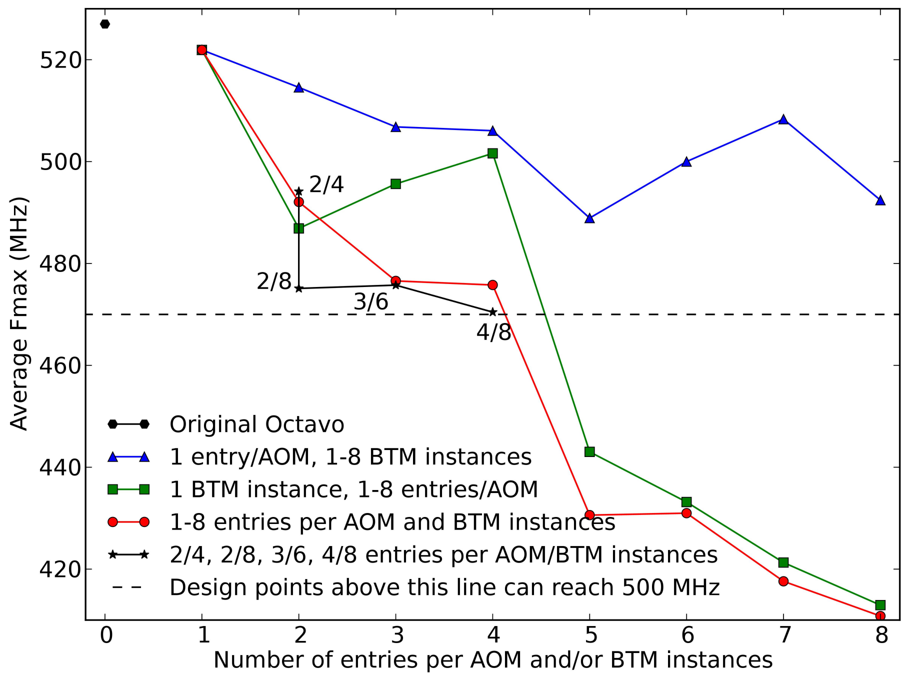


Figure 6.5: Average Octavo F_{max}, varying the number of entries per AOM instance and the number of BTM instances.

can automatically merge separate but logically contiguous MLABs. Therefore, we do not know if the area results will be comparable across design points, and cannot report detailed area results at this time.

Fig. 6.5 outlines the major features of the design space, charting the F_{max} of Octavo versions with 0 to 8 AOM entries and/or BTM instances, and a few relevant combinations thereof. All values represent the average achievable clock frequency. All points above the horizontal dotted line (470 MHz) denote designs which can reach or exceed 500 MHz after place-and-route. Meaning that for such designs, across 10 place-and-route runs, at least one produced an implementation whose F_{max} reached or exceeded 500 MHz.

The single hexagon in the top-left corner represents the original Octavo at 527 MHz (averaged across 10 P&R runs). The triangle markers show the improved Octavo with a minimal set of 3 single-entry AOMs and 1 to 8 BTMs. Since the BTM modules avoid multiplexing and sit in a long, sparsely used pipeline, the average F_{max} scales quite well, staying above 490 MHz at up to 8 instances, and staying above the dotted line at up to

16 instances (not shown).

Conversely, the square markers show an improved Octavo with 1 BTM instance, and AOM instances with 1 to 8 entries each. Note that since each addressing path to memory requires an AOM instance, we must always have three instances: one to support each instruction operand. Because of these multiple instances, their use of multiplexing, and the tighter pipeline between instruction fetch and data read, F_{max} scales poorly as the number of entries increases, falling off quickly for case with more than 4 entries per AOM.

The circle markers show the line through the design space with equal numbers of BTM instances and entries per AOM, 1 through 8, suggesting that the number of AOM entries tends to dominate the scaling of the system as a whole.

Finally, given the previous lines through the design space, the star markers point out some useful configurations: 2 AOM entries with 4 BTM instances (2/4), 2 AOM entries with 8 BTM instances (2/8), 3 AOM entries with 6 BTM instances (3/6), and 4 AOM entries with 8 BTM instances (4/8), which likely represents the largest useful configuration which can still reach 500 MHz after place-and-route.

For the remaining results in this section, we used the first point, 2 AOM entries with 4 BTM instances (2/4), as our benchmarking configuration since some benchmarks will require more than these resources, demonstrating benefits even without complete AOM/BTM support and at 0.939x of the original average F_{max} (495 MHz, down from 527 MHz). The peak F_{max} still reaches 510 MHz – 0.927x of the absolute maximum 550 MHz rating. The total equivalent ALMs (eALMs) area, which accounts for RAM and DSP blocks (Chapter 4), rose 1.73x from 1087 eALMs to 1878 eALMs[3].

6.8.1 Benchmarks

Since no compiler currently supports AOM/BTM functions, we wrote our own set of micro-benchmarks in assembly. We previously described Hailstone in Section 6.3.2, and the other benchmarks in Appendix H.

[3] but see caveat about area near beginning of section.

These benchmarks represent various kinds of sequential programs, including simple loops, complex branching, and numerical processing, and sometimes show the behaviour of our improvements under non-ideal conditions where we cannot eliminate all overhead. All benchmarks run under an outermost infinite loop, and we base our measurements on the number of completed benchmark passes over 200,000 simulation cycles.

We wrote the benchmarks in a strict load-compute-store form, emulating a MIPS-like processor. Normally, Octavo's register-less, flat memory model combines loads and stores with ALU operations (Chapter 3.8), which would make the comparison unfair. Specifically, we would be unable to isolate the speedup and efficiency improvement that arise solely from incorporating the BTM and AOM, as some additional speedup/efficiency would originate from the elimination of loads/stores to/from registers. Thus, for fairness, we report *conservative* results, avoiding Octavo-specific advantages and measured against the emulated ideal MIPS-like CPU, affected only by our use of BTM and AOM. Furthermore, all processing happens at Octavo's native word width of 36 bits.

Table 6.1 summarizes the cycle count speedups and execution efficiency improvement of our benchmarks. We measure efficiency as the ratio of "useful" instructions (Section 6.3.2). We show both looping and unrolled versions, as unrolling reveals the fundamentally useful instructions. However, unrolling itself impractically increases code size to nearly fill Octavo's instruction memory. The "MIPS" entries refer to benchmarks run on the emulated ideal MIPS-like CPU, and the "Octavo" entries refer to optimized benchmarks using AOMs and BTMs. Reading across columns shows the impact of the AOMs and BTMs, while reading down columns shows the impact of loop unrolling.

Hailstone We previously described the hailstone benchmark in Section 6.3.2. Hailstone exercises various computing and branching operations. When using the AOMs/BTMs, we see a 1.92x speedup and 1.32x efficiency increase. Using AOMs/BTMs on unrolled code yields worse results since we must keep re-loading the BTM with branch data for each unrolled loop instance.

Table 6.1: Benchmark cycle count speedup and efficiency improvements.

Benchmark	Cycles per Pass			Execution Efficiency		
Hailstone	MIPS	Octavo	Speedup	MIPS	Octavo	Increase
Looping	970	504	1.92x	0.655	0.863	1.32x
Unrolled	769	701	1.10x	0.824	0.899	1.09x
Speed./Incr.	1.26x	0.72x	—	1.26x	1.04x	—
Increment	MIPS	Octavo	Speedup	MIPS	Octavo	Increase
Looping	716	376	1.90x	0.631	0.907	1.44x
Unrolled	431	331	1.39x	1.000	1.00	1.00x
Speed./Incr.	1.66x	1.14x	—	1.58x	1.10x	—
Reverse	MIPS	Octavo	Speedup	MIPS	Octavo	Increase
Looping	404	354	1.14x	0.748	0.856	1.14x
Unrolled	309	309	1.00x	1.000	1.000	1.00x
Speed./Incr.	1.31x	1.15x	—	1.34x	1.17x	—
FIR	MIPS	Octavo	Speedup	MIPS	Octavo	Increase
Looping	2902	2502	1.16x	0.897	0.960	1.07x
Unrolled	2614	2406	1.09x	0.996	0.998	1.00x
Speed./Incr.	1.11x	1.04x	—	1.11x	1.04x	—
FSM	MIPS	Octavo	Speedup	MIPS	Octavo	Increase
—	807	753	1.07x	0.564	0.467	0.83x

Array Increment We increment an array of 10 elements by 1, repeated 10 times, then output the entire array, showing a simple iterated calculation with a separate output loop. This benchmark requires five branches, forcing us to periodically reload one of the four BTM entries to support it. However, we can hoist that overhead outside of the loop, and still obtain a significant speedup. Using AOMs/BTMs grants a speedup of 1.90x and an efficiency improvement of 1.44x. The efficiency suffers, relative to loop unrolling, due to the overhead of reloading the BTM with data for the fifth branch, associated with the inner loop. See Appendix H.1 for source code.

Array Reverse We traverse an array of 100 elements using two pointers, top-to-middle and bottom-to-middle, loading and storing to swap their values without any computation or other I/O. This memory-bound benchmark forces the bottom-to-middle pointer to decrement, which our current AOM does not support. Furthermore, due to the write-only nature of the AOM, we have to keep a copy of the bottom-to-middle pointer, add -1

to it, then update its AOM entry, all within the main loop. Nonetheless, AOMs/BTMs enable a speedup and efficiency increase of 1.14x. See Appendix H.2 for source code.

8-Tap FIR Filter We sequentially read a 100-entry input buffer, applying an 8-tap FIR filter at each step, and output the filtered values to a 100-entry output buffer. We keep all 8 taps and 8 coefficients in registers, shifting values down the taps then performing the multiplications and additions, both as unrolled inner loops. Despite the sequential bulk of the buffering and convolution code, AOMs/BTMs speed up FIR by 1.16x and improve its efficiency by 1.07x. Note that available memory limited us to unroll 25 times instead of 100, thus some loop overhead remained in all cases. See Appendix H.3 for source code.

Floating-Point Number FSM We parse a stream of 100 characters using a Finite State Machine, looking for simple space-delimited floating point numbers such as 5.5, −3., +.8, etc..., and raise either an Accept or Reject signal. The 100 input characters contain 25 valid numbers and one invalid one, forcing the FSM to walk through all its paths. We implement the FSM directly in the program structure, alternating tests and branches to determine state and action. FSM contains no real loops, no hoistable computations, no significant addressing, no basic block longer than 2 or 3 instructions, and has 34 unique branches, greatly exceeding the capacity of the four-entry BTM and forcing us to continually reload a single BTM entry with the data for the next upcoming branch. However, we can fold that BTM entry reload (for the next branch) with the branch set by the previous reload, and we can use the BTM to statically cache three other branches to save cycles on the most commonly traversed edges, granting a 1.07x speedup and 0.83x efficiency improvement. We cannot unroll FSM due to its complex and variable execution, and efficiency suffers due to some BTM re-loads not folding into branches. See Appendix H.4 for source code and algorithmic details.

Summary The use of BTMs and AOMs *always* speeds up sequential code, conservatively measured relative to a "perfect" MIPS-like CPU, and even taking into account the 0.939x change in F_{max} from their implementation. Using AOMs/BTMs also improves execution efficiency of our benchmarks to a minimum of 0.856x of the maximum achievable via loop unrolling, up to a 1.05x (0.863/0.824) improvement for Hailstone, without the associated code size increase.

6.9 Programming Model: Parallel and Shared Code

Introducing the AOM and BTM improves Octavo's programming model in two major ways when exploiting data parallelism with multiple Octavo threads running identical code on subsets of the entire data: we eliminate the amplification of branching overhead, and the need to create per-thread private code copies, as we originally described in Section 3.13.2.

Since branches now effectively execute in zero cycles instead of one, we eliminate long bubbles of branches from Octavo's pipeline when all threads reach the same branch when executing the same code in lock-step. Also, since each thread now has a global private address offset into memory and private special offsets to implement indirect memory accesses, all the threads can now share the same code, without self-modifying private copies as described in Section 3.12. We only need small, thread-private code preambles to initialize the BTM and AOM entries of each thread before branching to the common, shared code.

6.10 Conclusions

In this work, we located intrinsic overheads in sequential programs as separate addressing and flow-control sub-graphs interleaved with the actual desired computations. We extracted these sub-graphs as data describing sets of conditions the processor can pattern-match against, using Address Offset Modules (AOMs) and Branch Trigger Modules (BTMs) operating in parallel with the instruction fetch.

We found AOMs and BTMs to scale to useful levels, supporting up to 4 active pointers per instruction operand and up to 8 active branches before the average F_{max} dropped below 500 MHz on a modified Octavo soft-processor on a Stratix IV FPGA. Furthermore, benchmarking revealed that using BTMs and AOMs *always* reduced the cycle count, speeding up execution by 1.07x for branch-heavy control code, up to 1.92x for looping code. In all but one case, which had incomplete AOM support, the speed-up of ordinary looping code *always* exceeded that granted by total loop unrolling, since AOMs/BTMs can additionally eliminate pointer increments and fold branches with useful work.

Using AOMs and BTMs also improved the ratio of executed instructions which perform the actual desired computations. Versus the best-possible case granted by total loop unrolling, which often reaches a ratio of 1.00, our benchmarks improved their efficiency to a minimum of 0.856, reaching up to 1.05x the efficiency of unrolled code, all without the associated code size increase. Only control-heavy code, impossible to unroll, suffered a decrease in efficiency from 0.564 to 0.467.

Chapter 7

Benchmarking Overlay, HLS, and HDL Implementations

The most exciting phrase to hear in science, the one that heralds new discoveries, is not "Eureka!" (I found it!) but "That's funny..."

Isaac Asimov (1920 - 1992)

In previous chapters, we defined the Octavo overlay architecture, preserved its performance under tiling, improved its system-level flexibility, and reduced its execution overhead. We now benchmark Octavo against other FPGA design alternatives to compare their sequential and parallel performance and highlight directions for improvement.

We benchmark the three major FPGA design process alternatives we originally described in Chapter 2: Overlays, High-Level Synthesis (HLS), and Hardware Description Language (HDL). We compare Overlays to uncover sources of overhead and suggest improvements to Overlay architecture in general. We also create equivalent HDL implementations, implementing the same algorithms and tuned for maximal performance, to estimate the performance gap and (more importantly) the area penalty of Overlays. Finally, we compare HLS against both Overlays and HDL since HLS has similar ease of development as Overlays, with a final area and performance closer to HDL, but without the fine implementation control of the latter.

Overlays Unfortunately, most overlay architectures (Section 2.2.3) never leave the lab (e.g.: MARC [79]), leaving only two to compare: our own Octavo [77, 78], and MXP [105, 107], a vector soft-processor derived from UBC's VENICE [106] 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.

High-Level Synthesis There exist multiple HLS systems (Section 2.2.2), but we compare Octavo to LegUp [22] as it synthesizes plain C code, making the benchmarks more portable to MXP and Octavo than OpenCL kernels which assume a certain memory model [109], or higher-level Bluespec descriptions which describe parallelism as Guarded Atomic Actions [95]. We use LegUp 3.0, with some in-progress improvements to pipelining and local memories [21, 46], in a pure hardware flow without the included Tiger "MIPS" [89] core normally used to execute code. LegUp uses 32-bit values for synthesis, automatically optimized by Quartus where possible.

Hardware Description Language Finally, for each benchmark, we manually create a hand-optimized Verilog-2001 implementation which executes the benchmark in the same manner as the Overlay and HLS solutions, aiming for the highest performance possible without changing the underlying algorithm. We report the unrestricted F_{max}, to show that most HDL implementations can exceed the 550 MHz BRAM limit, maximizing the use of the FPGA for that computation. All HDL implementations work with 32-bit data, and narrower words as required for addresses, counters, etc...

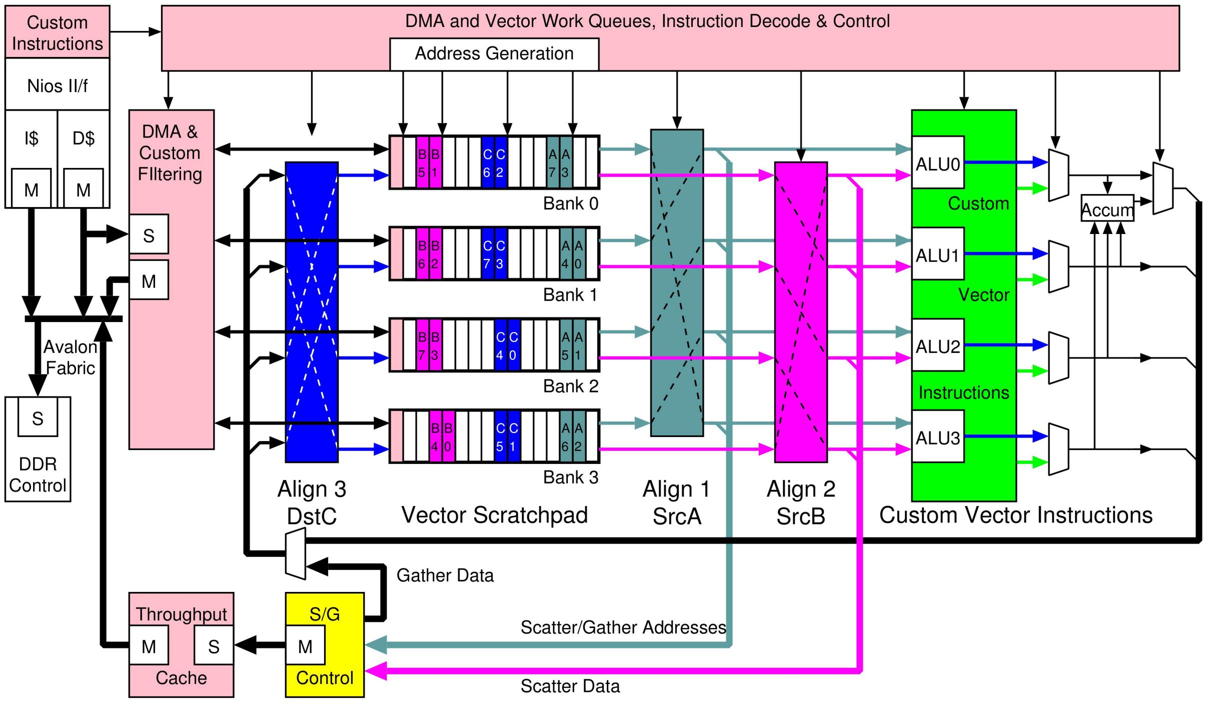


Figure 7.1: MXP Block Diagram [107]

7.1 MXP

Figure 7.1 shows the block diagram of a 4-lane example configuration (denoted as MXP-V4) of the MXP soft vector processor [107]. In a nutshell, the MXP vector lanes operate under the control of a Nios II/f scalar soft-processor which sends vector instructions to the various MXP units to perform vector DMA transfers to/from main memory, scatter/gather addressing [108], and optional custom vector instructions for specific tasks.

7.1.1 Relevant Architectural Features

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: 2 vector reads, 1 vector write, and a DMA transfer to/from main memory. However, the double-pumped scratchpad ultimately limits MXP's overall 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 7.1 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 which provide basic conditional execution in the absence of vector branches.

Custom Vector Instructions We can add special-purpose functional units to MXP's pipeline to accelerate application-specific code, at the cost of re-synthesizing the entire processor. In this work, the designers of MXP added custom vector instructions to support per-vector-lane table lookups to enable the sequential Hailstone-A benchmark.

7.1.2 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 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 2-D 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.

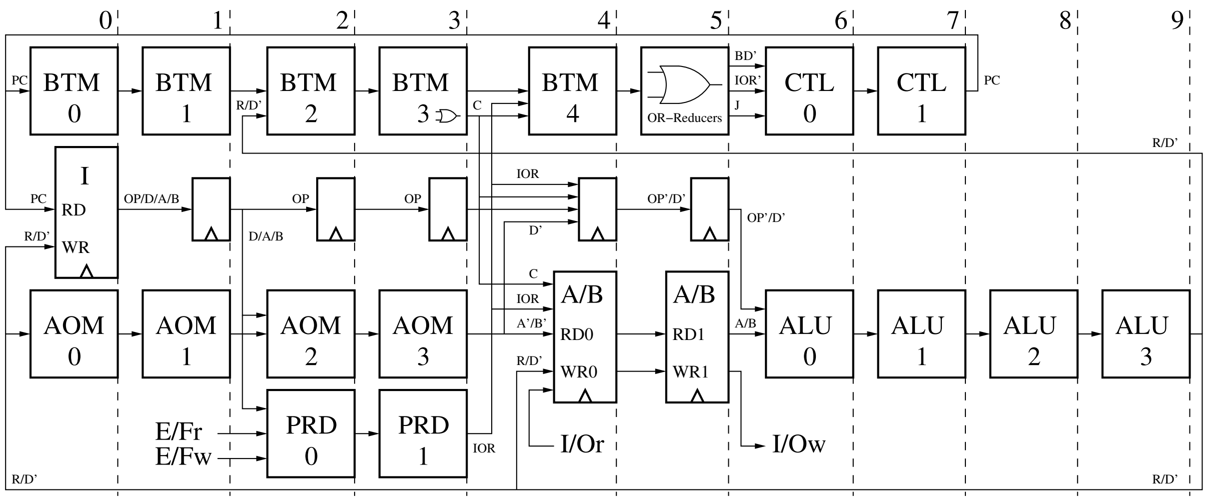


Figure 7.2: Completed Scalar Octavo Block Diagram

7.2 Octavo

Figure 7.2 shows the block diagram of the completed scalar Octavo soft-processor (denoted as Octavo-L1). Starting from the original design in Figure 3.7 (pg. 42), we added I/O Predication (`PRD`) from Figure 5.2 (pg. 89), plus Branch Trigger and Address Offset Modules (`BTM` and `AOM`) from Figures 6.2 (pg. 109) and 6.3 (pg. 111). Replicating the `A/B` memories and the `ALU` provides SIMD parallelism (Figure 4.1, pg. 62), with each SIMD lane logically partitioned to improve F_{max} (Chapter 6).

7.2.1 Relevant Architectural Features

Strict Round-Robin Multi-Threading Each cycle, one of 8 independent threads, sharing the I/A/B memories, issues an instruction in turn. The order *never* varies. This strict multi-threading, more than any other feature, enables Octavo's high F_{max} and dictates its programming model.

Unified Scratchpad Memory Octavo does not separate register file and memory, using scratchpad memories instead and memory-mapping hardware into those address spaces. Both instruction read operands address separate scratchpad memories (A and B), and the destination operand can write to any memory (Chapter 5.1).

Memory-Mapped I/O Ports Instead of loads/stores, Octavo maps I/O port into the A/B memories, addressing them using the instruction operands like any other memory location. This external interface allows tight integration of external accelerators for special functions.

Accelerators To approximate MXP's vector accumulator and alignment networks, we attached an Accumulator to the I/O ports of each datapath to provide chained Multiply-Accumulate operation for the parallel Finite Impulse Response (FIR) benchmark. We also similarly inter-connected the datapaths with pipelined channels to support the parallel Reverse-4 benchmark. We describe these additions below.

Branch Trigger Module The BTM enables multi-way zero-cycle branches based on the result of the previous thread instruction, possibly cancelling the current instruction if the branch does not go as statically predicted (Chapter 6.5.2).

Address Offset Module The AOM enables indirect and offset memory addressing to share code across threads, and can concurrently perform post-incrementing addressing (Chapter 6.5.1) after an indirect memory read/write.

SIMD Lanes By replicating the datapath (A/B memories and the ALU), and feeding the copies a common instruction stream, we create optional SIMD lanes. Each lane has its own A/B memories and 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. Contrary to previous SIMD implementations shown in Figure 4.1 (pg. 62), we do not pipeline the SIMD lanes one instruction behind the primary datapath. We thus exchange a 2-3% loss in average F_{max} for a simpler programming model, especially when SIMD lanes communicate with each other and the primary datapath.

7.2.2 Programming Model

The Octavo programming model strongly depends on multi-threading the target application. Each of the 8 threads executes an instruction in turn, at an effective $\frac{1}{8}^{th}$ the actual F_{max}. Thus, to use Octavo's full capacity, parallel software must divide the work across all threads, each executing the same code on a subset of the data. Adding SIMD parallelism simply extends this model since each additional datapath executes the same 8 threads, but operating on different data subsets. Once initiated, all threads proceed efficiently without branching overhead or pipeline hazards. Unfortunately, each parallel program must be tailored to the particular Octavo configuration that will execute it, manually distributing data to each thread before computation. We discuss the details of Octavo's programming model in Sections 3.13, 4.9, 5.3, and 6.9.

7.2.3 Accelerators

Customizing MXP requires directly altering its datapaths to add custom vector instructions. However, on Octavo, we can simply attach custom hardware to the I/O ports, without touching the datapaths or the ISA. Thus, rather than providing general, built-in solutions for a broad set of problems, we make it simple to add new primitives to Octavo and to integrate them into its programming model.

The first Accelerator, the Array Reversal Channel, implements a permutation tailored to array reversals. It was far easier to design a specialized channel than an exact equivalent to MXP's vector alignment networks. Making changing Accelerators easy favours creating simpler, specialized solutions. The second Accelerator, the Accumulator, implements the same Multiply-Accumulate functionality as in MXP. However, we do end up with multiple Accumulators as a natural extension into SIMD parallelism.

Note that a designer could also add array reversal permutations to the MXP's alignment networks, and also add an accumulator to each vector lane ALU, yielding identical

Listing 7.1: Pseudo-code to drive Array Reversal Channel

```
ADD I/Ow,    Top,     0
ADD I/Ow,    Bottom,  0
ADD Bottom,  I/Or,    0
ADD Top,     I/Or,    0
```

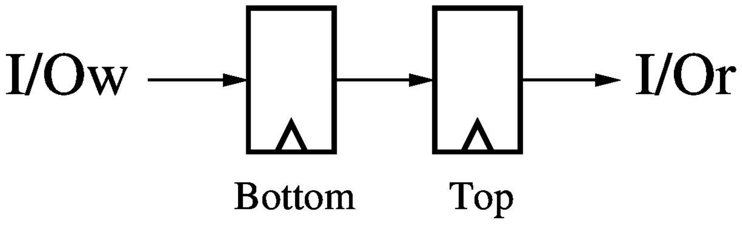


Figure 7.3: Per-Thread View of Array Reversal Channel

functionality to Octavo's Accelerators. However, these changes would require considerably more design and implementation effort, and would likely reduce the F_{max}, since they would lie in the datapath. On the other hand, designing, implementing, verifying, and attaching these Accelerators to Octavo took only about two evenings' worth of work each, and had negligible impact on F_{max}.

Array Reversal Channel

To enable communication across its vector lanes, MXP has read/write alignment networks which can offset a vector across lanes as desired. In contrast, Octavo's SIMD lanes do not communicate at all, except for receiving a common instruction stream, and could not execute an equivalent parallel array reversal benchmark.

Listing 7.1 outlines the work of a thread reversing an array possibly spanning multiple SIMD lanes: a thread reads (by `ADD`ing to `0`) from the Top and Bottom array pointers, writing the data to the write port (I/Ow), then reads from the read port (I/Or) and writes the received data back to the array in reverse order.

Figure 7.3 shows the communication channel between two such threads: the writing thread writes the target of the Top and Bottom pointers to a write port (I/Ow), in that

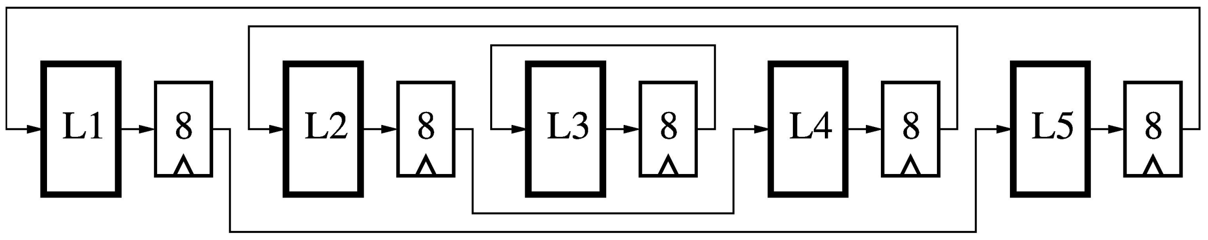


Figure 7.4: Octavo-L5 Lanes with 8-stage Array Reversal Channels

order, to a 2-stage pipeline which the receiving thread then reads from a read port (I/Or) and stores in reverse order.

Since Octavo executes 8 threads, and each thread needs to see a 2-stage pipeline, we need 16 pipeline stages total to contain all the writes before we begin reading them back. 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.

Figure 7.4 shows the channel connections for parallel array reversal across the 5 lanes of an Octavo-L5: if we distribute an array across all 5 lanes, then the first 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.

SIMD Accumulators

For a Finite Impulse Response (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.

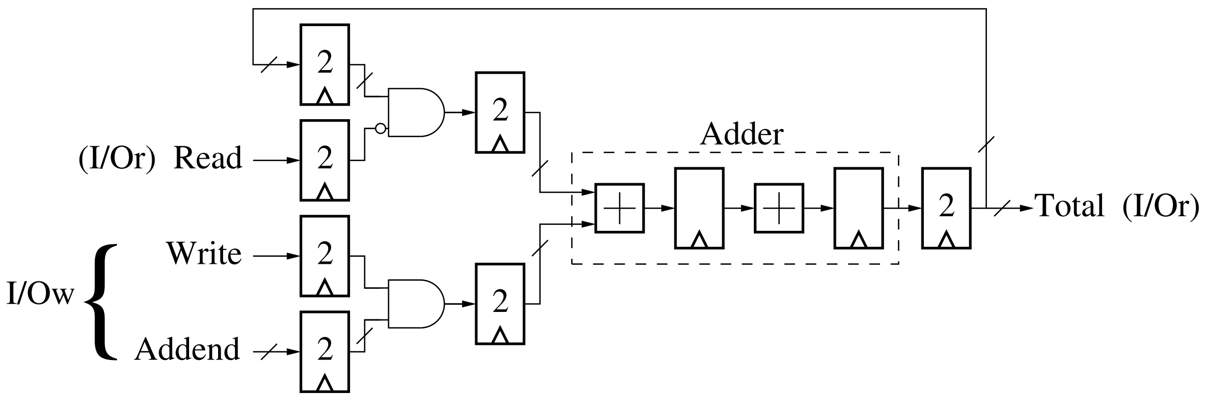


Figure 7.5: Pipelined Per-Lane Accumulator, with 2-stage delays

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 7.5 shows the Accumulator implementation. In the centre, we have a copy of Octavo's Adder (Chapter 3.9), which takes 2 pipeline stages. We add additional pipeline registers, in groups of 2 for convenience, to add up to 8 pipeline stages. After writing to the Accumulator, the new Total sum shows up at the output 8 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 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.

7.3 Benchmarks

We use an extended set of the benchmarks used to measure execution overheads in Chapter 6, further divided into Sequential and Parallel groups. For each benchmark, we measure the following values and calculate their ratio relative to those of a scalar Octavo core (Octavo-L1) which forms the baseline for all speed and area comparisons.

- 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 in the results of each benchmark (Section 7.4). Typically, the computation of one output value defines the unit of work.

- The F_{max} of the implementation. Contrary to the rest of this work, we choose the implementation with the *highest* F_{max} out of 10 random seed P&R runs so as to compare the best possible case of each approach.

- 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.* [125, 126]. 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.

Experimental Framework All benchmarks execute entirely from on-chip memory to avoid the complications of a memory hierarchy, with the exception of some NiosII/f results which access external memory via a direct-mapped cache. The MXP system uses GCC 4.1.2 to compile code to the NiosII/f controlling processor, and synthesizes the vector lanes using Altera 13.0sp1. We use Quartus 13.1 to synthesize Octavo (Appendix A), with an aggressively tuned configuration for maximum F_{max} (Appendix B). LegUp uses Quartus 10.1sp1 for synthesis, while the HDL implementations use Quartus 13.1. Both also use the same aggressive configuration as the Octavo synthesis. All systems target the Altera Stratix IV `EP4SGX230KF40C2` device on the Terasic DE4-230 board.

Data Types All overlay benchmark implementations operate on word-wide signed integers: 32 bits for MXP and LegUp, and 36 bits for Octavo. Both also use whole words for addressing and counting. The HDL implementations use 32 bits 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.

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 set-up overheads and avoid artificial inefficiencies such as pipeline hazards, while still fitting into on-chip memory.

Listing 7.2: Reverse-2D MXP Pseudo-Code

```
for(i = 0; i < COUNT; i++){
  temp_vector[(COUNT-1)-i] = data_vector[i];
}
data_vector = temp_vector;
```

Therefore, we report benchmarks results per processed working set item (“unit of work”) to abstract away the different raw working set sizes.

In general, the HDL and LegUp benchmarks work on sets of 100 items, while Octavo works on sets of 1024 items, evenly divisible into per-thread working sets of 128 elements, and replicates the working set for each additional SIMD lane.

MXP scales the working sets to the size of its scratchpad, divided by 4 to leave space for temporary vectors. MXP-V1 and MXP-V2 use an 8 kilobyte scratchpad, with 4 bytes per word (2 kilowords), further divided by 4 for input, output, and temporaries, 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 1024 items, MXP-V8 on 2048, etc. . .).

7.3.1 Sequential Benchmarks

The Sequential Benchmarks represent tasks which 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.

Reverse-3 Reverses an array of word-wide integers using the conventional 3-step approach which assumes all element are local to the same memory: $A \to t, B \to A, t \to B$. MXP cannot implement a 3-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 7.2 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.

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 a large number (100 or more, depending on the platform) of 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.

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 precomputed table lookups which allow each step to calculate the 8th value after the current seed, rather that the immediately consecutive value. Thus, we can sequentially pre-compute the first 8 members of the Hailstone sequence then use these to run 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 out of a total of 224 steps (28×8), producing two extra terminal sequence results $(..., 2, 1)$ 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. See Appendix G for algorithmic details.

FSM-S and FSM-A Executes a Finite State Machine (FSM) which recognizes simple floating-point numbers (e.g.: 0.5, -9., .3, 4.2, etc...) from a stream of characters[1], 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

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

Listing 7.3: Reverse-Recursive MXP Pseudo-Code

```
// Swap individual (even/odd) elements using conditional move
// Make CMOV only write every other element in vector
cond = set_mask_every_other();

// Move odd elements into even element positions
CMOV(temp_vector[0:COUNT-2], cond, data_vector[1:COUNT-1]);

// Move even elements into odd element positions
CMOV(temp_vector[1:COUNT-1], cond, data_vector[0:COUNT-2]);

// <repeat for pairs, quartets, etc...>
```

invalid number which 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 2-3 instructions only and no regular loops. See Appendix H.4 for algorithmic details.

7.3.2 Parallel Benchmarks

The Parallel Benchmarks represent tasks which 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 system scales.

Increment Adds 1 to each of 10 word-wide 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. Contrary to other benchmarks, Increment focuses on the process of looping itself, and thus has a small working set size.

Reverse-4 Reverses an array of word-wide integers using a generalized swap which does not assume that the elements reside in the same memory: $A \rightarrow t_1, B \rightarrow t_2, t_1 \rightarrow B, t_2 \rightarrow A$. 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 which uses conditional moves (`CMOV`) to recursively swap quadratically-increasing large subsets (even/odd singletons, pairs, quartets, etc. . .) of a vector.

Listing 7.3 outlines the core of the algorithm, which completes in approximately $2log_2n$ vector swaps for a vector of length n. We first set a vector mask which 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, 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, etc. . . 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.

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, 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` $\oplus$ `even_branch) & mask)` $\oplus$ `even_branch`) to save a cycle, and MXP uses conditional moves.

Listing 7.4: Pipelined MAC FIR Filter MXP Pseudo-Code

```
for i = 1 to num_samples
   dotp = 0;
   for k = 1 to num_coeff
       dotp = dotp + coeff[k] * in[i+k];
   end for
   out[i] = dotp;
end for
```

Listing 7.5: Transposed FIR Filter MXP Pseudo-Code

```
for i = 1 to num_samples
    out[i] = 0;
end for
for k = 1 to num_coeff
   const = coeff[k];
   for i = 1 to num_samples
       out[i] = out[i] + const * in[i+k];
   end for
end for
```

FIR Computes an 8-tap Finite Impulse Response (FIR) filter over an input array of approximately 100 word-wide integer elements (depending on platform) into a similar and separate output array. For best performance, the HDL implementation uses a separate 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 MXP FIR filter implementation varies, as the final Accumulator (upper right in Figure 7.1) can only sum up to the number of taps, despite scaling to the number of 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 8-tap filter, for 16 lanes and fewer, MXP concurrently performs each of the coefficient multiplications (one per vector lane), then uses the final Accumulator to sum the products into a scalar result, implementing an efficient pipelined Multiply-Accumulate operation (Listing 7.4). For more than 16 lanes, MXP instead successively multiplies

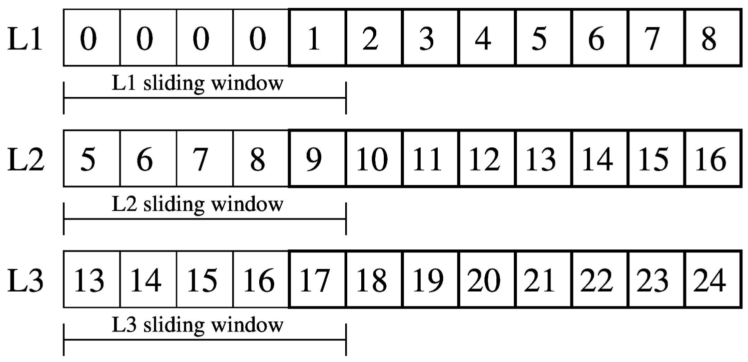


Figure 7.6: Data and extra headers for a 5-tap FIR filter on an Octavo-L3, over 24 input data points.

each coefficient over an entire input vector and sums the products into an accumulation vector (Listing 7.5). 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[2], 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 sequentially to each subset. We accelerate each convolution by providing each SIMD lane with an Accumulator to allow chaining multiplications and sums (Section 7.2). To avoid introducing gaps in the output data, we prepend a header of the last $n-1$ input data 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 7.6 illustrates the headers and data for a hypothetical Octavo-L3 implementing a 5-tap FIR filter over 24 input data points (in the bold boxes).

[2]MXP's Direct Memory Access (DMA) controller would transfer data as needed for continuous, unwindowed filtering.

Table 7.1: 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

7.4 Results

Here we tabulate the raw measurements of clock speed and cycle count, calculate the area of each implementation, and discuss differences in execution and scaling between implementations. Not all implementations are feasible in all cases.

7.4.1 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 NiosII/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 240 MHz) by itself [11].

Reverse-3

Table 7.1 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 2 array elements.

Octavo executes Reverse-3 sequentially and plainly, executing a 6-instruction loop 64 times to reverse a 128-element array. Of the 6 instructions, 3 perform the swap, and the

Table 7.2: 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

other 3 deal with a decrementing a pointer (not supported by the AOM) and the loop counter (not supported by the BTM). The loop branch folds into the last 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 and compare similarly to Octavo: both improve on Area and Area-Delay, but perform less than 3–4× faster.

Hailstone-S

Table 7.2 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 3 folded and cancelling branches to compact the loop counter decrement and both even/odd branches into 5 instructions. MXP's vector lanes cannot compute Hailstone-S since they do not support branches, thus we executed the benchmark on the NiosII/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 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

Table 7.3: 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

single M9K BRAM, reducing its area and maximizing its speed, but still only achieving a 5× speedup over software running on Octavo.

Hailstone-A

Table 7.3 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 8 independent "slices" of the entire Hailstone sequence. MXP requires only 6 instructions, but a few hazards cause them to take 7 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 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 look-up tables per lane. We also included the NiosII/f results which highlight the positive impact of having all data in on-chip scratchpads. Most of the 22.3 cycles/unit are spent doing table look-ups from (cached) main memory.

LegUp appears to perform all 8 slices concurrently, as it requires 8 DSP Blocks, suggesting 8 parallel multiplications, and 6 M9K BRAMs, suggesting concurrent table look-ups for each of the 8 "slices". However, LegUp does not seem to fully pipeline the

Table 7.4: 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 7.5: "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

process, resulting in a relatively low F_{max}, albeit with a 45% net speedup. In contrast, the HDL implementation fully pipelines the 8 sets of calculations, using only 3 M9K BRAMs (2 for tables, 1 for seeds) and re-using Octavo's high-speed Multiplier (2 DSP Blocks). The HDL implementation cannot reach maximum speed since we are limited to 8 pipeline stages (one for each sequence "slice"), and the final adder becomes a critical path. Adding a 9th pipeline stage could conceivably increase F_{max} to 600 MHz (+18%), but at the cost of a pipeline bubble every 9th 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×).

FMS-S and FSM-A

Tables 7.4 and 7.5 show the results for both the "Structured" and "Assembly" versions of the floating-point recognizer Finite State Machine (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 which 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 NiosII/f controlling processor. Since we cannot distribute the work of the FSM over multiple Octavo threads[3], we list the effective F_{max} of a single Octavo-L1 thread: 65.6 MHz. The other 7 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 folding the setup of the next branch in 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 NiosII/f. However, with 7 threads remaining, there remains a lot of margin for additional work "for free" on Octavo.

7.4.2 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

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

Table 7.6: 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

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 as the number of lanes increases (See Chapter 4 as to why). In contrast, MXP's F_{max} first increases due to using fewer and wider BRAMs as the number of vector lanes increases, but then drops as the routing in the ever-widening double-pumped vector scratchpad becomes the critical path.

Array Increment

Table 7.6 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 immediately 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.

Table 7.7: 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

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.

Hailstone-N

Table 7.7 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, 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

Table 7.8: 4-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

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 approximate the performance of HDL hardware, at a considerable area cost.

Reverse-4

Table 7.8 shows the results of the Reverse-4 benchmark. We define the unit of work as swapping 2 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 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

Table 7.9: 8-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

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 4-step swap operation, moving 2 items per cycle instead of 1. Again, Quartus fails to infer BRAMs in the LegUp implementation and instead implements the entire storage as registers, increasing its area and possibly slowing down its operation.

FIR

Table 7.9 shows the results of the 8-tap Finite Impulse Response (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 Multiply-Accumulate (MAC) operations (Listing 7.4), while Octavo-L1 requires 12 as it must also manually re-initialize the post-incrementing pointer which implements 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 7.5) 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 8 DSP blocks to multiply all coefficients in parallel, but unexpectedly required 5 BRAMs, rather than the expected 2 (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 about one quarter the area. In contrast, the HDL implementation uses 2 BRAMs and re-uses Octavo's double-pipeline Multiplier and pipelined Adder (Chapter 3.9) to implement a fully parallel 10-stage pipelined FIR filter which exceeds the 550 MHz BRAM limit, at twice the area cost of LegUp, but over 11x the performance.

7.5 Summary of Octavo's Strengths and Weaknesses

From the results of the benchmarks, and their analysis, we can summarize the strengths and weaknesses of Octavo relative to MXP, HLS, and HDL solutions, and identify opportunities for improvement. Octavo's strengths originate in mutually-supporting features of high operating frequency, fixed multi-threading, and easy extension with hardware accelerators. Octavo's weaknesses stem from incomplete hardware support for loops and pointers, and lack of a conditional move mechanism. We discuss non-benchmark issues and planned improvements in Chapter 8.

7.5.1 High Operating Frequency

Overall, Octavo benefits most from its raw high F_{max}. With all 8 threads busy, fully utilizing the pipeline, a scalar Octavo performs within an order of magnitude of HLS-generated and custom HDL hardware, but at an order of magnitude area penalty. Octavo's strict round-robin multi-threading order enables Octavo's high operating frequency. By constraining thread order, we eliminate the need to forward results across pipeline stages, avoiding costly backwards paths and multiplexers.

7.5.2 Scalable Multi-Threaded Parallelism

Octavo's multi-threaded programming model works efficiently at a small scale, magnifying its benefits at larger scales. Within a scalar pipeline, all threads share the same physical memory in a fixed access order. The same multi-threaded model naturally extends to multiple SIMD pipelines, where each SIMD lane runs duplicates of the 8 original scalar threads, but on a private local memory. Thus, a parallel solution which works on a scalar Octavo naturally extends to SIMD Octavo configurations.

In contrast, MXP's vector approach sometimes suffers from poor initial scaling due to pipeline hazards with short vectors (re: FIR benchmark) or more complex set-up (re: Reverse-4 benchmark), reducing overall throughput and handicapping performance gains from larger vector parallelism.

We can see this 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.

7.5.3 Easy Extension with Accelerators

Generally speaking, we call any hardware attached to Octavo for a specific purpose an "Accelerator". These include communication channels (re: Array Reversal Channels for Reverse-4) or functional units (re: Accumulators for FIR).

Contrary to MXP and other soft-processors [52,60,101,105] which place their custom functional units inside the ALU or directly in the datapath pipeline, Octavo attaches them as simple I/O devices. However, since we memory-map I/O onto scratchpad memory addressed by instruction operands, we keep a tight coupling to the ALU and memory. We can easily chain the output of the ALU to the input of an Accelerator, or vice-versa, by simply addressing the Accelerator's I/O ports as destination or source operands. This tight coupling, but external interface, simplifies the implementation of Accelerators. We can control Accelerators in software with little overhead, and we don't have to alter Octavo's datapath to integrate them.

Furthermore, Octavo's strict round-robin multi-threading ensures that consecutive instructions within a given thread always issue every 8th clock cycle, and the interlock-free pipeline always executes that instruction's reads 8 cycles before its write to memory or I/O. These regularities allow us to design regular, pipelined, and high-frequency Accelerator implementations. Combined with I/O handshaking and instruction predication mechanisms (Chapter 5.2) we can support Accelerators with variable latency, such as DRAM controllers. See Appendix F for a discussion of external memory interfaces.

7.5.4 Incomplete Hardware Support for Loops and Pointers

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 BTM and AOM (Chapter 6). 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. In contrast, MXP's address generators and vector loop counters do not have these limitations. Removing counter and pointer overheads would significantly improve Octavo's performance, but we lacked the time to implement the necessary changes, while these overheads do not obscure the comparisons. For example, Octavo's cycle count for the Reverse-3 benchmark would cut in half, increasing its performance to within a factor of 2 of custom HDL hardware. We discuss these and other related improvements in Chapter 8.6.

7.5.5 Lack of Conditional Move Support

Octavo moderately outperforms MXP at the Hailstone-N benchmark, but only because Octavo's greater clock speed offsets the overhead of simulating MXP's conditional move (`CMOV`) instruction as Boolean operations. On Octavo, selecting one of two results without branching requires computing a word-wide mask (all-zeros or all-ones), then using it 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.

In contrast, MXP can efficiently select one of two results with a single `CMOV` instruction based on an implicit or pre-computed flag, in 1 or 2 cycles total. To maintain Octavo's high efficiency when making code branch-free for SIMD parallelism, we must include some form of conditional move. We discuss possible conditional move implementations in Chapter 8.6.1.

Chapter 8

Summary and Further Work

The answers we have found have only served to raise a whole set of new questions. In some ways we feel that we are as confused as ever, but we think we are confused on a higher level and about more important things. [...] And this is a progress report, rendered with humility because of the unsolved problems we see now which we could not see before.

Earl C. Kelley [63]

8.1 Summary of Contributions

Chapter 3: Self-Loop Characterization We demonstrated the utility of *self-loop characterization*, where we connect a component's outputs to its inputs to take into account the FPGA interconnect, for reasoning about the pipelining requirements of processor components on FPGAs. We used self-loop characterization to determine the pipelining required to achieve the highest possible operating frequency (F_{max}) for the memory and ALU building blocks of a soft-processor. We discovered that minimum clock period restrictions on the RAM (550 MHz) and DSP Blocks (480 MHz) ultimately limit the F_{max} of pipelines on Altera Stratix IV devices. In response, we designed a dual-pipeline Multiplier which acts as a single Multiplier with a 600 MHz F_{max}.

Chapter 3: Octavo Soft-Processor Architecture We discussed the problem of adapting overlay architectures to the underlying FPGA and enumerated the properties of an overlay architecture which maximizes the effective use of the FPGA structures, which led us to a fine-grained multi-threaded soft-processor architecture. Unlike prior multi-threaded processors, we used multi-threading to absorb internal circuit delay rather than external memory latency.

We also restricted ourselves to a strict round-robin thread ordering, where each thread issues an instruction in turn without variation, to simplify programming and composition into larger systems. This decision proved crucial to the rest of our work, enabling many later efficiency improvements, since each thread has the entire length of the pipeline before requiring the result of a computation.

We decided to use a flattened memory hierarchy, without caches and directly addressed by the instruction operands, to more efficiently use the on-chip Block RAMs. We used the available timing slack in Block RAMs, discovered by previous characterization, to memory-map I/O ports over some memory locations for efficient future expansion.

From these building blocks and design decisions, we built *Octavo*, a ten-pipeline-stage eight-threaded soft-processor, which can operate at the Block RAM maximum of 550MHz on a Stratix IV FPGA. Design space exploration showed that the entire family of Octavo designs scaled well over word-width, memory depth, and number of supported threads, presenting a promising foundation for later improvements.

This work was originally published at FPGA 2012, Monterey [77].

Chapter 4: Preserving Multi-Locality with Partitioning We addressed performance problems that emerged when scaling Octavo via *tiling*, which replicated its datapath for SIMD parallelism, or entire Octavo instances for MIMD parallelism. Tiling introduced *multi-localities* (multiple instances of equivalent logic), which the CAD tool optimized down to a single instance, creating artificial critical-path fanouts to physically distant tiles. Multi-local logic might have nothing more in common than simultaneous identical states, without any connection other than a common clock (e.g.: free-running thread number counters). We discovered that we can use logical netlist partitioning to preserve multi-locality and improve performance, with lower design effort than CAD tool options or HDL source annotations, and negligible CAD processing time cost.

Preserving multi-localities improved operating frequency and compute density (i.e.: work per unit area), with improvements scaling with the amount of tiling. For example, partitioning a mesh of 102 scalar Octavo cores improved its operating frequency 1.54x, from 284 MHz up to 437 MHz, but increased its area by only 0.85%.

This work was originally published at ICFPT 2013, Kyoto [78].

Chapter 5: Extended Write Address Space While Octavo exhibited high performance and useful scaling, we still had to address some internal inefficiencies which limited the size of programs and the amount of future hardware expansion.

We used the two remaining spare bits in Octavo's instruction format to expand the Destination operand (D) from 10 to 12 bits. We then consecutively mapped the 10-bit Instruction, A Data, and B Data read address spaces into this expanded write address space, rather than overlapping as before. By eliminating overlapping of code and data address spaces, we doubled the available data memory and removed all non-code data from instruction memory.

The extended write address space also created a 10-bit, write-only, "High" memory range to memory-map later hardware additions without consuming precious I/O ports, which we reserved for data-intensive operations rather than infrequent hardware control.

Chapter 5: I/O Predication Octavo's minimal I/O ports could only blindly send or receive single data words, forcing us to use software busy-wait loops for external interfaces with unpredictable latency, such as DRAM controllers.

An Octavo instruction may address up to 3 I/O ports (1 write and 2 reads) as regular operands. We added an Empty/Full bit to each I/O port to enable basic request/acknowledge handshaking, and test the readiness of all addressed I/O ports before allowing an instruction to continue. If any of the I/O ports are not ready, we annul the instruction (i.e.: replace it with a `NOP`) and re-issue it the next time the thread comes around. These *predicated* instructions eliminated I/O busy-wait loops and provided a basis for powerful conditional branches in Chapter 6.

Chapter 6: Reducing Addressing and Branching Overhead Regardless of apparent performance, Octavo, as a CPU, still exhibited intrinsic addressing and flow-control overheads separate from the actual desired computations. The sequential nature of programs interleaves addressing, branching, and useful computations, leading to a performance gap relative to custom FPGA implementations doing the same work, regardless of relative clock speeds.

We extracted concurrent addressing and flow-control sub-graphs out of the Control-Data Flow Graph (CDFG) of programs, and found that they contained relatively little information, and spent most of their time waiting for certain conditions to occur. We reduced the addressing and flow-control sub-graphs to table entries in special-purpose modules, indexed by the Program Counter and other CPU state, and checked in parallel with the instruction fetch.

The Address Offset Modules (AOMs) execute the addressing sub-graph and provide:

- shared code across threads by adding a per-thread default address offset to the instruction operands,
- indirect addressing by adding a programmed offset to instruction operands accessing pre-defined memory locations,
- post-incrementing addressing by adding, after each access to an indirect memory location, a programmable increment to the programmed offset.

The Branch Trigger Modules (BTMs) execute the flow-control sub-graph and enable:

- folded branches, which execute in parallel with ALU instructions,

- cancelling branches, based on the instruction annulling mechanism from Chapter 5, which cancel the parallel ALU instruction based on a Taken/Not-Taken static branch prediction, allowing us to always place a useful ALU instruction in parallel with a branch,
- multi-way branches, which fold multiple branches, triggered by mutually exclusive conditions, with a single ALU instruction.

As a side-effect of implementing branches as BTM entries, we also eliminated the need for branch instructions, freeing 5 opcodes for other use. We can also create branches on application-specific conditions, such as "Branch on Even", without altering the ISA.

A program can hoist addressing and branching work out of loops and into the AOMs/BTMs. We compared our optimized micro-benchmark code against an ideal "perfect" MIPS-like CPU, which has no stalls or delay slots. Against this ideal CPU, using the AOMs/BTMs achieved speedups ranging from 1.07x for control-heavy code, to 1.92x for looping code, never performed worse than the original sequential code, and always performed better than a totally unrolled loop. The AOMs/BTMs reduced raw clock speed by only 6.5%.

This work was originally published at ICFPT 2014, Shanghai [74].

8.2 Programming Support

Currently, we only have assembly language to program Octavo systems. Octavo's simple instruction set and strict round-robin multi-threading model simplify assembly programming greatly, but we still lack the productivity of even the simplest programming systems. Determining the best programming system (compiler, language, libraries, etc. . .) for Octavo lies outside the scope of this thesis, but we can point out pitfalls to avoid, challenges to solve, and desirable properties to favour.

8.2.1 Avoiding a Brittle Software Stack

Normally, a processor's instruction set and architecture remain fixed, or change incrementally over several years, and the programming system adapts to application needs[1] (e.g.: OpenCL, built atop C). Octavo turns that arrangement around by adapting the processor to the application via custom hardware Accelerators, new instructions, and SIMD/MIMD parallelism. To maintain productivity, the programming system must remain relatively constant above all that change.

Hence the main pitfall: the productivity gained from programming a tailored overlay could get eaten up by the increased maintenance to the programming system. Worse yet, if the programming system cannot consistently generate optimized code, that uncertainty undoes the hardware design effort spent improving Octavo's performance. Thus, much like Octavo's relationship to the underlying FPGA, the programming system should aim for initial simplicity, avoid too much up-front specification of how to express computation, and instead provide primitives for reliable and efficient incremental construction.

8.2.2 Compiling to a Reconfigurable Architecture

Every Octavo instance may contain application-specific features (e.g.: Jump on Even (Chapter 6) for Hailstone calculations (Chapter 7)). A compiler will have to deal with several challenges, including:

Accelerators These may range from simple channels with known latencies (e.g.: Array Reversal Channels from Chapter 7.2), stateful functional units (e.g.: Accumulators, *ibid.*), to entire remote processors. We should define compiler intrinsic functions or provide inlinable library functions to describe the computations done by the Accelerators and allow the compiler to infer dependencies and re-order instructions.

[1] Tensilica's Xtensa [52] processor architecture is one notable exception.

Reconfigurable Instructions We will need to infer the lifetimes of AOM and BTM entries (Chapter 6), manage them via a kind of register allocation, and spill/fill them as required. For example, when the number of live branches exceeds the number of BTM entries, we must re-use a less-frequently-triggered entry to prepare for the next upcoming branch. A similar case exists for proposed reconfigurable Boolean instructions, (Sections 8.6 and 8.7.3). Since we could alter the meaning of these instructions, we would have to use compiler intrinsics (i.e.: functions built-in to the compiler) to configure them, let the compiler infer the lifetimes of each configuration, and load them as required.

Heterogeneous SIMD We introduced a SIMD version of Octavo in Chapter 4, but did not address an optimization where the additional SIMD lanes have a narrower word width than the original datapath to save area and increase performance. The original datapath must remain at full word width to support instructions with the full address range. The compiler will have to generate code which computes the same results on the full word width of the original datapath and the narrower word width of the SIMD lanes. Additionally, we also did not address temporarily disabling SIMD lanes to allow scalar computations on the same processor. We could disable SIMD lanes by disabling the read and write enables of their local A/B memories, causing the SIMD pipelines to eventually empty themselves and remain idle.

8.2.3 Literal Pool Support

Octavo's ISA (Chapter 3.7) does not include any immediate literals. We needed as many operand bits as possible as scratchpad address space, and the resulting direct memory addressing allowed direct use of literals in memory. Without indirect addressing, all threads could use the same absolute address to refer to a shared literal value.

However, introducing the AOM (Chapter 6) adds per-thread indirection to memory addresses, which enables sharing code across threads, but now forces each thread to maintain a private pool of literal values. With multi-threading parallelism, all the pools contain identical values, wasting A/B memory.

The AOM does support a limited amount of direct addressing, in the form of Shared Memory address ranges for I/O ports and High Memory (see the `SM?` module in Figure 6.2, pg. 109). We can extend Shared Memory support to include a small region of A/B memory to contain a single common literal pool for all threads, without preventing per-thread private pools. We can place the literal pool at the beginning of A/B memory, as address zero has already been reserved as a "zero register" holding a literal `0`.

8.3 Debugging Support

Debugging software on a multi-core FPGA overlay poses some new challenges. On a CPU or GPU, all work essentially proceeds to/from a single, inspectable, global memory, or at least a small number of large memories from which we can easily copy to/from. In contrast, an FPGA overlay keeps its computations local, with each core holding intermediate values, and passing around only the required data. In effect, an overlay running an algorithm presents a black box very similar to the equivalent hardware implementation.

One possible solution to enable debugging the internal state of the Octavo overlay makes use of the ability to rewrite Instruction memory. We can implement a simple interpreter which can execute instructions from an external memory, read/write from local memory, and alter the debugged thread's control flow. We can also use the BTM to act as a breakpoint system by jumping to the interpreter as required, without having to alter the running thread. Finally, if all I/O ports make use of handshaking, stopping a thread will eventually non-destructively hang all other local threads, Accelerators, and remote threads the debugged thread communicates with, allowing us to resume it afterwards without ill effect (except maybe failing at some real-time tasks).

This approach has likely been reinvented many times over computing history, but the version explained here derives from Bill Henning's Large Memory Model for the Parallax Propeller microcontroller [54]. The core of the interpreter requires only 4 instructions:

Listing 8.1: Debugging Interpreter Core Routine

```
interpreter: ADD first,  I/Or, 0 // read from external mem.
             ADD second, I/Or, 0 // unrolled to avoid RAW
first:       <empty>             // 'first' in Instr. mem.
second:      <empty>             ; JMP interpreter
```

This loop, unrolled once to avoid a 1-cycle RAW hazard when writing to Instruction memory, reads instructions from an external memory connected to an I/O read port (`I/Or`), stores them ahead in Instruction memory, then executes them. We assume the read enable line of `I/Or` triggers a post-increment of an address register indexing the external memory, or other similar device.

This interpreter uses two `ADD` instructions to load and execute two external instructions at effectively 50% the rate at which they would execute from local memory, but without the local memory's limited space. We can implement flow control by executing short instruction sequences which alter the aforementioned implicit addressing of the external memory.

From this interpreter core routine, we can then invoke other small utility functions to read/write blocks of data and instructions, set a spare BTM entry as a breakpoint back into the debugger, or simply temporarily pipe the stream of external instructions to another I/O port connected to another processor in the overlay also running the same interpreter core routine.

With some care, the initial invocation of the core interpreter can originate from a folded multi-way branch triggered by a particular input signal or state transition, providing a resident remote debugger with zero execution overhead in normal circumstances.

8.4 Overlay Portability via Resource Diversity

We originally tuned Octavo's design to the Altera Stratix IV family of FPGA devices (Chapter 3). To properly port Octavo's design to another FPGA device, we would have to repeat the process of self-loop characterization on each sub-module to adjust their pipeline depths, as well as redesign some sub-modules entirely. For example, the DSP Blocks on Xilinx devices resemble more full ALUs [26, 27] than Altera's Multiply-Accumulate DSP Blocks, which would completely alter the design of the Multiplier and of the entire ALU (Section 3.9).

The number and ratio of various FPGA structures also affect portability. For example, the Cyclone IV FPGAs have fewer and smaller DSP blocks, possibly limiting the number of Octavo cores or datapaths, even if enough memory blocks exist to support more. We already provide multiplier implementation options: single or dual pipeline, each using DSP blocks or reconfigurable logic, all trading off resource usage and operating speed (Chapter 3.9). Some other multiplier resource diversity options include:

Conditional Add-and-Shift Replaces a full multiplier with an instruction which computes 1 bit of product per cycle, accelerating software multiplication at little hardware cost. This option would benefit applications with moderate multiplication needs.

Memory-Mapped Multiplier We could remove the multiplication instructions and instead attach a multiplier (of any implementation) to I/O ports, freeing opcodes and optionally allowing multiple processors to share infrequently used multipliers, at the price of one cycle of result latency.

Exploring resource diversity matters since a given application might benefit from more numerous Octavo cores, each using fewer instances of a constrained hard block, with a consequently lower core performance, but nonetheless higher total system performance. For example, switching to single-pipeline DSP-based multipliers limits Octavo to 480

Table 8.1: Octavo F_{max} on Various Altera Devices

Family	Device	Average (MHz)	Maximum (MHz)	Avg/Max (Ratio)	Limit (MHz)	Max/Lim (Ratio)
Stratix V	`5SGXEA7N2F45C1`	508	588	0.864	675	0.871
Stratix IV	`EP4S100G5H40I1`	470	493	0.953	550	0.896
Arria V	`5AGXFB5K4F40I3`	272	300	0.907	400	0.750
Cyclone V	`5CGXFC7D6F31C6`	239	267	0.895	315	0.848
Cyclone IV	`EP4CGX30CF19C6`	187	197	0.949	315	0.625

MHz on Stratix IV, but halves the number of DSP blocks required, possibly allowing for more cores with a greater net total performance.

Alternately, since the overall F_{max} decreases when we scale-up an Octavo system (Chapter 4), we might then deliberately use smaller, slower implementations of sub-modules not on the critical path to save area and power at little to no cost in performance. For example, the 32-lane SIMD Octavo in Chapter 7 reaches 370 MHz even with dual-pipeline DSP Block-based multipliers, which can reach 600 MHz themselves. If we assume that the overall F_{max} stems more from general FPGA overhead, then perhaps reducing the Multiplier to a single-pipeline DSP Block implementation, itself limited to 480 MHz, might save area without reducing performance, or might even improve performance slightly from reduced placement and routing difficulty.

Finally, we wrote the AOM/BTM implementations with a very sparse memory implementation: each individual entry (PO, DO, BO, etc...) is implemented as a separate MLAB RAM module for easy design space exploration, and it is unknown if the CAD tools can automatically pack multiple separate but logically contiguous memories into a single MLAB.

8.4.1 Porting Across Altera FPGA Device Families

Despite the portability concerns we mention above, since Altera's FPGA devices tend to have similar architectures across device families, we can expect good portability as-is, without re-design. Table 8.1 shows some measurements of Octavo's Average, Maximum, and absolute upper Limit on performance, measured in MHz, on a variety of Altera FPGA families. We use some of the same demonstrator devices used to publish Nios II performance data [11], and with the same Quartus settings and experimental methodology used throughout this work (Appendices A and B).

Absolute F_{max} Results

In all cases, memory blocks (MLABs or BRAMs) set the upper limit on F_{max}. We can pair DSP Blocks to attain higher speeds (Section 3.9), so they never cause a bottleneck.

For Stratix V, the MLAB memories, used to implement memories in the Controller (Chapters 3 and 5), and the AOM/BTM (Chapter 6, set an upper limit of 675 MHz [6], but only when used with depths of 16 words. Greater depths (32 or 64 words) have a limit of 450 MHz. Several of the Quartus runs resulted in identical F_{max} values, which strongly suggests an artificial limit caused by sub-optimal settings.

For Stratix IV, Arria V, and Cyclone V, the M9K (or M10K for "V" series devices) BRAMs set a limit of 550 MHz [10], 400 MHz [7], and 315 MHz [9] respectively.

For Cyclone IV, the M9K BRAMs also set a limit of 315 MHz [8], same as Cyclone V. However, the Cyclone IV devices do not contain MLABs, so all memories instantiate as M9K BRAMs, bringing their count up from 12 to 26, which will lower F_{max}. Also, contrary to all the other devices here, the Cyclone IV devices use 4-LUTs instead of 6-LUTs, or similarly larger fracturable LUTs, in their ALMs. We would have to either re-design Octavo, and/or accept smaller instances (e.g.: with less memory), to address these limitations and improve performance.

Relative F_{max} Results

We also compute two ratios from the absolute F_{max} values: $Maximum/Limit$ and $Average/Maximum$, which together allow us to see if Octavo carries over well from one device family to another.

Maximum/Limit Ideally, we want the Maximum (unrestricted) F_{max} to equal (or exceed) the Limit of the FPGA device. Although a maximal F_{max} by itself does not guarantee maximal performance (as shown in Chapter 6), a low one certainly prevents us from getting all the potential work out of the underlying FPGA (see the Design Goals in Section 3.1). For example, compare the Max/Lim ratios of the Cyclone IV and Cyclone V devices, which both have identical F_{max} Limits. As-is, Octavo does not make good use of the Cyclone IV FPGA hardware and needs re-characterization and some different design choices (e.g.: narrower word width) to improve raw performance.

Average/Maximum A larger variation between the Average and Maximum F_{max} may indicate a greater sensitivity to individual Place-and-Route (P&R) results. This sensitivity suggests either a revision of the self-loop characterization of the module(s) within the critical path to alter the number and/or the location of their pipeline stages, or an adjustment of the CAD tool settings. For example, despite Stratix V's higher performance, Octavo's F_{max} varies more over multiple P&R solutions. The observed repeated identical F_{max} results on Stratix V suggest adjusting the CAD tool settings.

However, a smaller variation may also simply indicate a dominant critical path which manifests in most P&R solutions. For example, the Cyclone IV's lower Max/Lim ratio strongly suggests that its lower variation between Average and Maximum F_{max} exists due Octavo not matching the underlying FPGA structures well.

8.5 Automating Partitioning of Tiled Overlays

Although manual partitioning of tiled overlays requires relatively little work from the designer (Chapter 4), we believe the CAD tool might automatically detect multi-local logic and take some action other than complete redundancy elimination:

- The CAD tool could detect repeated deduplication of the same logic across multiple modules and declare that logic multi-local, optionally alerting the designer.

- The CAD tool could "restart" the optimization of multi-local logic after a certain number of deduplications, ensuring that sufficient copies remain to avoid large critical paths, while still reducing area.

- The CAD tool could use the location of the multi-local logic to automatically partition its enclosing modules, automating our manual approach. Some prior work by Dehkordi, Brown, and Borer [37] has explored this kind of automated modular partitioning, but only to control incremental recompilation time, with no impact on frequency.

Power Trade-off Since partitioning can increase the operating frequency more than it does logic area, the power density and power consumption of the system may also increase. Alternately, the power consumption may also decrease due to driving fewer device-wide critical paths when partitioned. For designs under power constraints, the CAD tool could explore the dynamic and static power trade-offs of full or partial partitioning.

Area Overhead Finally, although the CAD tool preserves the performance of partitioned tiles very well as their number increases, it seems to add an unexplained and consistent area overhead when tiling, causing a drop in compute density. Future CAD tools should limit this overhead to better enable large-scale tiled overlays.

8.6 Reducing Computation Overhead

Although the BTM and AOM (Chapter 6) can fold a lot of "overhead" addressing and branching computations into instructions performing useful work, the current implementations have only the minimal set of features possible as a proof of concept. Some possible improvements include:

More Branch Conditions We can add more conditions for the BTM to branch against:

- a comparison of the result of the previous thread instruction against a loadable sentinel value (e.g.: zero to detect string termination)
- a counter, stepping either each clock/thread cycle as a cycle counter, or each time we test a BTM entry as a loop counter
- some custom flag calculation logic (e.g.: accumulator overflow)

These branch conditions, especially loop counters, would further accelerate loops and tests (see Chapter 7.5.4).

Subroutines and Interrupts We can extend the BTM to support folded subroutine CALL and RETURN operations, signalling the Controller to push/pop a PC stack. The BTM could also push/pop an internal stack pointer, providing the low-order bits to the BTM's memories, to automatically switch to the subroutines' pre-set BTM entries. These changes would eliminate the need to inline code for performance reasons, as well as make code more compact in the limited Instruction memory. Subroutine support also sets the stage for adding interrupts as hardware-invoked subroutines.

Greater Address Offsets We need to extend the range of address offsets in the AOM to support larger strides when operating on data divided across threads, and negative strides for backward loops. To keep the positive and negative ranges equal, a signed magnitude representation would work best.

Generalized Sliding Window The AOM could implement sliding window and modulo addressing access patterns (such as seen in the FIR benchmark) by incrementing by an offset A for the first $N-1$ accesses, followed by a one time increment by an offset B (likely negative) on the last N^{th} access. DSP processors commonly implement modulo addressing, to implement circular buffers, by masking the high-order bits of an address.

Instead, a generalized sliding window approach to modulo addressing would not require aligning a buffer to a power-of-2 memory location.

Scatter/Gather Memory Access Octavo's single AOM performs memory indirection for only the threads on the initial datapath. All SIMD datpaths will see the same final indirected addresses, preventing these SIMD lanes from performing independent table lookups and pointer-chasing. This uniformity poses no problems for the regular memory access patterns of the Parallel Benchmarks shown here (e.g: FIR). However, some types of parallel computations require scatter/gather memory accesses (e.g.: sparse arrays, unstructured grids [15]), and could parallelize to more than the 8 threads of a single Octavo datapath. Adding an AOM to each SIMD lane would enable each lane to work on independent, but irregular, parallel (sub-)problems. For example, we could run multiple Hailstone-A benchmarks on SIMD lanes, parallelized in the same manner as Hailstone-N, but taking 7 cycles/result rather than 11, for a 1.57x speedup.

8.6.1 Conditional Move

Adapting non-trivial code to SIMD parallelism requires converting branches to non-branching code ("if-conversion"), where we compute both alternatives then select one at the end with a conditional move (`CMOV`) instruction. While Octavo can emulate `CMOV` using Boolean operations (e.g.: the Hailstone-N benchmark in Chapters 7.3.2, 7.4.2, and 7.5.5), this approach takes 5 instructions rather than the 1 or 2 of a conventional `CMOV`. This overhead remains proportionally constant with SIMD scaling, thus Octavo requires a faster `CMOV` implementation to maintain efficiency. A few design alternatives present themselves (see Figure 7.2, pg. 126, for reference), with different trade-offs:

Single-Source `CMOV` Instruction A common form of conditional move tests the first source operand (`test`) for a given condition (`cond`), and if met, moves the second source operand (`src`) to the destination operand (`dest`): `CMOV_cond dest,test,src`. This form has the advantage that we can freely (re-)use `test` at any time. On Octavo, the single-source form would mean calculating `cond` from `test` inside the ALU, passing `src` through, and setting a write-enable signal to `dest`.

However, calculating `cond` puts yet another functional unit in the ALU, and the additional result multiplexing might become a critical path. Also, we cannot add opcodes for all possible `CMOV` conditions (see Table 3.2), so we must either implement a subset and sometimes pay a little overhead to compute the desired `test`, or reconfigure `cond` in the functional unit via a write to High memory (Chapter 5) each time we change condition.

Dual-Source `CMOV` Instruction Another form of conditional move tests an implicit condition (`cond`) (e.g.: from the previous instruction) and uses it to selectively move one of the two source operands (`src1` and `src2`) to the destination operand (`dest`), acting like a multiplexer: `CMOV_cond dest,src1,src2`. This form has the disadvantage that we must compute the implicit condition immediately before `CMOV`. On the other hand, the dual-source form does not require potentially writing to `dest` twice, which can avoid side-effects (e.g.: if `dest` is an I/O port).

On Octavo, the dual-source form matches the form of branch operations (Chapter 6), which use the result of the previous thread instruction to determine one of several flags, and thus spreads the work over the entire pipeline. We would pipeline the selected flag bit to the ALU, which then simply passes one of `src1` or `src2` to `dest`. This approach matches Octavo's multi-threaded pipeline, and might even re-use some of the "fast compare" [62, 87] flag generation and selection hardware developed for the Branch Trigger Modules (BTMs) in Chapter 6. However, we still cannot assign opcodes for all possible conditions, leading to the same compromises as the single-source form.

Improved Boolean `CMOV` Emulation Alternatively, we could continue to emulate `CMOV` using Boolean operations, but apply the suggested bit-level parallelism improvements from Section 8.7.3 to improve their efficiency. For example, given a mask bit (selected by hardware or software) and some ALU support for reconfigurable 3-term Boolean operations, we could emulate `CMOV` in a single cycle instead of 3: `((src1` $\oplus$ `src2) & mask)` $\oplus$ `src2`. This approach depends on the feasibility of the ALU modifications, but would not permanently consume opcodes.

8.7 Approaching the FPGA's Native Performance

Despite Octavo's performance improvements, a CPU-based approach to FPGA overlay architecture still forces us to think in terms of conventional CPU processing, which ignores the fundamental properties of FPGAs which benefit custom hardware implementations. These FPGA properties include: a high internal memory bandwidth, bit-level parallelism, more numerous and better-utilized functional units, all of which also contribute to reduced power. Thus, some changes to Octavo's architecture and instruction set motivate themselves to better utilize the underlying FPGA hardware.

8.7.1 Increasing Internal Memory Bandwidth

Octavo performs its scratchpad memory accesses before the ALU, opposite to typical MIPS-like pipelines, which perform a memory load or store after computing the address in the ALU. By memory-mapping the I/O ports into the scratchpad, rather than using load/store instructions, placing the memory before the ALU grants Octavo significantly more external memory bandwidth. A scalar MIPS-like pipeline can, at best, write or read one value to/from the outside world every other cycle, before or after an internal computation instruction which generates or consumes the data. In contrast, in Octavo's best case, any one instruction may read up to two external values from I/O ports and write back the result to a third.

And yet, this approach still leaves potential memory bandwidth unused. When moving data, we must use the usual "`ADD to 0`" idiom, wasting half of the memory read bandwidth [64]. Furthermore, the A and B memories have independent write ports, yet we only ever use one at a time when writing the result of an instruction. The memory-mapped I/O ports also remain underused for these reasons, presenting a potentially serious bottleneck between Octavo cores within a larger overlay. In theory, Octavo could perform two reads and two writes to memory or I/O ports every clock cycle.

For an instruction to perform two writes at once, it would need two Destination (D) fields (Chapter 3.7). We can create an instruction which splits the 12-bit D field into two 6-bit D fields, one each for the A and B memories. This "double-move" instruction can then read from A and B as usual, pass the values unaltered through the ALU, possibly swapping them, and then write one of each back into 64-word windows in A and B. We could make the 64-word windows movable by creating some additional hardware, similar to the AOM, to add an offset to their memory-mapped locations, if the impact on F_{max} is not too great. Lastly, the AOM and BTM would have to be altered to handle the dual D fields.

A double-move instruction would improve Octavo's operation in several ways:

Better Accelerator Utilization We can move two values at once to an Accelerator attached to the I/O ports, or feed two Accelerators at once. Similarly, we can move two values at once between the A and B memories to accelerate swaps and other data transformations by movement.

Greater Bisection Bandwidth We can move twice as much data per cycle over I/O ports, enough for one Octavo core to supply both source arguments for an instruction on another Octavo core, or move data to/from two other Octavo cores in a point-to-point mesh overlay architecture, potentially doubling the bisection bandwidth.

Reduced Overhead A double-move amplifies the overhead reduction of the AOM and BTM: we can now post-increment up to 4 indirect memory accesses at once (2 reads and 2 writes), and further shorten the body of loops by combining data movements.

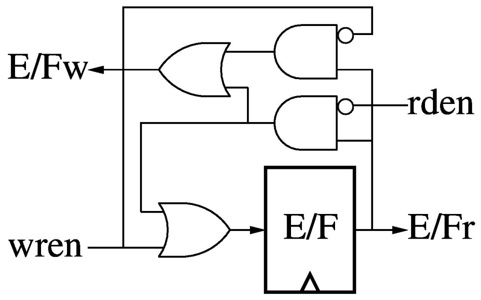


Figure 8.1: The Empty/Full bit, with support for back-to-back transfer.

8.7.2 Improving I/O Handshaking Throughput

If we improve Octavo's memory bandwidth, we must also ensure that the I/O ports do not become the new bottleneck. When connecting read and write I/O ports together to transfer data, a direct link suffices to synchronize both ends: feed the write or read enable signal to the Empty/Full port on the other side, and the first of the reader or writer to access the link will hang until the other arrives to complete the transfer (Chapter 5.2).

However, a direct synchronized link prevents overlapping communication and computation, suggesting a FIFO buffer between sender and receiver. Unfortunately, by itself, buffering halves the peak transfer rate by preventing back-to-back transfers: after writing into the first FIFO stage, the sender cannot immediately write again on the next cycle, as the FIFO's first Empty/Full bit will still read as "Full" at the beginning of the clock cycle, even though the associated data register could send off the previous data as it receives the new ones.

In the spirit of Octavo's focus on small-scale, fine-grained efficiency, we need to enable simultaneous reads and writes through an Empty/Full bit so as to both allow overlapped communication and computation while not penalizing already synchronized transfers by making them stall every other cycle.

Figure 8.1, Table 8.2, and Equations 8.1, 8.2, and 8.3 show the schematic, the truth table, and the Boolean equations of an enhanced Empty/Full bit. Normally, asserting write enable (wren) sets the Empty/Full bit (E/F) to "Full" (1) if "Empty" (0), while

Table 8.2: Truth table for Empty/Full bit with back-to-back transfers.

rden	wren	E/F	next E/F	E/Fr	E/Fw
0	0	0	0	0	0
0	0	1	1	1	1
0	1	0	1	0	0
0	1	1	1	1	1
1	0	0	0	0	0
1	0	1	0	1	1
1	1	0	1	0	0
1	1	1	1	1	**0**

$$next\,E/F = wren + (\overline{rden} \cdot E/F) \tag{8.1}$$

$$E/Fr = E/F \tag{8.2}$$

$$E/Fw = (\overline{rden} + \overline{wren}) \cdot E/F \tag{8.3}$$

asserting read enable (rden) does the opposite. The read and write E/F ports each receive a copy of the current E/F bit (E/Fr and E/Fw).

The only deviation from this behaviour, described by the last line of the truth table, occurs in the case of a simultaneous read and write on a "Full" E/F bit: the read port will see a "Full" E/Fr, but the write port will see an "Empty" E/Fw, allowing both to proceed. The E/F bit remains unchanged, while the associated data register gets loaded with new data. Thus the sender and receiver can synchronize as usual, and still transfer without interruption if already synchronized.

8.7.3 Improving Bit-Level Parallelism

NVIDIA's Maxwell GPU architecture introduced a `LOP3` instruction [96] which implements a word-wide 3-LUT using 3 source operands and an 8-bit literal representing a Boolean truth table[2]. The instruction then computes the bit-wise result of all the source bits, based off the truth table, collapsing multiple Boolean operations on three inputs down to a single instruction:

[2]The `VPTERNLOG` instruction will do the same in upcoming Intel Xeon processors with Advanced Vector Extensions (AVX-512) support.

```
LOP3.LUT dest, src1, src2, src3, TruthTableByte
```

Even with Octavo's two source operands, we could construct similar LUT instructions to collapse any of the 16 possible 2-input Boolean functions [34] into a single step.

Properly configured, such instructions would reduce any sequence of Boolean operations on N inputs to $N-1$ steps by effectively eliminating all explicit negations which could not be optimized away via DeMorgan's Law transformations at compile-time. For example, we could perform $\neg A \wedge B$ in one instruction instead of two, or $(\neg A \wedge B) \vee C$ in two instructions instead of three, etc... We could extend the LUT instructions to 3 inputs if we also include the result of the previous thread instruction, which may itself be a 3-input LUT instruction, rapidly collapsing arbitrary sequences of Boolean operations into few instructions.

We could implement LUT instructions by loading small memories in the ALU, mapped into High Memory, with the 4 or 8-bit truth tables of each function. Several truth tables easily fit in one memory word, with a few bits left-over for extra functionality such as input selection. Some possible extra inputs could include word-wide masks constructed from replicated single bits (e.g.: sign bit, least-significant bit, all-zero condition) allowing fast bit-masks, conditional moves, etc...

Furthermore, by placing (or chaining, in the case of 3-input versions) these configurable Boolean instructions in series with the adder/subtractor in the ALU (Figure 3.5, pg. 40), we can compose many sophisticated branch-free parallel bit manipulation and mathematical operations built up from atoms of the form $(A+B) \odot C$, where $\odot$ signifies a Boolean operation [123]. Finally, new kinds of bit-manipulation functional units (as attached accelerators) could also improve many applications in the domains of compression, cryptology, image processing, bioinformatics, and communications [57].

8.7.4 Keeping Functional Units Fully-Utilized

Even if all Octavo threads work fully and use the ALU every single instruction, the individual functional units within the ALU cannot reach full utilization, possibly limiting the maximum computational density (work per unit area) of the overlay.

Each functional unit is fully pipelined, but each thread can only have one ALU operation in flight at a time, thus some functional units in the ALU must remain either totally idle, or have some idle stages at any time. For example, three consecutive threads each issuing a multiply instruction will fill the multiplier's pipeline, but leave the adder/subtractor and logic units completely idle for the duration (Figure 3.5, pg. 40). Issuing any one ALU operation guarantees an idle stage in all other functional units.

Octavo's strict round-robin multi-threading works against solutions using multi-cycle functional units. All but one of multiple threads trying to use a multi-cycle unit will find themselves stalled while the unit computes, creating a bottleneck which we cannot avoid by re-scheduling threads.

A simpler solution than superscalar issue or complex instructions would be to redesign the ALU to combine functional units together (e.g.: by using the adder/subtractor inside the DSP blocks for ADD instructions) to increase their utilization.

8.7.5 Reducing Power

Currently, Octavo's entire pipeline runs without pause, even if executing `NOP`s, reading from all memories each cycle and latching new values at each pipeline stage even if they will never be used, all wasting power. When scaled up via SIMD/MIMD tiling (Chapter 4), Octavo's high F_{max} makes it likely that the overlay will approach or exceed the FPGAs thermal design limits. Furthermore, the overlay will definitely consume more power than the equivalent custom hardware (see area benchmarks in Chapter 7), limiting its use in embedded and high-performance computing, both sensitive to power.

We might reduce power usage by adding a little decoding logic to disable the write to registers at various stages of the CPU pipeline when unused, and/or explicitly disable the read/write ports of memories during `NOP`s, or in unused SIMD lanes. Also, the I/O write ports currently update all data registers in parallel, using the write enable signal to signal the intended port. Using the write enable signal to mask the data register writes would eliminate this wasteful switching activity.

8.8 Emulating Reconfigurable Hardware

Listing 8.2: Emulated Reconfigurable Hardware Loops

```
loop1: XOR I/Ow, I/Or, key1 ; JMP loop1
loop2: XOR I/Ow, I/Or, key2 ; JMP loop2
loop3: XOR I/Ow, I/Or, key3 ; JMP loop3
...
```

Listing 8.2 outlines an idea for an Octavo thread to temporarily behave as a single fragment of hardware, using a (very!) contrived encryption system as an example. Each line describes an infinite loop composed of a single instruction and a BTM entry. In this case, each loop reads in a value from an I/O read port (`I/Or`), `XOR`s it with a unique `key`, then outputs the cyphertext to an I/O write port (`I/Ow`), and instantly loops back to itself via `JMP`. We can increase the effective "clock speed" of this logic function by having multiple threads run the same loop.

Although useless as-is, if another thread or external event could cause the `JMP` to fall through to the next loop, then we can sequence this thread through a series of basic functions. Chaining multiple such threads in one or more Octavo cores could thus construct a reconfigurable functional unit, not unlike conventional FPGA dynamic partial reconfiguration.

Appendices

Appendix A

Experimental Framework

We evaluate Octavo and its components on Altera Stratix IV FPGAs, although we expect proportionate results on other FPGA devices given suitable tuning of the pipeline.

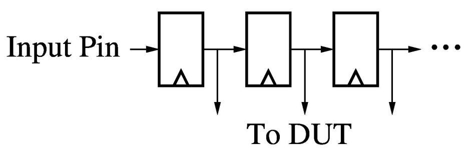
Test Harness We place our circuits inside a synthesis test harness designed to both: (i) register all inputs and outputs to ensure an accurate timing analysis, and (ii) to reduce the number of I/O pins to a minimum as larger circuits will not otherwise fit on the FPGA. The test harness also avoids any loss of circuitry caused by Boolean optimization. Shift registers expand single-pin inputs (Figure A.1(a)), while registered AND-reducers compact word-wide signals to a single output pin (Figure A.1(b)).

Synthesis We use Altera's Quartus[1] to target a Stratix IV `EP4SE230F29C2`[2] FPGA device of the highest available speed grade. For maximum portability, we implement the design in generic Verilog-2001, with some LPM[3] components. We configure the synthesis process to favour speed over area and enable all relevant optimizations. To confirm the intrinsic performance of a circuit without interference from optimizations—such as register retiming, which can blur the distinction between the circuit under test and the

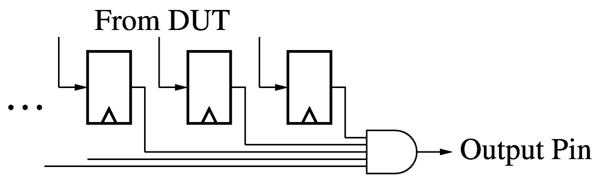
[1]Version 10.1 for Chapter 3, 12.1 for Chapter 4, and 13.1 for Chapters 6 and 7.

[2]Except for Chapter 7, where we target the `EP4SGX230KF40C2` device on a Terasic DE4-230 board.

[3]*Library of Parametrized Modules* (LPM) are used to describe hardware that is too complex to infer automatically from behavioural code.



(a) Test Harness Shift Register



(b) Test Harness Registered AND-reducer

Figure A.1: Test Harness for Device Under Test (DUT)

test harness—we constrain a circuit to its own logical design partition, excluding the test harness. Any test harness circuitry remains logically separate from the actual circuit under test.

In Chapter 3, we needed to avoid getting optimistic results from spatial proximity of the test harness to a small design. Thus, we also restricted placement to within a single rectangular floorplan (*LogicLock* area) containing only the circuit under test, excluding the test harness. Any test harness circuitry remained spatially and logically separate from the actual circuit under test. Quartus automatically determined the floorplan.

In Chapter 4, we generally did not automatically separate the test harness with a floorplan (only a partition), as the initial experiments explicitly dealt with floorplans. We did not floorplan larger and later designs (Chapters 5 through 7) as a result of our floorplanning and partitioning experiments in Chapter 4..

Place and Route We configure the place and route process to exert maximal effort at fitting with only two constraints: (i) to avoid using I/O pin registers to prevent artificially long paths that would affect the clock frequency, and (ii) to set the target clock frequency to 550MHz, which is the maximum clock frequency specified for M9K BRAMs. Setting a target frequency higher than 550MHz does not improve results and could in fact degrade them: for example, we use a half-speed derived clock in our designs, which would then aim towards an unnecessarily high target frequency, competing for fast routing paths.

Frequency We report the unrestricted maximum operating frequency (F_{max}) by averaging the results of ten place and route runs, each starting with a different random seed for initial placement. We construct the average F_{max} from the slow-corner timing reports which assume a die temperature of 85° and a supply voltage of 900mV. Note that minimum clock pulse width limitations in the BRAMs restrict the actual operating frequency to 550MHz, regardless of actual propagation delay. Reported F_{max} in excess of this limit indicates timing slack available to the designer.

Area In Chapter 3, we measure area as the count of Adaptive Lookup Tables (ALUTs) in use. We also measure the area efficiency as the percentage of ALUTs actually in use relative to the total number of ALUTs within the rectangular *LogicLock* area which contains the circuit under test, including any BRAMs or DSP Blocks. We allow Quartus to automatically place and size the *LogicLock* area. Area does not vary significantly between place and route runs, so we report the first computed result.

In later chapters (4–7), we measure area 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.* [125, 126][4]. By itself, a Stratix IV ALM roughly contains two 6-LUTs, two flip-flops, and two full-adders with carry-chain logic. These later designs have larger and more variable area, so we also average the area over the ten random initial seeds.

We count the number of ALMs and DSPs pessimistically from Quartus' Fitter report. We take the total count of used or partially-used ALMs, and the total count of entire DSP blocks used. Since we only use 36-bit words in Chapters 4–7, each Octavo instance always uses an integral number of DSP blocks: a 36×36 multiplier requires an entire Stratix IV DSP block.

[4]We had already performed the experiments of Chapter 3 before those area results appeared in publication.

Appendix B

Quartus Configuration

Listing B.1 contains the essential elements of the Quartus configuration used throughout this work, aimed at synthesizing fast hardware at the expense of area and CAD time. These settings originate from Quartus II 64-Bit, Version 13.1.0 (Build 162 10/23/2013 SJ Full Version), but have remained unchanged since we began with Quartus 10.1.

Some significant points:

- We disable multi-corner timing analysis, reporting only the worst-case slow process corner at 85°C and 900mV.
- We disable all shift register inference, as they operate too slowly and interfere with proper pipelining.
- We disable RAM to logic conversion (LCELL), as RAM results in denser, faster circuits in most cases.
- We disable IOC register placement, otherwise the test harness registers get placed in I/O registers along the edge of the device, introducing artificially long paths.
- We increase the placement and routing effort multipliers to 4, resulting in a small increase in CAD time, and ensuring best effort.

Listing B.1: Significant Quartus Configuration Details

```
# Project-Wide Assignments
# ========================
set_global_assignment -name LAST_QUARTUS_VERSION 13.1
set_global_assignment -name FLOW_DISABLE_ASSEMBLER ON
set_global_assignment -name SMART_RECOMPILE ON
set_global_assignment -name NUM_PARALLEL_PROCESSORS 2
<source and constraint file entries removed for brevity>
set_global_assignment -name FLOW_ENABLE_RTL_VIEWER OFF

# Classic Timing Assignments
# ==========================
set_global_assignment -name MIN_CORE_JUNCTION_TEMP 0
set_global_assignment -name MAX_CORE_JUNCTION_TEMP 85
set_global_assignment -name TIMEQUEST_MULTICORNER_ANALYSIS OFF
set_global_assignment -name TIMEQUEST_DO_REPORT_TIMING ON

# Analysis & Synthesis Assignments
# ================================
set_global_assignment -name FAMILY "Stratix IV"
set_global_assignment -name DEVICE_FILTER_SPEED_GRADE FASTEST
set_global_assignment -name OPTIMIZATION_TECHNIQUE SPEED
set_global_assignment -name ADV_NETLIST_OPT_SYNTH_WYSIWYG_REMAP ON
set_global_assignment -name AUTO_SHIFT_REGISTER_RECOGNITION OFF
set_global_assignment -name REMOVE_REDUNDANT_LOGIC_CELLS ON
set_global_assignment -name MUX_RESTRUCTURE ON
set_global_assignment -name ALLOW_ANY_ROM_SIZE_FOR_RECOGNITION ON
set_global_assignment -name ALLOW_ANY_RAM_SIZE_FOR_RECOGNITION ON
set_global_assignment -name ALLOW_ANY_SHIFT_REGISTER_SIZE_FOR_RECOGNITION OFF
set_global_assignment -name AUTO_RAM_RECOGNITION ON
set_global_assignment -name AUTO_RAM_TO_LCELL_CONVERSION OFF
set_global_assignment -name SYNTH_TIMING_DRIVEN_SYNTHESIS ON
set_global_assignment -name USE_LOGICLOCK_CONSTRAINTS_IN_BALANCING ON
set_global_assignment -name SAVE_DISK_SPACE OFF
set_global_assignment -name REMOVE_DUPLICATE_REGISTERS ON

# Fitter Assignments
# ==================
set_global_assignment -name DEVICE EP4SGX230KF40C2
set_global_assignment -name FITTER_EFFORT "STANDARD FIT"
set_global_assignment -name OPTIMIZE_IOC_REGISTER_PLACEMENT_FOR_TIMING OFF
set_global_assignment -name PHYSICAL_SYNTHESIS_COMBO_LOGIC ON
set_global_assignment -name PHYSICAL_SYNTHESIS_REGISTER_RETIMING ON
set_global_assignment -name PHYSICAL_SYNTHESIS_REGISTER_DUPLICATION ON
set_global_assignment -name PHYSICAL_SYNTHESIS_EFFORT EXTRA
set_global_assignment -name ROUTER_LCELL_INSERTION_AND_LOGIC_DUPLICATION ON
set_global_assignment -name ROUTER_TIMING_OPTIMIZATION_LEVEL MAXIMUM
set_global_assignment -name PHYSICAL_SYNTHESIS_COMBO_LOGIC_FOR_AREA OFF
set_global_assignment -name AUTO_PACKED_REGISTERS_STRATIXII AUTO
set_global_assignment -name ROUTER_CLOCKING_TOPOLOGY_ANALYSIS ON
set_global_assignment -name PHYSICAL_SYNTHESIS_MAP_LOGIC_TO_MEMORY_FOR_AREA OFF
set_global_assignment -name BLOCK_RAM_TO_MLAB_CELL_CONVERSION OFF
set_global_assignment -name SEED 1
set_global_assignment -name PLACEMENT_EFFORT_MULTIPLIER 4
set_global_assignment -name ROUTER_EFFORT_MULTIPLIER 4
set_global_assignment -name AUTO_DELAY_CHAINS OFF
set_global_assignment -name OPTIMIZE_HOLD_TIMING "ALL PATHS"

# Power Estimation Assignments
# ============================
set_global_assignment -name POWER_PRESET_COOLING_SOLUTION "23 MM HEAT SINK WITH 200 LFPM AIRFLOW"
set_global_assignment -name POWER_BOARD_THERMAL_MODEL "NONE (CONSERVATIVE)"

# Incremental Compilation Assignments
# ===================================
set_global_assignment -name RAPID_RECOMPILE_MODE OFF

# Netlist Viewer Assignments
# ==========================
set_global_assignment -name RTLV_GROUP_COMB_LOGIC_IN_CLOUD_TMV OFF

<partitioning assignments removed for brevity>
```

Appendix C

Suggested FPGA and CAD Changes

Throughout this work, we constantly strove to maximize the raw F_{max} of the Octavo architecture. Without a maximized operating speed, within the limitations of the underlying FPGA and its CAD tools, any efficiency or performance improvement rings hollow as the hardware still spends much time idle, waiting for the next clock edge.

We spent much of our efforts measuring and working around properties of the FPGA which normally don't manifest at lower speeds. In a nutshell, various hard blocks (DSP, BRAM) cannot operate as quickly as the reconfigurable logic, and the CAD tool prevents proper pipelining in some cases.

Often, overlays and soft-processors do not even reach 300 MHz on a Stratix IV device, beyond which we begin to observe difficulties maintaining speed. We outline these difficulties and suggest some changes to avoid them, with the understanding that the underlying causes may necessarily exist due to engineering and/or commercial constraints.

BRAM Clock Pulse Width Early self-loop characterization experiments (Chapter 3) showed that, based on propagation delay alone, a suitably pipelined M9K Block RAM could operate at over 700 MHz (out of a clock tree maximum rating of 800 MHz). However, limitations to the minimum clock pulse width prevent any operation past 550 MHz at best [10], or more typically 500 MHz based on a real device (Terasic DE4 using a

`EP4SGX230KF40C2` Stratix IV device). This single limitation currently defines the absolute ceiling F_{max} of Octavo.

BRAM Interconnections The same aforementioned self-loop characterization experiments revealed that when passing data from one BRAM to another, we must always include some pipelining to maximize F_{max}. This constraint creates the unusual offset between the Control Path and Data Path in Octavo (Figure 3.7), where we must place 3 pipeline stages between the Instruction memory and the A/B Data Memories. In this case, we suspect the first pipeline stage retimes into the Instruction memory output to offset a long t_{co} (Clock-to-Output Time) of the SRAM proper, while the second seems to offset the FPGA interconnect delay between BRAMs, regardless of actual distance. Adding a third stage improved F_{max} for wider-word designs (e.g.: 64 or 72 bits), and brought the Control Path pipeline depth up to 8, matching that of the Data Path. We don't have a clear picture of the causes, but some optimizations to reduce the need to pipeline between BRAMs would shorten overall processing system pipelines.

BRAM Forwarding Logic Quartus can automatically implement write-forwarding logic around BRAMs, so that a read coinciding with a write to the same address returns the newly written value rather than the original memory contents. This logic also enables the higher 500 MHz operating frequency, since the BRAM can internally perform both the read and write on the same clock edge, relying on the forwarding logic to discard any read data potentially corrupted by a simultaneous write to the same SRAM cells. Without forwarding, the reads and writes must internally occur on different edges, limiting operating frequency to 375 MHz. We have always found it advantageous to implement write forwarding to obtain the higher speed, then register the read or write data and address lines to emulate the original behaviour if necessary [76]. Thus, we suggest reversing the process: integrate the forwarding logic into the BRAM ASIC, defaulting to a higher F_{max}, and let Quartus auto-generate the required registering to undo its effect in the FPGA fabric if desired.

DSP Clock Pulse Width The DSP Blocks also suffer a minimum clock pulse width limitation similar to that of BRAMs, restricting their full-word (36 bits) operation to 480 MHz. Unlike BRAMs however, we can use two DSP Blocks in a ping-pong fashion (Chapter 3.9) to act as a single DSP Block operating at up to 600 MHz. Altering the DSP Blocks to run at higher speeds, even if this required using the full 3 stages of internal pipelining, would halve the number of DSP Blocks required in Octavo.

Register Retiming At the time of writing, Quartus did not allow retiming registers into auto-generated modules, which artificially limits F_{max}. In our case, we use a vendor-supplied Verilog module definition to specify the exact operation we required of the DSP block, leaving Quartus to synthesize the proper logic required for various word widths not equal to the native 36 bits. For larger word widths (e.g.: 72 bits), Quartus correctly infers multiple 36-bit multipliers with extra adders to combine their partial results. However, since this inferred hardware lies within the vendor-supplied module definition, and since the definition does not provide means to specify additional pipelining (at the time of writing), those unpipelined extra adders become the critical path. Manually added pipeline registers at the output do not get retimed into the adder tree, ultimately (and artificially) limiting the performance of wider-word versions of Octavo. Thus, we need some means of controlling register retiming into vendor-generated logic, or manually inserting registers after generation.

Vendor Module Pipelining Related to the register retiming issue above, we have to rely on a retiming trick to specify the pipelining we need in the DSP Blocks. We use the DSP Block purely as a multiplier, ignoring the final adder meant for Multiply-Accumulate functionality. However, the vendor-supplied module definition only allows enabling the 3 internal pipeline stages in sequence from input towards output, whereas we need only the first and last pipeline stages to absorb the delay of the unused MAC adder. We solve the problem by enabling the first DSP Block pipeline stage, then manually adding a pipeline register after the module output. Later post-synthesis optimizations in Quartus will retime that final register into the DSP Block's internal third pipeline stage, despite the lack of register retiming seen for larger multipliers above.

Appendix D

Limitations Of The Verilog HDL

Engineers often criticize HDLs as too low-level for large-scale system development, akin to assembly language or plain C. Instead, this work claims the unsuitability of HDLs stems from how they hamper disciplined and sophisticated development techniques and tools, and not from the usual low-level concerns of signed arithmetic [117][1], blocking vs. non-blocking assignments and race conditions across processes [58], memory block inference [4], etc... *none of which caused significant difficulties in this work.*

However, the creation of this work suffered from the following limitations in Verilog-2001, which generally prevented constructing abstractions to support large-system development. At the time of writing, SystemVerilog, an updated version of Verilog, fixed many of these limitations, but support for it was relatively new and porting Octavo was too risky at the time.

[1] Even though the proposed signed addition solution does not work on our FPGA CAD tool (Quartus).

Macros Verilog macros only perform simple text substitution with no possible processing, making it impossible to generate file names, variable names, or module names. The only method for concatenation treats strings as arrays of literal ASCII bytes, so appending the number '20' to a string requires an explicit manual translation of the integer '20' to the bytes representing the characters '2' and '0'. SystemVerilog's macros have new system functions to concatenate strings and convert to/from integer, hex, and floating-point numbers.

Parameters Module parameters can only hold integers or strings, preventing programmatic generation of similar but not identical modules (e.g. different memory contents for each instance), and also increasing the number of parameters to manage. Under SystemVerilog, parameters can hold any data type.

Introspection We cannot peer inside the data structures of the HDL to locally obtain relevant data. For example, one module cannot access the parameters of another, forcing large, hierarchical designs to manually pass all possible parameters top-down. Furthermore, since we cannot store parameters in a structure or variable, we must pass them manually one-by-one: the number of parameters in module definitions and instantiations rapidly exceeds the number of actual lines of code. So far, SystemVerilog does not support introspection across modules, but can create "packages" of global data containing parameters and functions accesible to all modules, which would avoid having to pass all paramters down the module hierarchy.

Connecting Ports Similarly, we cannot directly refer to the ports of another module to express connections (e.g.: `moduleA.portB(moduleB.portC)`). Instead, the upper-level encapsulating module must create vectors of (arbitrarily named) wires for each of the sub-module ports (again, without being able to query the sub-modules themselves), and create `assign` statements to connect the wires together. Furthermore, `assign` statements have a different precedence than the usual non-blocking assignments in the Verilog simulation model, potentially causing simulation/synthesis mismatches. It is unclear if SystemVerilog can address this problem.

Passing Vectors Through Ports Multi-dimensional vectors (e.g.: a 10x10 array of 8-bit bytes) cannot pass through module ports, forcing difficult and error prone use of macros and functions to pack/unpack vectors at each module boundary. Also, the number of ports in a module cannot be parameterized, only their bit-width, also forcing the packing and unpacking of related ports into a single port passing a flat bit vector. SystemVerilog can pass vectors through ports, and can parameterize the number of ports via interface definitions, which place the ports into a data structure rather than manually listing them in the source code.

Module Instance Arrays We can only instantiate unidimensional arrays of identical modules. Combined with the above limitations on ports and parameters, this limitation makes the creation of systolic arrays, meshes, networks, and other non-trivial parallel processing systems nearly impossible. The equivalent difficulty in C would resemble processing multi-dimensional data with only linear arrays and without the use of pointers, forcing the designer to manually implement every single array address calculation. SystemVerilog can instantiate multi-dimensional arrays of module instances.

Vector Reduction Verilog only supplies vector-to-bit reductions (e.g.: `^foo`, which `XOR`s all the bits of `foo` to a single bit). Expressing vector reduction as multiple parallel bit reductions failed to synthesize. The only synthesizable way to express a vector-to-vector reduction requires an iterated reduction over each array index i, where $\odot$ refers to any 2-input operator:

$$result = result \odot array[i] \tag{D.1}$$

which corresponds to a *left fold* operation:

$$result = ((((nil \odot array[0]) \odot array[1]) \odot array[2]) \odot \ldots) \tag{D.2}$$

where *nil* refers to an initial value of *result*, usually one which makes $\odot$ act as the identity function (e.g.: 0 for `XOR`). This approach may fail if $\odot$ does not have a constant *nil* value, or may return inaccurate results if $\odot$ represents a non-associative operation (e.g.: floating-point), requiring a careful selection of the iteration order or more complex code to express a tree reduction. Compounding the problem, no mechanism exists for defining new operators, and we cannot use functions which could serve that purpose outside of their defining module. In SystemVerilog, we could place an appropriate vector reduction function inside a global package, accessible to all modules.

Lack of Abstraction While SystemVerilog addresses most of these issues, we wanted to outline the overall problem which generally plagues HDL development: we can straightforwardly describe any *one* instance of a system, but we find describing *kinds* of systems nearly impossible. Without a capacity to transparently compose larger systems from smaller ones, we must tailor each system to fit and cannot create a common system architecture upon which we can make incremental progress.

Summary This work does not significantly suffer from the low-level, register-and-logic hardware descriptions of Verilog. We can easily describe each module in a flexible, parameterized manner for design space exploration, and we need that fine control to optimize our logic to the underlying FPGA. However, as we compose modules together and increase the scale of the design, we find that the above limitations prevent us from increasing Verilog's level of abstraction by the same amount, forcing a low-level description of high-level designs.

Appendix E

I/O Predication Of Branches

We introduced the Branch Trigger Module (BTM) in Chapter 6 to fold branches, executing them in parallel with other instructions. However, in the interest of simplicity, we presented the BTM mostly independently of Octavo's pipeline. This simplification ignores the impact of the I/O Predication mechanism we introduced in Chapter 5.

If an instruction accesses an un-ready I/O port, the `I/O Ready` signal (Figure 5.5, pg. 93) will not assert, which annuls the instruction into a `NOP` and causes the Controller (Figure 5.6, pg. 94) to re-issue that instruction's Program Counter value. However, since a folded branch may depend on the result of the previous thread instruction, and a re-issued instruction always has a `NOP` as its previous instruction, we have lost the original previous instruction result. The folded branch might then proceed incorrectly.

Figure E.1 shows the actual Octavo BTM, which preserve the flag used by a folded branch across instruction annullment and re-issue(s). Figure 6.3 (pg. 111) shows the simplified BTM for comparison. In the lower right, we preserve copies of the original Jump (`J`) and `I/O Ready` flags out of Stage 3, and pipeline them 6 and 7 stages respectively 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 the current instruction is a re-issue of the previous one. If so (i.e.: `I/O Ready` not asserted), then we re-use the delayed Jump flag instead of the current selected flag `F`, re-creating the original conditions for the folded branch.

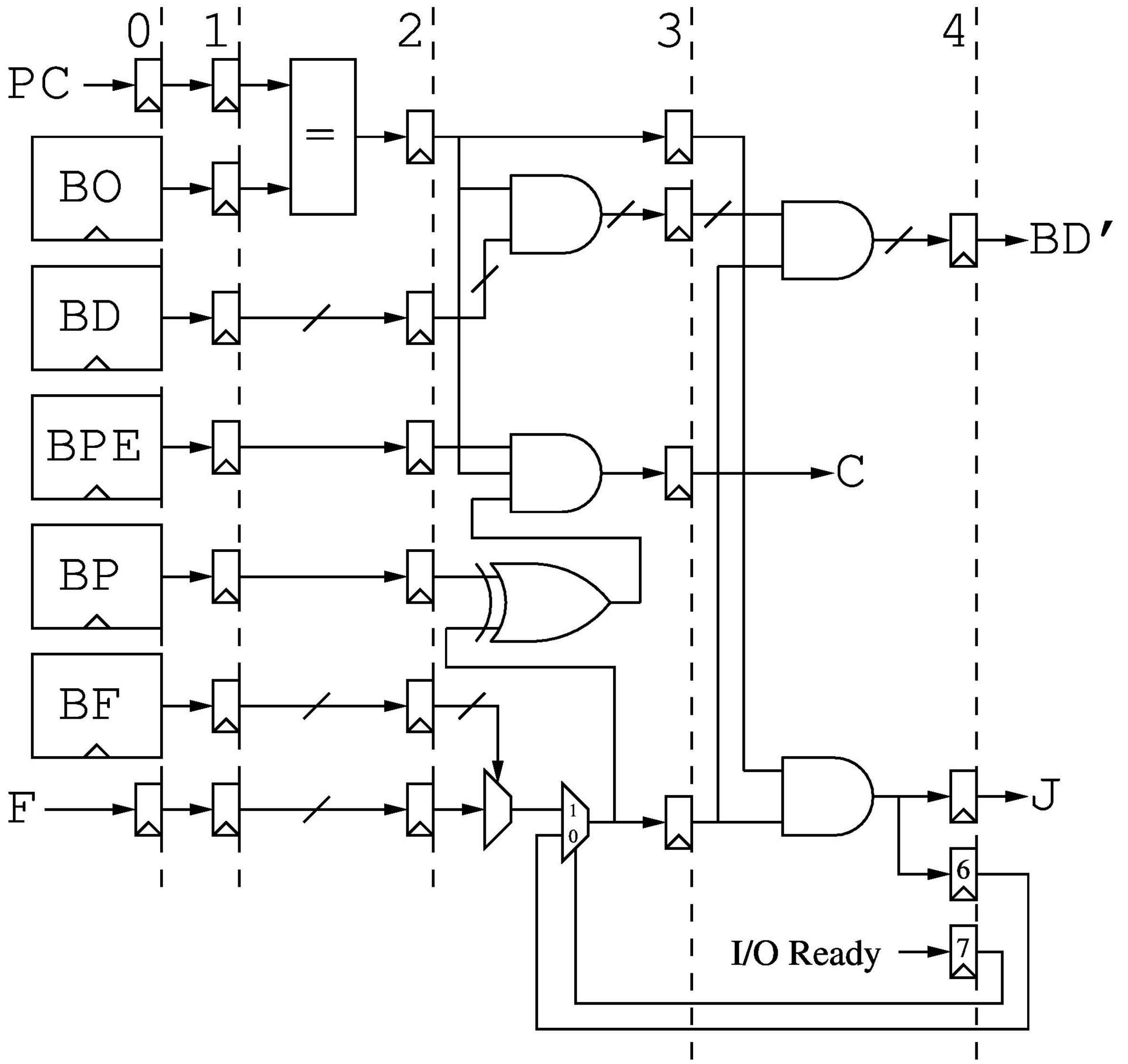


Figure E.1: Branch Trigger Module Altered for I/O Predication

Appendix F

External Memory Interfaces

Octavo's original design premises (Chapter 3) state that we should avoid the associative structure overhead of caches on FPGAs (see Severance and Lemieux [108] for an overview), and instead collapse the on-chip memory hierarchy into simpler, faster scratchpad memories. Thus, Octavo's address spaces do not extend beyond its scratchpads and internal memory-mapped hardware. Any data from outside must come in through the A/B memory I/O read ports, while outgoing data must go through I/O write ports or through hardware mapped into High memory (Chapter 5).

If we assume an external DRAM memory attached via a simple controller to handle basic row/column access and refresh protocols, reading and writing to memory requires sending an address, then either sending or receiving data, both via handshaking I/O ports. Each step requires an instruction, which can also fold some or all of the address and data calculations with the I/O transfers, as well as allow the programming system to mitigate memory latency.

Listings F.1 and F.2 show the Octavo equivalents of the MIPS IV "Load Word" and "Store Word" instructions [98]. For both loads and stores, the first instruction adds the address `base` and `offset` together and writes them, via an I/O write port (`I/Ow`), to the memory controller. A load will then read the data from an I/O read port (`I/Or`) and store it into `data`. A store does the opposite, writing `data` to `I/Ow`.

Listing F.1: External Memory Load

```
// data <-- MEM[base+offset]
ADD I/Ow, base, offset
ADD data, I/Or, 0
```

Listing F.2: External Memory Store

```
// MEM[base+offset] <-- data
ADD I/Ow, base, offset
ADD I/Ow, data, 0
```

However, unlike MIPS IV, we have avenues to optimize load/store performance without additional hardware. We can replace the initial address write with arbitrary calculations writing to the same I/O write port. We can also replace the data read/write with the calculation which generates or consumes `data` in the first place. If the programming system knows the expected memory latency, it can insert other instructions between the address write and the data read/write. Finally, in all cases, if the memory controller cannot yet receive an address or send/receive data the relevant instruction will "hang", via the I/O Predication mechanism (Chapter 5), until the operation completes.

Additionally, if the application has connected (e.g.: graph traversal) or highly-local (e.g.: array sub-tiles) memory access patterns, we can design and attach an Accelerator to programmatically interact with memory, while the main processor proceeds with computation[1].

Finally, some thought will have to go into the design of the memory controller, its memory channels, and the software's access patterns, as Octavo's multiple threads could end up interleaving memory requests with little relative locality. Without some planning, we would end up wasting most of the potential memory bandwidth waiting for the controller to open/close unrelated rows in DRAM. We can base ourselves on the prior work of Labrecque *et al.* [73], which explored memory hierarchies for multi-threaded multi-processors.

[1]The process of designing a memory Accelerator strikingly resembles that of designing a Display Processor [94].

Appendix G

Accelerated Hailstone Algorithm

This appendix describes the construction of the Accelerated Hailstone algorithm used in Chapter 7. We lifted much of the text verbatim[1] from the "Collatz conjecture" Wikipedia article [124], as it extracts the algorithm out of far less tractable mathematical publications, primarily a 2007 paper by Scollo [103], with some deeper details buried in a (difficult!) 1976 article by Riho [113].

G.1 Introduction

The Collatz conjecture is a conjecture in mathematics named after Lothar Collatz, who first proposed it in 1937. Take any natural number n. If n is even, divide it by 2 to get $n/2$. If n is odd, multiply it by 3 and add 1 to obtain $3n+1$:

$$f(n) = \begin{cases} n/2 & \text{if } n \equiv 0 \\ 3n+1 & \text{if } n \equiv 1 \end{cases} \pmod 2$$

If we apply the formula indefinitely, the conjecture states that no matter the initial value of n, you will always eventually reach 1. Any further iteration remains within a

[1]The Wikimedia Foundation publishes under the Creative Commons Attribution-ShareAlike License, which allows creating new derived works, so long as they cite the source and fall under the same license, which this Appendix thus does.

$1, 4, 2, 1 \ldots$ cycle. We refer to the sequence of numbers involved as the Hailstone sequence or Hailstone numbers, because the sequence values usually rise and fall like hailstones in a cloud, before ultimately falling to 1.

G.2 As a parity sequence

For this section, consider the Collatz function in the slightly modified form

$$f(n) = \begin{cases} n/2 & \text{if } n \equiv 0 \\ (3n+1)/2 & \text{if } n \equiv 1 \end{cases} \pmod{2}$$

This modification works because when n is odd, $3n + 1$ is always even, keeping n in the natural numbers. Also, as a consequence, we step through the sequence of numbers faster, a presumably useful property to mathematicians trying to disprove the conjecture by finding cycles in Hailstone sequences.

If $P(\ldots)$ is the parity of a number, that is $P(2n) = 0$ and $P(2n + 1) = 1$, then we can define the Hailstone parity sequence (or parity vector) for a number n as $p_i = P(a_i)$, where $a_0 = n$, and $a_i + 1 = f(a_i)$. What operation is performed ($(3n + 1)/2$ or $n/2$) depends on the parity. The parity sequence is the same as the sequence of operations.

Using this form for $f(n)$, it can be shown that the parity sequences for two numbers m and n will agree in the first k terms if and only if m and n are equivalent modulo 2^k. This implies that every number is uniquely identified by its parity sequence, and moreover that if there are multiple Hailstone cycles, then their corresponding parity cycles must be different.

Applying the f function k times to the number $a \cdot 2^k + b$ will give the result $a \cdot 3c + d$, where d is the result of applying the f function k times to b, and c is how many odd numbers were encountered during that sequence.

G.3 Space-time Trade-Off

The calculation of parity sequences gives a way to speed up the calculation of Hailstone sequences. To jump ahead k steps on each Hailstone iteration (using the aforementioned f function), break up the current number into two parts, b (the k least significant bits, interpreted as an integer), and a (the rest of the bits as an integer). The result of jumping ahead k steps can be found as:

$$f^k(a \cdot 2^k + b) = a \cdot 3^{c[b]} + d[b]$$

The c and d arrays are precalculated for all possible k-bit numbers b, where $d[b]$ is the result of applying the f function k times to b, and $c[b]$ is the number of odd numbers encountered on the way [103]. For example, if $k = 5$, you can jump ahead 5 steps on each iteration by separating out the 5 least significant bits of a number and using:

$$c = [0, 3, 2, 2, 2, 2, 2, 4, 1, 4, 1, 3, 2, 2, 3, 4, 1, 2, 3, 3, 1, 1, 3, 3, 2, 3, 2, 4, 3, 3, 4, 5]$$

$$d = [0, 2, 1, 1, 2, 2, 2, 20, 1, 26, 1, 10, 4, 4, 13, 40, 2, 5, 17, 17, 2, 2, 20, 20, 8, 22, 8, 71, 26, 26, 80, 242]$$

Creating each of c and d requires 2^k precomputations and storage elements to speed up the resulting calculation by a factor of k, a space-time trade-off.

Since the Octavo and MXP systems in this work use limited on-chip scratchpad memory, and compute over pipeline depths of at or near 8, precomputing tables for $k = 8$ presented a natural choice. We can then conventionally pre-compute the first k values of a Hailstone sequence, then interleave k accelerated computations, each starting with one of the k initial values, to fill the pipeline and compute the entire Hailstone sequence k steps at a time.

Appendix H

Benchmark Code From Chapter 6

This appendix gives the pseudo-code, MIPS-like assembly, and optimized Octavo assembly code for the benchmarks we use in Chapter 6, except for Hailstone, whose code we describe in Chapter 6.3.2 (pg. 102). These benchmarks test the execution overhead reductions of the Address Offset Module (AOM) and Branch Trigger Module (BTM).

We omit some details for clarity, such as the memory mapping and data format of the AOM/BTM entries. The MIPS-like code resembles the MIPS IV ISA [98], while Table 3.2 (pg. 35) describes the Octavo ISA.

Some notation details:

- All names refer to values held in registers. We implement moves with the "`ADD` To Zero" idiom.

- We describe I/O ports as simple pointers in the pseudo-code, or addresses (e.g.: "`out`") in the MIPS-like and Octavo assembly. Most Octavo pointer addresses shown here post-increment after use.

- We omit initialization code, such as BTM and AOM entries which do not change during execution.

- We denote updating a BTM entry by writing the kind of branch to the BTM (e.g.: `ADD BTM, JNEn, 0`), configuring the BTM to implement the next branch of that kind seen in the source code.

- We similarly denote updates to the AOM (e.g.: `ADD AOM, 0, ptr_init`), implying that the written value contains the offsets, increments, etc...of an AOM entry.

- Both AOM and BTM updates have a 1-cycle RAW hazard before they take effect. A useful instruction almost always fills the delay slot, though you will see the occasional `NOP` opcode as shorthand for `XOR, 0, 0, 0` (the all-zero instruction).

- In Octavo code, we place folded branches on the same line as the regular instruction, separated by a semi-colon. The branch executes concurrently, with conditionals based on the result of the previous instruction. We denote "Predict Taken" and "Predict Not Taken" cancelling branches by appending "`t`" or "`n`" to the branch opcode. An absent letter suffix denotes a non-cancelling branch, which makes no prediction and never cancels the concurrent instruction. Note that the branch opcode acts as a source notation only: the BTM eliminates all branch instructions, and the equivalent functionality now resides in BTM entries.

H.1 Array Increment

The Array Increment benchmark increments by 1 a negative-terminated array of 10 elements (`in` inner loop), repeated 10 times (`out` outer loop), then outputs the entire array (`oput` loop), showing a simple iterated calculation with a separate output loop. On Octavo, this benchmark requires five branches, forcing us to periodically reload one of the four BTM entries to support it. Listings H.1, H.2, and H.3 contain the pseudo-code, MIPS-like assembly, and optimized Octavo assembly code.

Listing H.1: Array Increment Pseudo-Code

```
// out is an I/O write port
// array = [...,-1];

init: cnt = 10;
out:  ptr = &array;           // outer loop
in:   tmp = *ptr;             // inner loop, incr. array
      if (tmp < 0) goto brk;
      tmp += 1;
      *ptr = tmp
      ptr += 1;
      goto in;
brk:  cnt -= 1;               // Repeat inner cnt times
      if (cnt > 0) goto out;
      ptr = &array;
oput: tmp = *ptr              // Output array and re-start
      if (tmp < 0) goto init;
      *out = tmp;
      ptr += 1;
      goto oput;
```

Listing H.2: Array Increment MIPS-like Code

```
// out is an I/O write port
// array    = [...,-1];
// cnt_init = 10

init: ADD  cnt,  cnt_init, 0
out:  ADD  ptr,  &array,   0 // outer loop
in:   LW   tmp,  ptr         // inner loop, incr. array
      BLTZ brk,  tmp
      ADD  tmp,  tmp,      1
      SW   tmp,  ptr
      ADD  ptr,  ptr,      1
      JMP  in
brk:  SUB  cnt,  cnt,      1 // Repeat inner cnt times
      BGTZ out,  cnt
      ADD  ptr,  &array,   0
oput: LW   tmp,  ptr
      BLTZ init, tmp         // Output array and re-start
      SW   tmp,  out
      ADD  ptr,  ptr,      1
      JMP  oput
```

Listing H.3: Array Increment Octavo Code

```
// out is an I/O write port
// array = [...,-1];
// cnt_init = 10
// ptr_init contains &array offset and increment values
// one   =  1
// m_one = -1

init: ADD BTM, JNEn,  0                      // setup branch
      ADD cnt, 0,     cnt_init
      ADD AOM, 0,     ptr_init
out:  NOP                                    // outer loop
in:   ADD tmp, 0,     ptr                    // inner loop
      ADD tmp, one,   tmp      ; JNEn break
      ADD ptr, 0,     tmp      ; JMP  in
brk:  ADD cnt, m_one, cnt                    // do cnt times
      ADD AOM, 0,     ptr_init ; JNZ  out
      ADD BTM, JPO,   0                      // re-use entry
oput: ADD tmp, 0,     ptr                    // output array
      ADD out, 0,     tmp      ; JNEn init ; JPO oput
```

H.2 Array Reverse

The Array Reverse benchmark traverses an array of 100 elements using two pointers, top-to-middle (`top`) and bottom-to-middle (`bot`), loading and storing to swap their values without any computation or other I/O. The bottom-to-middle pointer decrements, which (on Octavo) our current AOM does not support and thus we must do manually. Furthermore, due to the write-only nature of the AOM, we have to keep a copy of the bottom-to-middle pointer (`bot_tmp`)), add −1 to it (`bot_decr`), then update its AOM entry, all within the main loop. Listings H.4, H.5, and H.6 contain the pseudo-code, MIPS-like assembly, and optimized Octavo assembly code.

Listing H.4: Array Reverse Pseudo-Code

```
// array = [ ... ];
// len = length(array)
// half_len = len / 2;
// top_init = &array;
// bot_init = &array + len - 1;

init: top = top_init;
      bot = bot_init;
      cnt = half_len;
next: tmp_top = *top;          // reverse values
      tmp_bot = *bot;
      *top    = tmp_bot;
      *bot    = tmp_top;
      top += 1;                // incr/decr pointers
      bot -= 1;
      cnt -= 1;
      if (cnt > 0) goto next;
      goto init                // reverse again
```

Listing H.5: Array Reverse MIPS-like Code

```
// array = [ ... ];
// len = length(array)
// half_len = len / 2;
// top_init = &array;
// bot_init = &array + len - 1;

init: ADD  top,      0,  top_init
      ADD  bot,      0,  bot_init
      ADD  cnt,      0,  half_len
next: LW   tmp_top,  top           // reverse values
      LW   tmp_bot,  bot
      SW   tmp_bot,  top
      SW   tmp_top,  bot
      ADD  top,       1, top       // incr/decr pointers
      ADD  bot,      -1, bot
      ADD  cnt,      -1, cnt
      BGTZ next,         cnt
      JMP  init                    // reverse again
```

Listing H.6: Array Reverse Octavo Code

```
// array = [ ... ];
// len = length(array)
// half_len = len / 2;
// top_init = &array;           (plus other AOM data)
// bot_init = &array + len - 1; (plus other AOM data)
// bot_decr = -1                (formatted for bot_init)
// m_one    = -1

      ADD bot_tmp, 0,         bot_init // local copy
init: ADD AOM,     0,         top_init
      ADD AOM,     0,         bot_init
      ADD cnt,     0,         half_len
next: ADD tmp_top, 0,         top      // reverse values
      ADD tmp_bot, 0,         bot
      ADD bot,     0,         tmp_top
      ADD top,     0,         tmp_bot
      ADD bot_tmp, bot_decr,  bot_tmp  // decr. bot manually
      ADD cnt,     m_one,     cnt
      ADD AOM,     0,         bot_tmp  ; JNZ next
      ADD bot_tmp, 0,         bot_init ; JMP init
```

H.3 8-Tap FIR Filter

We sequentially read a 100-entry input buffer (`in`), applying an 8-tap FIR filter at each step, and output the filtered values to a 100-entry output buffer (`out`). We keep all 8 taps and 8 coefficients in registers, shifting values down a taps buffer (`buf0--buf7`) then performing the multiplications and additions (`acc_tmp` and `mul_tmp`), both as unrolled inner loops. Listings H.7, H.8, and H.9 contain the pseudo-code, MIPS-like assembly, and optimized Octavo assembly code.

Listing H.7: FIR Filter Pseudo-Code

```
// input_array  = [ ... ];
// output_array = [ ... ];
// in_init      = &input_array;
// out_init     = &output_array;
// cnt_init     = length(input_array);
// buf0 to buf7 are buffer registers
// cft0 to cft7 are coefficient registers

init: in  = in_init;
      out = out_init;
      cnt = cnt_init;
loop: buf7 = buf6;           // shift buffer
      buf6 = buf5;
      buf5 = buf4;
      buf4 = buf3;
      buf3 = buf2;
      buf2 = buf1;
      buf1 = buf0;
      buf0 = *in;            // load new input
      acc_tmp = buf7 * cft7; // multiply-accumulate
      mul_tmp = buf6 * cft6;
      acc_tmp += mul_tmp;
      mul_tmp = buf5 * cft5;
      acc_tmp += mul_tmp;
      mul_tmp = buf4 * cft4;
      acc_tmp += mul_tmp;
      mul_tmp = buf3 * cft3;
      acc_tmp += mul_tmp;
      mul_tmp = buf2 * cft2;
      acc_tmp += mul_tmp;
      mul_tmp = buf1 * cft1;
      acc_tmp += mul_tmp;
      mul_tmp = buf0 * cft0;
      acc_tmp += mul_tmp;
      *out = acc_tmp;        // store output
      in  += 1;              // update pointers
      out += 1;
      cnt -= 1;
      if (cnt == 0) goto init;
      goto loop;
```

Listing H.8: FIR Filter MIPS-like Code

```
// input_array  = [ ... ];
// output_array = [ ... ];
// in_init      = &input_array;
// out_init     = &output_array;
// cnt_init     = length(input_array);
// buf0 to buf7 are buffer registers
// cft0 to cft7 are coefficient registers

init: ADD  in,      in_init,  0
      ADD  out,     out_init, 0
      ADD  cnt,     cnt_init, 0
loop: ADD  buf7,    buf6,     0       // shift buffer
      ADD  buf6,    buf5,     0
      ADD  buf5,    buf4,     0
      ADD  buf4,    buf3,     0
      ADD  buf3,    buf2,     0
      ADD  buf2,    buf1,     0
      ADD  buf1,    buf0,     0
      LW   buf0,    in                // load new input
      MUL  acc_tmp, buf7,     cft7    // multiply-accum.
      MUL  mul_tmp, buf6,     cft6
      ADD  acc_tmp, acc_tmp,  mul_tmp
      MUL  mul_tmp, buf5,     cft5
      ADD  acc_tmp, acc_tmp,  mul_tmp
      MUL  mul_tmp, buf4,     cft4
      ADD  acc_tmp, acc_tmp,  mul_tmp
      MUL  mul_tmp, buf3,     cft3
      ADD  acc_tmp, acc_tmp,  mul_tmp
      MUL  mul_tmp, buf2,     cft2
      ADD  acc_tmp, acc_tmp,  mul_tmp
      MUL  mul_tmp, buf1,     cft1
      ADD  acc_tmp, acc_tmp,  mul_tmp
      MUL  mul_tmp, buf0,     cft0
      ADD  acc_tmp, acc_tmp,  mul_tmp
      SW   acc_tmp, out               // store output
      ADD  in,      in,       1       // update pointers
      ADD  out,     out,      1
      ADD  cnt,     cnt,     -1
      BEQZ init,    cnt
      JMP  loop
```

Listing H.9: FIR Filter Octavo Code

```
// input_array  = [ ... ];
// output_array = [ ... ];
// in_init      = &input_array;
// out_init     = &output_array;
// cnt_init     = length(input_array);
// ptr_init contains AOM data for both in_init and out_init
// buf0 to buf7 are buffer registers
// cft0 to cft7 are coefficient registers
// m_one        = -1

init: ADD AOM,     0,        ptr_init
      ADD cnt,     0,        cnt_init
loop: ADD buf7,    0,        buf6     // shift buffer
      ADD buf6,    0,        buf5
      ADD buf5,    0,        buf4
      ADD buf4,    0,        buf3
      ADD buf3,    0,        buf2
      ADD buf2,    0,        buf1
      ADD buf1,    0,        buf0
      ADD buf0,    0,        in       // load new input
      MLS acc_tmp, cft7,     buf7     // multiply-accum.
      MLS mul_tmp, cft6,     buf6
      ADD acc_tmp, acc_tmp,  mul_tmp
      MLS mul_tmp, cft5,     buf5
      ADD acc_tmp, acc_tmp,  mul_tmp
      MLS mul_tmp, cft4,     buf4
      ADD acc_tmp, acc_tmp,  mul_tmp
      MLS mul_tmp, cft3,     buf3
      ADD acc_tmp, acc_tmp,  mul_tmp
      MLS mul_tmp, cft2,     buf2
      ADD acc_tmp, acc_tmp,  mul_tmp
      MLS mul_tmp, cft1,     buf1
      ADD acc_tmp, acc_tmp,  mul_tmp
      MLS mul_tmp, cft0,     buf0
      ADD acc_tmp, acc_tmp,  mul_tmp
      ADD cnt,     m_one,    cnt
      ADD out,     acc_tmp,  0,       ; JZE init ; JNZ loop
```

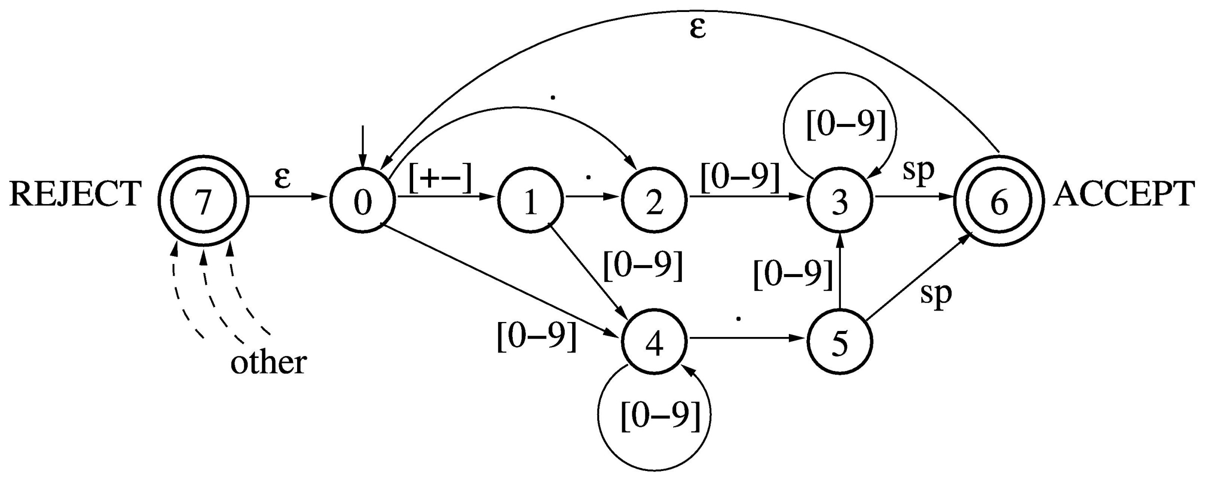


Figure H.1: Simple Floating-Point Number Recognizer

H.4 Floating-Point FSM

The Floating-Point Number FSM (Chapter 6), FSM-S, and FSM-A benchmarks (Chapter 7) all implement a Finite-State Machine (FSM) which recognizes simple floating-point numbers without exponent notation (e.g.: 5., −.5, .5, −5., −5.5, 5.5). Figure H.1 shows the state diagram for the associated regular expression `[+-]?(\.[0-9]+|[0-9]+\.[0-9]*)`, with the input alphabet `0 to 9, +, -, ., sp (space),` ϵ `(empty)`, and the output alphabet `ACCEPT, REJECT`. Any other transitions enter the `REJECT` state.

The following code examples show the FSM-A version, where we implement the FSM directly into the program structure, alternating tests and branches to determine state and action. FSM-A contains no real loops, no hoistable computations, no significant addressing, no basic block longer than 2 or 3 instructions, and has 34 unique branches, greatly exceeding the capacity of Octavo's four-entry BTM and forcing us to continually reload a single BTM entry with the data for the next upcoming branch. However, we can fold that BTM entry reload (for the next branch) with the branch set by the previous reload, and we can use the BTM to statically cache three other branches to save cycles on the most commonly traversed edges (denoted by a "`// BTM`" comment). Listings H.10, H.11, and H.12 contain the pseudo-code, MIPS-like assembly, and optimized Octavo assembly code.

Listing H.10: FSM-A Pseudo-Code

```
// array    = [ ..., -1 ]
// ptr_init = &array
// accept and reject are I/O ports

init:   ptr = ptr_init;

state0: tmp = *ptr;
        if (tmp < 0)    goto init;
        ptr += 1;
        if (tmp == ' ') goto state0;
        if (tmp == '+') goto state1;
        if (tmp == '-') goto state1;
        if (tmp == '.') goto state2;
        if (tmp <  '0') goto state7;
        if (tmp <= '9') goto state4;
        goto state7;

state1: tmp = *ptr;
        if (tmp < 0)    goto init;
        ptr += 1;
        if (tmp == '.') goto state2;
        if (tmp <  '0') goto state7;
        if (tmp <= '9') goto state4;
        goto state7;

state2: tmp = *ptr;
        if (tmp < 0)    goto init;
        ptr++;
        if (tmp <  '0') goto state7;
        if (tmp <= '9') goto state3;
        goto state7;

state3: tmp = *ptr;
        if (tmp < 0)    goto init;
        ptr += 1;
        if (tmp == ' ') goto state6;
        if (tmp <  '0') goto state7;
        if (tmp <= '9') goto state3;
        goto state7;

state4: tmp = *ptr;
        if (tmp < 0)    goto init;
        ptr += 1;
        if (tmp == '.') goto state5;
        if (tmp <  '0') goto state7;
        if (tmp <= '9') goto state4;
        goto state7;

state5: tmp = *ptr;
        if (tmp < 0)    goto init;
        ptr += 1;
        if (tmp == ' ') goto state6;
        if (tmp <  '0') goto state7;
        if (tmp <= '9') goto state3;
        goto state7;

state6: *accept = 1;
        goto state0;

state7: *reject = 1;
        goto state0;
```

Listing H.11: FSM-A MIPS-like Code

```
// array    = [ ..., -1 ]
// ptr_init = &array
// space    = ' '
// plus     = '+'
// minus    = '-'
// dot      = '.'
// char0    = '0' (decimal 48)
// accept and reject are I/O ports

init:   ADD  ptr,    0,   ptr_init

state0: LW   tmp,    ptr          // load next char
        BLTZ init,   tmp
        ADD  ptr,    1,   ptr
        XOR  tmp2,   tmp, space // is it a space?
        BEQZ state0, tmp2       // new state
        XOR  tmp2,   tmp, plus
        BEQZ state1, tmp2
        XOR  tmp2,   tmp, minus
        BEQZ state1, tmp2
        XOR  tmp2,   tmp, dot
        BEQZ state2, tmp2
        SUB  tmp2,   tmp, char0 // range check [0-9]
        BLTZ state7, tmp2
        SUB  tmp2,   10,  tmp2
        BGEZ state4, tmp2
        JMP  state7             // else: reject!

state1: LW   tmp,    ptr
        BLTZ init,   tmp
        ADD  ptr,    1,   ptr
        XOR  tmp2,   tmp, dot
        BEQZ state2, tmp2
        SUB  tmp2,   tmp, char0
        BLTZ state7, tmp2
        SUB  tmp2,   10,  tmp2
        BGEZ state4, tmp2
        JMP  state7

state2: LW   tmp,    ptr
        BLTZ init,   tmp
        ADD  ptr,    1,   ptr
        SUB  tmp2,   tmp, char0
        BLTZ state7, tmp2
        SUB  tmp2,   10,  tmp2
        BGEZ state3, tmp2
        JMP  state7

state3: LW   tmp,    ptr
        BLTZ init,   tmp
        ADD  ptr,    1,   ptr
        XOR  tmp2,   tmp, space
        BEQZ state6, tmp2
        SUB  tmp2,   tmp, char0
        BLTZ state7, tmp2
        SUB  tmp2,   10,  tmp2
        BGEZ state3, tmp2
        JMP  state7

state4: LW   tmp,    ptr
        BLTZ init,   tmp
        ADD  ptr,    1,   ptr
        XOR  tmp2,   tmp, dot
        BEQZ state5, tmp2
        SUB  tmp2,   tmp, char0
        BLTZ state7, tmp2
        SUB  tmp2,   10,  tmp2
        BGEZ state4, tmp2
        JMP  state7
```

```

state5: LW   tmp,    ptr
        BLTZ init,   tmp
        ADD  ptr,    1,   ptr
        XOR  tmp2,   tmp, space
        BEQZ state6, tmp2
        SUB  tmp2,   tmp, char0
        BLTZ state7, tmp2
        SUB  tmp2,   10,  tmp2
        BGEZ state3, tmp2
        JMP  state7

state6: SW   1,      accept
        JMP  state0

state7: SW   1,      reject
        JMP  state0
```

Listing H.12: FSM-A Octavo Code

```
// array    =  [ ..., -1 ]
// ptr_init =  &array (and other AOM data such as increment)
// space    =  ' '
// plus     =  '+'
// minus    =  '-'
// dot      =  '.'
// char0    =  '0' (decimal  48)
// m_char0  = -'0' (decimal -48)
// ten      =  10
// one      =   1
// accept and reject are I/O ports

init:   ADD AOM,    0,       ptr_init
        NOP                                   // RAW haz.

state0: ADD tmp,    0,       ptr              // load char
        ADD BTM,    JZEn,    0    ; JNEn init   // BTM
        XOR tmp2,   space,   tmp              // is space?
        ADD BTM,    JZEn,    0    ; JZEn state0 // new state
        XOR tmp2,   plus,    tmp
        ADD BTM,    JZEn,    0    ; JZEn state1
        XOR tmp2,   minus,   tmp
        ADD BTM,    JZEn,    0    ; JZEn state1
        XOR tmp2,   dot,     tmp
        ADD BTM,    JZEn,    0    ; JZEn state2
        ADD tmp2,   m_char0, tmp                    // range chk
        ADD BTM,    JPOn,    0     ; JNEn state7
        SUB tmp2,   ten,     tmp2
        ADD BTM,    JMP,     0     ; JPOn state4
        ADD reject, one,     0                      // reject!
        NOP                        ; JMP  state0 // RAW haz.

state1: ADD BTM,    JNEn,    0
        ADD tmp,    0,       ptr
        ADD BTM,    JZEn,    0     ; JNEn init
        XOR tmp2,   dot,     tmp
        ADD BTM,    JNEn,    0     ; JZEn state2
        ADD tmp2,   m_char0, tmp
        ADD BTM,    JPOn,    0     ; JNEn state7
        SUB tmp2,   ten,     tmp2
        ADD BTM,    JMP,     0     ; JPOn state4
        ADD reject, one,     0
        NOP                        ; JMP  state0

state2: ADD BTM,    JNEn,    0
        ADD tmp,    0,       ptr
        ADD BTM,    JNEn,    0     ; JNEn init
        ADD tmp2,   m_char0, tmp
        ADD BTM,    JPOn,    0     ; JNEn state7
        SUB tmp2,   ten,     tmp2
        ADD BTM,    JMP,     0     ; JPOn state3
        ADD reject, one,     0
        NOP                        ; JMP  state0

state3: ADD BTM,    JNEn,    0
        ADD tmp,    0,       ptr
        ADD BTM,    JZEn,    0     ; JNEn init
        XOR tmp2,   space,   tmp
        ADD BTM,    JNEn,    0     ; JZEn state6
        ADD tmp2,   m_char0, tmp
        ADD BTM,    JPOn,    0     ; JNEn state7
        SUB tmp2,   ten,     tmp2
        ADD BTM,    JMP,     0     ; JPOn state3
        ADD reject, one,     0
        NOP                        ; JMP  state0

state4: ADD BTM,    JNEn,    0
        ADD tmp,    0,       ptr
        ADD BTM,    JZEn,    0     ; JNEn init
        XOR tmp2,   dot,     tmp
```

```
         ADD BTM,    JNEn,    0    ; JZEn state5
         ADD tmp2,   m_char0, tmp
         ADD BTM,    JPOn,    0    ; JNEn state7
         SUB tmp2,   ten,     tmp2
         ADD BTM,    JMP,     0    ; JPOn state4
         ADD reject, one,     0
         NOP                       ; JMP  state0

state5:  ADD BTM,    JNEn,    0
         ADD tmp,    0,       ptr
         ADD BTM,    JZEn,    0    ; JNEn init
         XOR tmp2,   space,   tmp
         ADD BTM,    JNEn,    0    ; JZEn state6
         ADD tmp2,   m_char0, tmp
         ADD BTM,    JPOn,    0    ; JNEn state7
         SUB tmp2,   ten,     tmp2
         ADD BTM,    JMP,     0    ; JPOn state3
         ADD reject, one,     0
         NOP                       ; JMP  state0

state6:  ADD accept, one,     0    ; JMP  state0 // BTM

state7:  ADD reject, one,     0    ; JMP  state0 // BTM
```

The End.

Bibliography

[1] Aasaraai, K., and Moshovos, A. Towards a viable out-of-order soft core: Copy-Free, checkpointed register renaming. *International Conference on Field Programmable Logic and Applications (FPL)* (2009), 79–85.

[2] Aasaraai, K., and Moshovos, A. Design Space Exploration of Instruction Schedulers for Out-of-Order Soft Processors. In *International Conference on Field-Programmable Technology (FPT)* (2010), pp. 385–388.

[3] Altera. Nios II Processor. http://www.altera.com/devices/processor/nios2/ni2-index.html, October 2011. Accessed June 2014.

[4] Altera. Best Practices for Incremental Compilation Partitions and Floorplan Assignments. *Quartus II Handbook Version 13.0 1* (November 2012), 1–54.

[5] Altera. *Stratix IV Device Handbook*, vol. 1. 2012. http://www.altera.com/literature/hb/stratix-iv/stratix4_handbook.pdf.

[6] Altera. Stratix V Device Datasheet. http://www.altera.com/literature/hb/stratix-v/stx5_53001.pdf, November 2013. Version 3.2.

[7] Altera. Arria V Device Datasheet. http://www.altera.com/literature/hb/arria-v/av_51002.pdf, July 2014. Version 3.8.

[8] Altera. Cyclone IV Device Datasheet. http://www.altera.com/literature/hb/cyclone-iv/cyiv-53001.pdf, October 2014. Version 1.9.

[9] ALTERA. Cyclone V Device Datasheet. http://www.altera.com/literature/hb/cyclone-v/cv_51002.pdf, July 2014. Version 3.9.

[10] ALTERA. DC and Switching Characteristics for Stratix IV Devices. http://www.altera.com/literature/hb/stratix-iv/stx4_siv54001.pdf, March 2014. Version 5.8.

[11] ALTERA. Nios II Performance Benchmarks. http://www.altera.com/literature/ds/ds_nios2_perf.pdf, August 2014. Version 11.0.

[12] ANALOG DEVICES. ADSP-TS101 TigerSHARC DSP Programming Reference, 2005.

[13] ANALOG DEVICES. Blackfin Processor Programming Reference, 2008.

[14] ANJAM, F., NADEEM, M., AND WONG, S. A VLIW softcore processor with dynamically adjustable issue-slots. In *International Conference on Field-Programmable Technology (FPT)* (December 2010), pp. 393–398.

[15] ASANOVIC, K., BODIK, R., CATANZARO, B. C., GEBIS, J. J., HUSBANDS, P., KEUTZER, K., PATTERSON, D. A., PLISHKER, W. L., SHALF, J., WILLIAMS, S. W., AND YELICK, K. A. The Landscape of Parallel Computing Research: A View from Berkeley. Tech. Rep. UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006. More Dwarfs described at http://view.eecs.berkeley.edu/wiki/Dwarfs.

[16] BACON, D., RABBAH, R., AND SHUKLA, S. FPGA Programming for the Masses. *ACM Queue 11*, 2 (Feb. 2013), 40:40–40:52.

[17] BEHRENS, D., HARBICH, K., AND BARKE, E. Hierarchical Partitioning. *International Conference on Computer Aided Design (ICCAD)* (1996), 470–477.

[18] Bell, C. G. What Have We Learned from the PDP-11? In *Perspectives on computer science: from the 10th anniversary symposium at the Computer Science Department, Carnegie-Mellon University*, A. Jones and C.-M. U. C. S. Dept, Eds., no. v. 4-6 in ACM monograph series. Academic Press, 1977.

[19] Brant, A., and Lemieux, G. G. ZUMA: An Open FPGA Overlay Architecture. *International Symposium on Field-Programmable Custom Computing Machines (FCCM)* (Apr. 2012), 93–96.

[20] Brunett, S., Thornley, J., and Ellenbecker, M. An initial evaluation of the Tera multithreaded architecture and programming system using the C3I parallel benchmark suite. *ACM/IEEE Conference on Supercomputing (SC'98)* (1998), 1–24.

[21] Canis, A., Brown, S., and Anderson, J. H. Modulo SDC scheduling with recurrence minimization in high-Level synthesis. In *International Conference on Field-Programmable Logic and Applications (FPL)* (September 2014), pp. 1–8.

[22] Canis, A., Choi, J., Aldham, M., Zhang, V., Kammoona, A., Czajkowski, T., Brown, S. D., and Anderson, J. H. LegUp: An Open-source High-level Synthesis Tool for FPGA-based Processor/Accelerator Systems. *ACM Trans. Embed. Comput. Syst. 13*, 2 (Sept. 2013), 24:1–24:27.

[23] Capalija, D., and Abdelrahman, T. A high-performance overlay architecture for pipelined execution of data flow graphs. In *International Conference on Field Programmable Logic and Applications (FPL)* (Sept 2013), pp. 1–8.

[24] Capalija, D., and Abdelrahman, T. S. An Architecture for Exploiting Coarse-Grain Parallelism on FPGAs. In *IEEE International Conference on Field Programmable Technology (FPT)* (2009), pp. 285–291.

[25] CDC. *CDC CYBER 170 COMPUTER SYSTEMS MODELS 720, 730, 750, AND 760 Model 176 (LEVEL B): CPU INSTRUCTION SET.* Control Data Corporation, 1979. See page 2-44 for barrel diagram (pg. 49).

[26] Cheah, H. Y., Fahmy, S., and Maskel, D. L. iDEA: A DSP block based FPGA soft processor. *IEEE International Conference on Field Programmable Technology (FPT)* (2012), 151–158.

[27] Cheah, H. Y., Fahmy, S. A., Maskell, D. L., and Kulkarni, C. A Lean FPGA Soft Processor Built Using a DSP Block. In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)* (2012), pp. 237–240.

[28] Chen, C.-M., Chen, Y.-Y., and King, C.-T. Branch Merging for Effective Exploitation of Instruction-level Parallelism. In *International Symposium on Microarchitecture (MICRO)* (1992), pp. 37–40.

[29] Chen, D., and Singh, D. Line-level incremental resynthesis techniques for FPGAs. *ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)* (2011), 133–142.

[30] Choong, A., Beidas, R., and Zhu, J. Parallelizing Simulated Annealing-Based Placement Using GPGPU. *International Conference on Field Programmable Logic and Applications (FPL)* (Aug. 2010), 31–34.

[31] Chou, C. H., Severance, A., Brant, A. D., Liu, Z., Sant, S., and Lemieux, G. G. VEGAS: soft vector processor with scratchpad memory. In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)* (2011), pp. 15–24.

[32] Chung, E. S., Papamichael, M. K., Nurvitadhi, E., Hoe, J. C., Mai, K., and Falsafi, B. ProtoFlex: Towards Scalable, Full-System Multiprocessor

Simulations Using FPGAs. *ACM Trans. Reconfigurable Technol. Syst. (TRETS) 2*, 2 (June 2009), 15:1–15:32.

[33] Cong, J., and Zou, Y. FPGA-Based Hardware Acceleration of Lithographic Aerial Image Simulation. *ACM Transactions on Reconfigurable Technology and Systems (TRETS) 2*, 3 (2009), 1–29.

[34] Correia, V. P., and Reis, A. I. Classifying n-Input Boolean Functions. In *VII Workshop IBERCHIP* (Montevideo, Uruguay, March 2001).

[35] Dai, Z., and Zhu, J. Saturating the Transceiver Bandwidth : Switch Fabric Design on FPGAs. In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)* (2012), pp. 67–75.

[36] Davidson, J. W., and Whalley, D. B. Reducing the Cost of Branches by Using Registers. In *International Symposium on Computer Architecture (ISCA)* (1990), pp. 182–191.

[37] Dehkordi, M. E., Brown, S. D., and Borer, T. P. Modular Partitioning for Incremental Compilation. *International Conference on Field Programmable Logic and Applications (FPL)* (2006), 1–6.

[38] Dehon, A. The Density Advantage of Configurable Computing. *IEEE Computer* (April 2000), 41–49.

[39] Denelcor. *HEP Principles of Operation.* Denver, Colorado, 1981.

[40] Dijkstra, E. W. The Humble Programmer. *Commun. ACM 15*, 10 (Oct. 1972), 859–866. 1972 Turing Award Lecture.

[41] Dimond, R., Mencer, O., and Luk, W. CUSTARD - A Customisable Threaded FPGA Soft Processor and Tools. In *International Conference on Field Programmable Logic (FPL)* (August 2005), pp. 1–6.

[42] DITZEL, D. R., AND MCLELLAN, H. R. Branch Folding in the CRISP Microprocessor: Reducing Branch Delay to Zero. In *International Symposium on Computer Architecture (ISCA)* (1987), pp. 2–8.

[43] EHLIAR, A., KARLSTRÖM, P., AND LIU, D. A High Performance Microprocessor with DSP Extensions Optimized for the Virtex-4 FPGA. In *International Conference on Field Programmable Logic and Applications (FPL)* (2008), pp. 599–602.

[44] FANG, W.-J., AND WU, A. C.-H. Performance-driven multi-FPGA partitioning using functional clustering and replication. *Proceedings of the 35th IEEE/ACM Design Automation Conference (DAC)* (1998), 283–286.

[45] FANG, W.-J., AND WU, A. C.-H. Multiway FPGA partitioning by fully exploiting design hierarchy. *ACM Transactions on Design Automation of Electronic Systems (TODAES) 5*, 1 (Jan. 2000), 34–50.

[46] FORT, B., CANIS, A., CHOI, J., CALAGAR, N., LIAN, R., HADJIS, S., CHEN, Y. T., HALL, M., SYROWIK, B., CZAJKOWSKI, T., BROWN, S., AND ANDERSON, J. H. Automating the Design of Processor/Accelerator Embedded Systems with LegUp High-Level Synthesis. In *IEEE International Conference on Embedded and Ubiquitous Computing (EUC)* (August 2014).

[47] FORT, B., CAPALIJA, D., VRANESIC, Z., AND BROWN, S. A Multithreaded Soft Processor for SoPC Area Reduction. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)* (April 2006), pp. 131–142.

[48] GONZALEZ, A., AND LLABERIA, J. Reducing branch delay to zero in pipelined processors. *IEEE Transactions on Computers 42*, 3 (Mar 1993), 363–371.

[49] GRAY, J. Designing a Simple FPGA-Optimized RISC CPU and System-on-a-Chip. http://www.fpgacpu.org/papers/soc-gr0040-paper.pdf, 2000. Accessed December 2011.

[50] Grotker, T. *System Design with SystemC.* Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[51] Guo, Z., Najjar, W., Vahid, F., and Vissers, K. A quantitative analysis of the speedup factors of FPGAs over processors. *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)* (2004), 162–170.

[52] Hameed, R., Qadeer, W., Wachs, M., Azizi, O., Solomatnikov, A., Lee, B. C., Richardson, S., Kozyrakis, C., and Horowitz, M. Understanding sources of inefficiency in general-purpose chips. In *International Symposium on Computer Architecture (ISCA)* (2010), pp. 37–47.

[53] Hauck, S., and DeHon, A. *Reconfigurable computing: the theory and practice of FPGA-based computation.* Morgan Kaufmann, 2008.

[54] Henning, B. Large memory model for Propeller assembly language programs. http://forums.parallax.com/showthread.php/89640-ANNOUNCING-Large-memory-model-for-Propeller-assembly-language-programs!, November 2006. Accessed June 2014.

[55] Herbordt, M. C., Gu, Y., Vancourt, T., Model, J., Sukhwani, B., and Chiu, M. Computing Models for FPGA-Based Accelerators. *Computing in Science & Engineering 10*, 6 (Oct. 2008), 35–45.

[56] Herzen, B. V. Signal processing at 250 MHz using high-performance FPGA's. *IEEE Transactions on Very Large Scale Integration Systems (TVLSI) 6*, 2 (1998), 238–246. Also published at FPGA 1997, and in the FPGA20 compilation.

[57] Hilewitz, Y. *Advanced Bit Manipulation Instructions: Architecture, Implementation and Applications.* Phd thesis, Princeton, 2008.

[58] IEEE. *Verilog IEEE Std 1364-2001-E*, vol. 4. IEEE, 2004.

[59] ITRS. International Roadmap For Semiconductors: Design. http://www.itrs.net/Links/2011itrs/2011Chapters/2011Design.pdf, 2011.

[60] JONES, A. K., HOARE, R., KUSIC, D., FAZEKAS, J., AND FOSTER, J. An FPGA-based VLIW processor with custom hardware execution. In *International Symposium on Field-Programmable Gate Arrays (FPGA)* (2005), pp. 107–117.

[61] JORDAN, H. Experience with pipelined multiple instruction streams. *Proceedings of the IEEE 72*, 1 (1984), 113–123.

[62] KATEVENIS, M. G. H. *Reduced Instruction Set: Computer Architectures for VLSI.* ACM doctoral dissertation awards. MIT Press, 1985.

[63] KELLEY, E. C. *The Workshop Way of Learning.* Harper & Brother, New York, 1951. Page 2.

[64] KIM, I., AND LIPASTI, M. Half-price architecture. In *International Symposium on Computer Architecture (ISCA)* (June 2003), ISCA, pp. 28–38.

[65] KINDRATENKO, V. V., BRUNNER, R. J., AND MYERS, A. D. Mitrion-C Application Development on SGI Altix 350/RC100. *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)* (Apr. 2007), 239–250.

[66] KNIESER, M. J., AND PAPACHRISTOU, C. A. Y-Pipe: A Conditional Branching Scheme Without Pipeline Delays. In *International Symposium on Microarchitecture (MICRO)* (1992), pp. 125–128.

[67] KNUTH, D. E. *The Art of Computer Programming, Volume 1 (3rd ed.): Fundamental Algorithms.* Addison-Wesley, 1997.

[68] KOCH, D., BECKHOFF, C., AND LEMIEUX, G. An efficient FPGA overlay for portable custom instruction set extensions. In *International Conference on Field Programmable Logic and Applications (FPL)* (Sept 2013), pp. 1–8.

[69] Krasnov, A., and Schultz, A. RAMP Blue: A message-passing manycore system in FPGAs. *International Conference on Field Programmable Logic and Applications (FPL)* (2007), 54–61.

[70] Kuon, I., and Rose, J. Measuring the gap between FPGAs and ASICs. *International Symposium on Field Programmable Gate Arrays (FPGA)* (2006), 21–30.

[71] Labrecque, M., and Steffan, J. Improving Pipelined Soft Processors with Multithreading. In *International Conference on Field-Programmable Logic and Applications (FPL)* (Aug 2007), pp. 210–215.

[72] Labrecque, M., and Steffan, J. G. Fast Critical Sections via Thread Scheduling for FPGA-based Multithreaded Processors. In *International Conference on Field Programmable Logic and Applications (FPL)* (August 2009), pp. 18–25.

[73] Labrecque, M., Yiannacouras, P., and Steffan, J. G. Scaling Soft Processor Systems. In *International Symposium on Field-Programmable Custom Computing Machines (FCCM)* (2008), pp. 195–205.

[74] LaForest, C. E., Anderson, J., and Steffan, J. G. Approaching Overhead-Free Execution on FPGA Soft-Processors. In *International Conference on Field-Programmable Technology (FPT)* (Shanghai, China, Dec 2014).

[75] Laforest, C. E., Li, Z., O'rourke, T., Liu, M. G., and Steffan, J. G. Composing Multi-Ported Memories on FPGAs. *ACM Transactions on Reconfigurable Technology and Systems (TRETS) 7*, 3 (Sept. 2014), 16:1–16:23.

[76] LaForest, C. E., Liu, M. G., Rapati, E. R., and Steffan, J. G. Multi-ported memories for FPGAs via XOR. In *ACM/SIGDA international symposium on Field Programmable Gate Arrays (FPGA)* (Feb. 2012), pp. 209–218.

[77] LaForest, C. E., and Steffan, J. G. OCTAVO: An FPGA-Centric Processor Family. In *ACM/SIGDA international symposium on Field Programmable Gate Arrays (FPGA)* (Feb. 2012), pp. 219–228.

[78] LaForest, C. E., and Steffan, J. G. Maximizing speed and density of tiled FPGA overlays via partitioning. In *International Conference on Field-Programmable Technology (FPT)* (Dec 2013), pp. 238–245.

[79] Lebedev, I., Cheng, S., Doupnik, A., Martin, J., Fletcher, C., Burke, D., Lin, M., and Wawrzynek, J. MARC: A Many-Core Approach to Reconfigurable Computing. *International Conference on Reconfigurable Computing and FPGAs (ReConFig)* (Dec. 2010), 7–12.

[80] Lee, L. H., Scott, J., Moyer, B., and Arends, J. Low-cost branch folding for embedded applications with small tight loops. In *International Symposium on Microarchitecture (MICRO)* (November 1999), pp. 103–111.

[81] Lewis, D., Ahmed, E., Cashman, D., Vanderhoek, T., Lane, C., Lee, A., and Pan, P. Architectural enhancements in Stratix-III and Stratix-IV. *ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)* (2009), 33–42.

[82] Lewis, D., Betz, V., Jefferson, D., and Lee, A. The Stratix Routing and Logic Architecture. *ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)* (2003), 12–20.

[83] Loken, C., Gruner, D., Groer, L., Peltier, R., Bunn, N., Craig, M., Henriques, T., Dempsey, J., Yu, C.-H., Chen, J., Dursi, L. J., Chong, J., Northrup, S., Pinto, J., Knecht, N., and Zon, R. V. SciNet: Lessons Learned from Building a Power-efficient Top-20 System and Data Centre. *Journal of Physics: Conference Series 256* (Nov. 2010), 12–26.

[84] Ludwin, A., and Betz, V. Efficient and Deterministic Parallel Placement for FPGAs. *ACM Transactions on Design Automation of Electronic Systems (TODAES) 16*, 3 (June 2011), 1–23.

[85] Ly, D. L., and Chow, P. A high-performance FPGA architecture for Restricted Boltzmann Machines. In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)* (2009), pp. 73–82.

[86] Matthews, E., Shannon, L., and Fedorova, A. POLYBLAZE: FROM ONE TO MANY. BRINGING THE MICROBLAZE INTO THE MULTICORE ERA WITH LINUX SMP SUPPORT. *International Conference on Field Programmable Logic (FPL)* (2012), 224–230.

[87] McFarling, S., and Hennesey, J. Reducing the Cost of Branches. In *International Symposium on Computer Architecture (ISCA)* (1986), pp. 396–403.

[88] Moon, S.-M., Carson, S. D., and Agrawala, A. K. Hardware Implementation of a General Multi-way Jump Mechanism. In *Workshop and Symposium on Microprogramming and Microarchitecture (MICRO)* (1990), pp. 38–45.

[89] Moore, S., and Chadwick, G. The Tiger "MIPS" Processor Home Page. `http://www.cl.cam.ac.uk/teaching/0910/ECAD+Arch/mips.html`, 2009. ECAD and Architecture Practical Classes, University of Cambridge.

[90] Moussali, R., Ghanem, N., and Saghir, M. Microarchitectural Enhancements for Configurable Multi-Threaded Soft Processors. In *International Conference on Field Programmable Logic and Applications (FPL)* (Aug. 2007), pp. 782–785.

[91] Moussali, R., Ghanem, N., and Saghir, M. A. R. Supporting multithreading in configurable soft processor cores. In *International conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)* (2007), pp. 155–159.

[92] Murray, K., Whitty, S., Liu, S., Luu, J., and Betz, V. Titan: Enabling large and complex benchmarks in academic CAD. In *International Conference on Field Programmable Logic and Applications (FPL)* (Sept 2013), pp. 1–8.

[93] Murray, K. E., and Betz, V. Quantifying the Cost and Benefit of Latency Insensitive Communication on FPGAs. In *ACM/SIGDA International Symposium on Field-programmable Gate Arrays (FPGA)* (2014), pp. 223–232.

[94] Myer, T. H., and Sutherland, I. E. On the design of display processors. *Communications of the ACM 11*, 6 (June 1968), 410–414.

[95] Nikhil, R. S., and Arvind. What is Bluespec? *SIGDA Newsl. 39*, 1 (Jan. 2009), 1–1.

[96] NVIDIA. CUDA Binary Utilities Documentation. http://docs.nvidia.com/cuda/cuda-binary-utilities/index.html#maxwell, 2014. Section 4.4: Maxwell Instruction Set.

[97] Ovtcharov, K., Tili, I., and Steffan, J. TILT: A multithreaded VLIW soft processor family. In *International Conference on Field Programmable Logic and Applications (FPL)* (Sept 2013), pp. 1–4.

[98] Price, C. *MIPS IV instruction set.* MIPS Technologies, Mountain View, California, 1995.

[99] Rojas, R. How to Make Zuse's Z3 a Universal Computer. *IEEE Annals of the History of Computing 20*, 3 (July 1998), 51–54.

[100] Rosière, M., Desbarbieux, J.-l., Drach, N., and Wajsbürt, F. An Out-of-Order Superscalar Processor on FPGA: The ReOrder Buffer Design. *Design, Automation, and Test in Europe (DATE)* (2012), 1549–1554.

[101] Saghir, M. A. R., El-Majzoub, M., and Akl, P. Datapath and ISA Customization for Soft VLIW Processors. In *IEEE International Conference on Reconfigurable Computing and FPGAs (ReConfig)* (Sept. 2006), pp. 1–10.

[102] Schoeberl, M. Leros: A Tiny Microcontroller for FPGAs. In *International Conference on Field Programmable Logic and Applications (FPL)* (Sept. 2011), pp. 10–14.

[103] Scollo, G. Looking for Class Records in the 3x + 1 Problem by means of the COMETA Grid Infrastructure. In *Grid Open Days at the University of Palermo* (Palermo, Italy, December 2007), pp. 6–7.

[104] Scoville, R. Register Duplication for Timing Closure. Tech. rep., Altera, 2011. http://www.alterawiki.com/uploads/a/aa/Register_Duplication_for_Timing_Closure.pdf.

[105] Severance, A., Edwards, J., Omidian, H., and Lemieux, G. Soft Vector Processors with Streaming Pipelines. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)* (2014), pp. 117–126.

[106] Severance, A., and Lemieux, G. VENICE: A compact vector processor for FPGA applications. *IEEE International Conference on Field Programmable Technology (FPT)* (Apr. 2012), 261–268.

[107] Severance, A., and Lemieux, G. Embedded supercomputing in FPGAs with the VectorBlox MXP Matrix Processor. In *International Conference on Hardware/-Software Codesign and System Synthesis (CODES+ISSS)* (Sept 2013), pp. 1–10.

[108] Severance, A., and Lemieux, G. TputCache: High-frequency, multi-way cache for high-throughput FPGA applications. *International Conference on Field Programmable Logic (FPL)* (2013), 1–6.

[109] SINGH, D. P., CZAJKOWSKI, T. S., AND LING, A. Harnessing the Power of FPGAs Using Altera's OpenCL Compiler. In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)* (2013), pp. 5–6.

[110] SIROWY, S., AND FORIN, A. Where's the beef? Why FPGAs are so fast. Tech. rep., Microsoft Research, Sept 2008. MSR-TR-2008-130.

[111] SNAVELY, A., CARTER, L., AND BOISSEAU, J. Multi-processor Performance on the Tera MTA. *ACM/IEEE Conference on Supercomputing (SC'98)* (1998), 1–8.

[112] STATISTICS CANADA. 2011 Census. Data requested by phone.

[113] TERRAS, R. A stopping time problem on the positive integers. *Acta Arithmetica XXX* (1976), 241–252.

[114] TEXAS INSTRUMENTS. *TMS320C64x/C64x+ DSP CPU and Instruction Set.* No. July. 2010. No. SPRU732J.

[115] THORNTON, J. E. The CDC 6600 Project. *IEEE Annals of the History of Computing 2*, 4 (Oct. 1980), 338–348.

[116] TSOI, K. H., AND LUK, W. Axel : A Heterogeneous Cluster with FPGAs and GPUs. *ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)* (2010), 115–124.

[117] TUMBUSH, G. Signed Arithmetic in Verilog 2001 - Opportunities and Hazards, 2001. http://www.uccs.edu/~gtumbush/published_papers/TumbushDVCon05.pdf.

[118] UNITED STATES BUREAU OF LABOR STATISTICS. *Occupational Outlook Handbook.* 2012.

[119] Unnikrishnan, D., Zhao, J., and Tessier, R. Application Specific Customization and Scalability of Soft Multiprocessors. *17th IEEE Symposium on Field Programmable Custom Computing Machines (FCCM)* (Apr. 2009), 123–130.

[120] Vahid, F., Le, T. D., and Hsu, Y.-C. A comparison of functional and structural partitioning. *9th International Symposium on System Synthesis (ISSS)* (1996), 121–126.

[121] Vahid, F., Le, T. D., and Hsu, Y.-C. Functional partitioning improvements over structural partitioning for packaging constraints and synthesis: tool performance. *ACM Transactions on Design Automation of Electronic Systems (TODAES) 3*, 2 (Apr. 1998), 181–208.

[122] Wang, S., and Provence, J. Branch-skipped pipelined microprocessor. *Electronics Letters 30*, 14 (Jul 1994), 1122–1123.

[123] Warren, H. S. *Hacker's Delight*, 2nd ed. Addison-Wesley Professional, 2012.

[124] Wikipedia. Collatz conjecture — wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Collatz_conjecture&oldid=621054097, 2014. [Online; accessed 21-August-2014].

[125] Wong, H., Betz, V., and Rose, J. Comparing FPGA vs. custom CMOS and the impact on processor microarchitecture. In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)* (2011), pp. 5–14.

[126] Wong, H., Betz, V., and Rose, J. Quantifying the Gap Between FPGA and Custom CMOS to Aid Microarchitectural Design. *IEEE Transactions on Very Large Scale Integration Systems (TVLSI) 22*, 10 (Oct 2014), 2067–2080.

[127] WU, Q., AND MCELVAIN, K. S. A fast discrete placement algorithm for FPGAs. *ACM/SIGDA international symposium on Field Programmable Gate Arrays (FPGA)* (2012), 115–118.

[128] XILINX. MicroBlaze Soft Processor. http://www.xilinx.com/microblaze, October 2011. Accessed December 2011.

[129] YIANNACOURAS, P., STEFFAN, J., AND ROSE, J. Data parallel FPGA workloads: Software versus hardware. In *International Conference on Field-Programmable Logic and Applications (FPL)* (Aug 2009), pp. 51–58.

[130] YIANNACOURAS, P., STEFFAN, J. G., AND ROSE, J. VESPA: portable, scalable, and flexible FPGA-based vector processors. In *International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)* (2008), pp. 61–70.

[131] YIANNACOURAS, P., STEFFAN, J. G., AND ROSE, J. Fine-grain performance scaling of soft vector processors. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)* (2009), pp. 97–106.

[132] YIANNACOURAS, P., STEFFAN, J. G., AND ROSE, J. Portable, Flexible, and Scalable Soft Vector Processors. *IEEE Trans. Very Large Scale Integr. Syst. (VLSI) 20*, 8 (Aug. 2012), 1429–1442.

[133] YU, J., EAGLESTON, C., CHOU, C. H.-Y., PERREAULT, M., AND LEMIEUX, G. Vector Processing as a Soft Processor Accelerator. *ACM Trans. Reconfigurable Technol. Syst. (TRETS) 2* (June 2009), 12:1–12:34.

[134] YU, J., AND LEMIEUX, G. A Case for Soft Vector Processors in FPGAs. In *IEEE International Conference on Field Programmable Technology (FPT)* (2007), pp. 1–4.

[135] Yu, J., Lemieux, G., and Eagleston, C. Vector processing as a soft-core CPU accelerator. *ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)* (2008), 222–232.

[136] Zeh, C. Incremental Design Reuse with Partitions. *Xilinx XAPP918 (v1.0) 918* (2007), 1–17.

[137] Zhang, W., Betz, V., and Rose, J. Portable and Scalable FPGA-Based Acceleration of a Direct Linear System Solver. *ACM Transactions on Reconfigurable Technology and Systems (TRETS) 5*, 1 (Mar. 2012), 1–26.