Octavo: an FPGA-Centric Processor Family

Charles Eric LaForest and J. Gregory Steffan
Department of Electrical and Computer Engineering
University of Toronto, Canada
{laforest,steffan}@eecg.toronto.edu

ABSTRACT

Overlay processor architectures allow FPGAs to be programmed by non-experts using software, but prior designs have mainly been based on the architecture of their ASIC predecessors. In this paper we develop a new processor architecture that from the beginning accounts for and exploits the predefined widths, depths, maximum operating frequencies, and other discretizations and limits of the underlying FPGA components. The result is Octavo, a ten-pipeline-stage eight-threaded processor that operates at the block RAM maximum of 550MHz on a Stratix IV FPGA. Octavo is highly parameterized, allowing us to explore trade-offs in datapath and memory width, memory depth, and number of supported thread contexts.

Categories and Subject Descriptors

C.1.3 [**Processor Architecture**]: Other Architecture Styles—*Adaptable Architectures*; C.4 [**Performance of Systems**]: Measurement Techniques, Design Studies

General Terms

Design Performance Measurement

Keywords

FPGA, soft processor, multithreading, microarchitecture

1. INTRODUCTION

Making FPGAs easier to program for non-experts is a challenge of increasing interest and importance. One approach is to enable FPGAs to be programmed using software via overlay architectures, for example conventional soft processors such as Altera's NIOS and Xilinx's Microblaze, or more aggressive designs such as soft vector processors [5, 18, 19]. Prior soft processor designs have mainly inherited the architecture of their ASIC-based predecessors with some optimization to better fit the underlying FPGA. However, FPGAs provide a much different substrate than raw transistors, including lookup tables (LUTs), block RAMs (BRAMs), multipliers/DSPs, and various routing resources—all of which have predefined widths, depths, maximum operating frequencies, and other discretizations and limits [1]. The existence of these artifacts and their characteristics suggests that an FPGA-centric processor architecture, one that is built from the "ground-up" with FPGA capabilities in mind, will differ from a conventional architecture in compelling ways, mainly by using the FPGA resources more efficiently.

FPGA'12, February 22–24, 2012, Monterey, California, USA.

1.1 How do FPGAs Want to Compute?

In this work we ask the fundamental question: *How do FPGAs want to compute?* A more exact (but less memorable) phrasing of this question is: *What processor architecture best fits the underlying structures and discretizations of an FPGA?* This question alone is still too broad for the scope of a single research paper, so we narrow our investigation by striving for the following goals for a processor design.

1. To support a highly-threaded data-parallel programming model, similar to OpenCL.
2. To run at the maximum operating frequency allowed by the particular FPGA resources used (e.g.: BRAMs).
3. To have high performance—i.e, not only high-frequency but also reasonable instruction count and processor-cycles-per-instruction.
4. To never stall due to hazards (such as control or data dependences).
5. To strive for simplicity and minimalism, rather than inherit all of the features of an existing processor design/ISA.
6. To match underlying FPGA structures; for example, to discover the most effective width for data elements for both datapaths and storage, as opposed to defaulting to the conventional 32-bit width.

This paper describes the process and results of developing an FPGA-based processor while striving for these goals.

1.2 Octavo

As a starting point, we show the simplest processor design we could imagine in Figure 1, which is composed of at least one multi-ported memory connected to an ALU, supplying its operands and control and receiving its results. We argue that separate data cache and register file storage is unnecessary: on an FPGA both are inevitably implemented using the same BRAMs. We eliminate separate 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.* [5]). We demonstrate that this single logical memory eliminates the need for immediate operands and

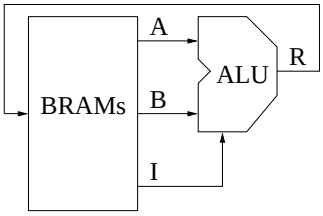


Figure 1: An overview of the architecture of Octavo, composed of a Memory ($BRAMs$) providing operands (A and B) and instructions (I) to an ALU which writes back its results (R) to the same Memory.

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 multithread the processor such that an instruction from a different thread resides in each pipeline stage [7, 8, 12, 14, 15], so that all stages are independent with no control or data hazards or result forwarding between them.

We name our processor architecture *Octavo*[1], 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.

1.3 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 achieved by Octavo. Examples include soft uniprocessors [3, 17], multithreaded soft processors [6–8, 12, 14, 15], soft VLIW processors [4, 10, 16], and soft vector processors [5, 18, 19]. Jan Gray has studied the optimization of processors specifically for FPGAs [9], where synthesis and technology mapping tricks are applied to all aspects of the design of a processor from the instruction set to the architecture.

1.4 Contributions

In future work we plan to extend Octavo to support SIMD/vector datapaths, multicore interconnection, connection to an OpenCL framework (for its abundance of thread and data parallelism), and evaluation of full applications. In this paper we focus on the architecture of a single Octavo core and provide the following four contributions:

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

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

2. 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 I/O optimization. Shift registers expand single-pin inputs, while registered AND-reducers compact word-wide signals to a single output pin.

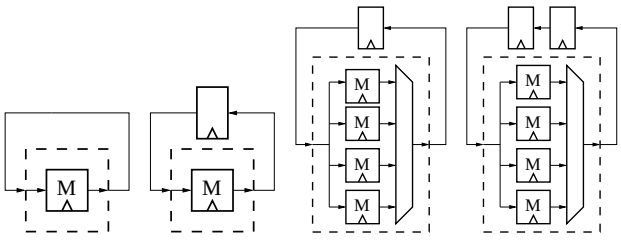
Synthesis We use Altera's Quartus 10.1 to target a Stratix IV `EP4SE230F29C2` FPGA device of the highest available speed grade. For maximum portability, we implement the design in generic Verilog-2001, with some LPM[2] components. We configure the synthesis process to favor speed over area and enable 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 test harness—we constrain a circuit to its own logical design partition and restrict its placement to within a single rectangular area (*LogicLock* area) containing only the circuit under test, excluding the test harness. Any test harness circuitry remains spatially and logically separate from the actual circuit under test.

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, a slower derived clock would aim towards an unnecessarily high target frequency, causing competition 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 from the worst-case F_{max} reports over the range of die temperatures between 0 to 85° at 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 Area does not vary significantly between place and route runs, so we report the first computed result. 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.

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



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

Figure 2: Self-loop characterization of memories reveal that different numbers of pipeline stages absorb the propagation delays depending on their internal configurations. Each of (a)-(d) lists the theoretical maximum frequency of the design, although the BRAM limit of 550MHz is the true limit.

3. 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 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 2 shows four 32-bit-wide memory configurations: 256-word memories using one BRAM with one (2(a)) and two (2(b)) pipeline stages, and 1024-word memories using four BRAMs with two (2(c)) and three (2(d)) pipeline stages. The result for a single BRAM (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. However, two pipeline stages (2(b)) 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 (2(c)) requires 4 BRAMs, additional routing, and some multiplexing logic—and reduces F_{max} to 531MHz. Adding a third pipeline stage (2(d)) absorbs the additional delay and increases F_{max} to 710MHz.

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 [5] 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.

4. 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 9. Despite its frugality, we believe that the Octavo ISA can emulate the MIPS ISA.

Table 1: Octavo's Instruction Word Format.

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

Table 2: Octavo's Instruction Set and Opcode Encoding.

Mnemonic	Opcode	Action
Logic Unit		
XOR	0000	D = A XOR B
AND	0001	D = A AND B
OR	0010	D = A OR B
SRL	0011	D = A » 1 (zero ext.)
SRA	0100	D = A » 1 (sign ext.)
ADD	0101	D = A + B
SUB	0110	D = A - B
—	0111	*(Unused, for expansion)*
Multiplier		
MLO	1000	D = A * B (Lower Word)
MHI	1001	D = A * B (Upper Word)
Controller		
JMP	1010	PC = D
JZE	1011	if (A == 0) PC = D
JNZ	1100	if (A != 0) PC = D
JPO	1101	if (A >= 0) PC = D
JNE	1110	if (A < 0) PC = D
—	1111	*(Unused, for expansion)*

Table 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 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. Table 2 shows Octavo's instruction set and opcode encoding, with ten opcodes allocated to ALU instructions and the remaining six allocated to control instructions. The Logic Unit opcodes are chosen carefully so that they can be broken into sub-opcodes to minimize decoding in the ALU implementation.

5. 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

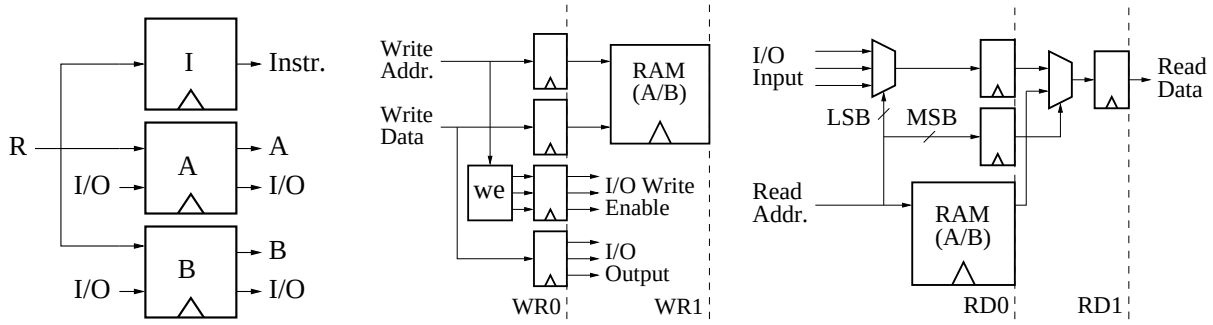


Figure 3: The overall connections of Octavo's memories and the implementation of the A and B Memories with integrated memory-mapped word-wide I/O ports. The RAM component is implemented using BRAMs. Note that both A/B read and writes complete in two cycles, but overlap only for one cycle at RD0/WR1. The I Memory has no I/O and thus reads and writes in a single cycle.

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. Similarly, having I/O in instruction memory could enable the PC to point to I/O to execute an external stream of instructions sent from another CPU (although we do not yet support this feature).

Implementation Figure 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. 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(b)) spends its first cycle registering the address and data to RAM, activating one of the I/O write port write-enable lines based on the write address, and registering the write data to all I/O write ports. The data write to the RAM occurs during the second cycle.[3] A read (Figure 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. 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 [4].

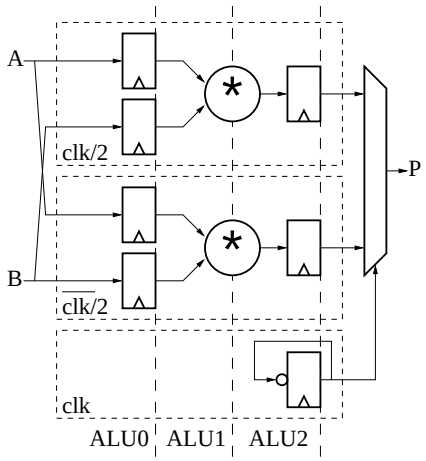


Figure 4: A detailed view of the Multiplier unit, which overcomes the minimum clock pulse width limit of a single multiplier by operating two word-wide multipliers on alternate edges of a half-rate clock $clk/2$, with the correct double-word product P selected by a single state bit driven by the system clock clk.

Figure 4 shows the internal structure of Octavo's Multiplier and our solution to the clocking limitation: we use two word-wide DSP block multipliers[5] in alternation on a synchronous half-rate clock

[3] 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 [1]. 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.

[4] For widths $\leq$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 [1] for high-frequency implementations.

[5] 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.

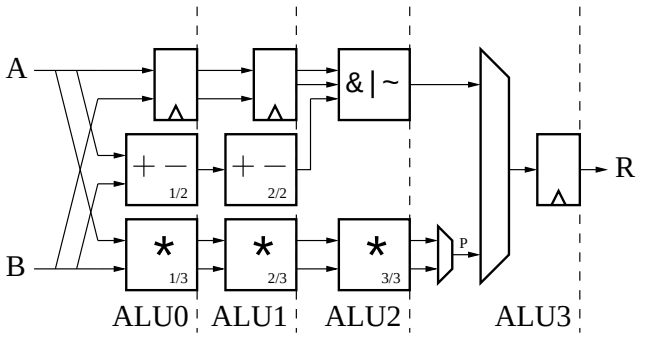


Figure 5: Organization of Octavo's ALU, containing an Adder/Subtractor (+−), a Logic Unit (& | ~), and a Multiplier (⋆). The Logic Unit also multiplexes between its results and those of the Adder/Subtractor.

($clk/2$), 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 and Logic Unit 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. 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.

The Logic Unit (`& | ~`) performs bit-wise `XOR`, `AND`, `OR`, `SRL`, and `SRA` operations (Table 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 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, totaling 6 bits and naturally mapping to a single Stratix IV 6-LUT per output bit.

Combined ALU Design Figure 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.

7. CONTROLLER

Figure 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[6] instead of a BRAM, given a typically narrow PC (< 20 bits) and a relatively small number of threads (8 to 16)—this also helps improve the resource-diversity of Octavo and will ease its 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.

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

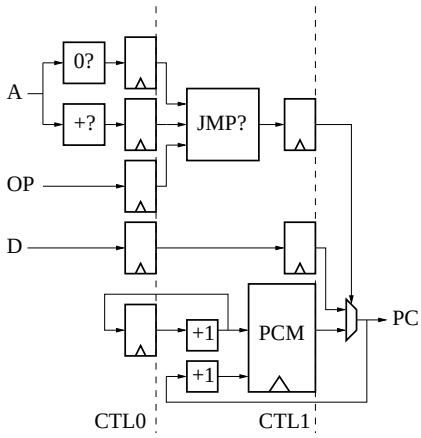


Figure 6: The Controller, which provides the Program Counter (PC) value for each thread of execution and implements flow-control. A Program Counter Memory (PCM) holds the next value of the PC for each thread of execution. Based on the opcode (OP) and the fetched value of operand A (if applicable), the controller may update the PC of a thread with the target address stored in the destination operand D.

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.

8. 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 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, 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

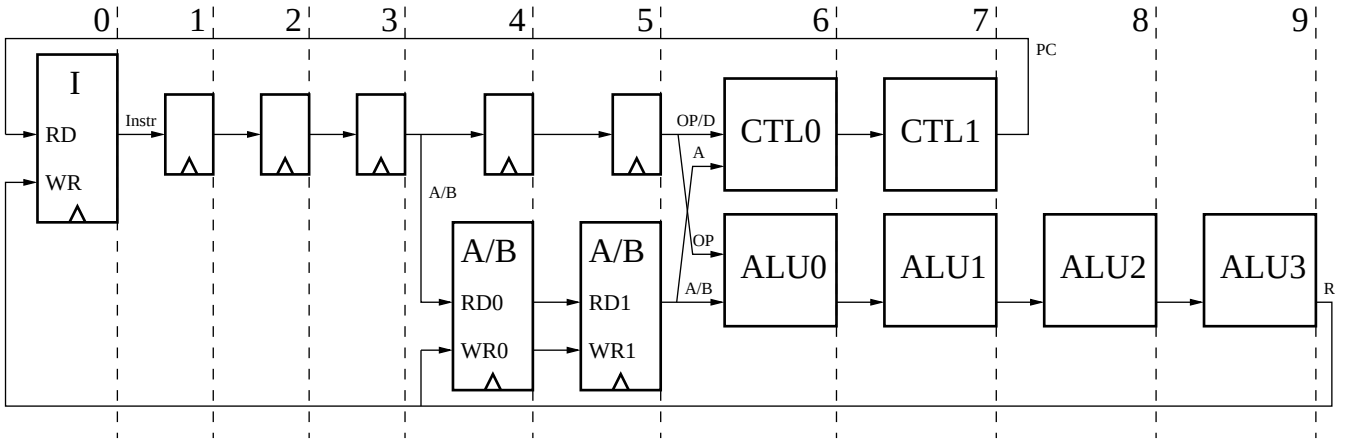


Figure 7: The complete Octavo system: an instruction Memory (I), two data Memories (A and B), an ALU, and a Controller (CTL).

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 4 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.

9. OCTAVO SOFTWARE

As described in Section 4, 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.

9.1 Pointer Dereference

The C code in Figure 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 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

```
1 int a = 42;
2 int c = 88;
3 int *b = &c;
4 ...
5 a = *b;
```

(a) `C` code

```
1 a: 42
2 c: 88
3 b: c
4     ...
5 la   r1, a
6 la   r2, c
7     ...
8 lw    r3, 0(r2)
9 sw    r3, 0(r1)
```

(b) Optimized MIPS code

```
1 Z: 0
2 a: 42
3 c: 88
4 b: c
5 ...
6    or  T, T, b
7    nop
8 T: add a, Z, 0
```

(c) Octavo code (pre-execution)

```
1 Z: 0
2 a: 88
3 b: c
4 c: 88
5 ...
6    or  T, T, b
7    nop
8 T: add a, Z, c
```

(d) Octavo code (post-execution)

Figure 8: Pointer dereference example.

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 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 8) if executing less than 10 threads. Figure 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

```
1  int A[] = { 42, ...};
2  int B[] = { 23, ...};
3  int C[] = { 88, ...};
4  ...
5  *A = *B + *C;
6  A++;
7  B++;
8  C++;
```

(a) C code

```
1  A : 42
2  A': ...
3  B : 23
4  B': ...
5  C : 88
6  C': ...
7       ...
8  la   r1, A
9  la   r2, B
10 la   r3, C
11      ...
12 lw   r5, 0(r2)
13 lw   r6, 0(r3)
14 add  r4, r5, r6
15 sw   r4, 0(r1)
16 addi r1, r1, 1
17 addi r2, r2, 1
18 addi r3, r3, 1
```

(b) MIPS code

```
1  A : 42
2  A': ...
3  B : 23
4  B': ...
5  C : 88
6  C': ...
7  I :   0 1, 1, 1
8       ...
9  T : add A, B, C
10    add T, T, I
```

(c) Octavo code (pre-execution)

```
1  A : 111
2  A': ...
3  B : 23
4  B': ...
5  C : 88
6  C': ...
7  I :   0 1, 1, 1
8       ...
9  T : add A', B', C'
10    add T, T, I
```

(d) Octavo code (post-execution)

Figure 9: Array access example.

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.

9.2 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 9(a) describes the core of a loop summing two arrays. Figure 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 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 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 9(d). Compared to the MIPS code in Figure 9(b), Octavo requires only two instructions instead of seven to compute the same loop body.

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.

```
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 10: Call/return example, in Octavo code.

9.3 Synthesizing Subroutine Calls

Without call stack hardware support, Octavo must synthesize code to implement subroutine linkage using a method previously described by Knuth [11]. 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 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 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.

This subroutine linkage scheme does not allow re-entrancy: threads cannot intersect in the call graph, including with themselves (i.e., recursive calls must be converted to iterative ones). The compiler must create private copies of the subroutine in such cases.

10. 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 1.1) over a wide range of configurations.

10.1 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 11 shows the maximum operating frequency F_{max} of Octavo for word widths ranging from 8 to 72 bits, and for Octavo

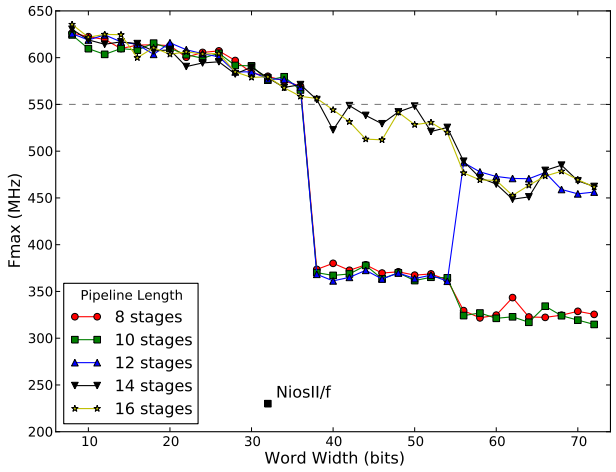


Figure 11: Maximum operating frequency F_{max} vs Octavo word widths ranging from 8 to 72 bits, for Octavo instances with 8 to 16 pipeline stages. In all cases, we limit the memory depth to 256 words to avoid any effect on F_{max}.

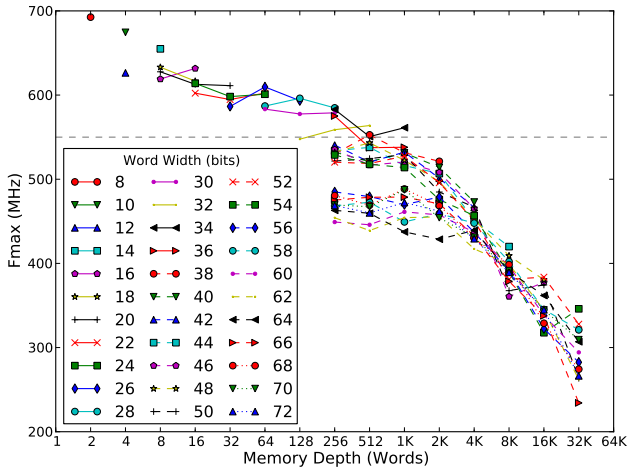


Figure 12: Maximum operating frequency (F_{max}) for a *16-stage* Octavo design over addressable memory depths ranging from 2 to 32,768 words, for word widths from 8 to 72 bits.

instances with 8 to 16 pipeline stages. The 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 [2]. 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 behavior 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 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 11 for Octavo instances with widths of 56 to 72 bits is especially 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 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 greater than 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.

10.2 Area Usage

Our next experiments tests if Octavo's area scales practically as word width and memory depth increase. Figure 13 shows the area used in ALUTs, 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 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 [2].

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

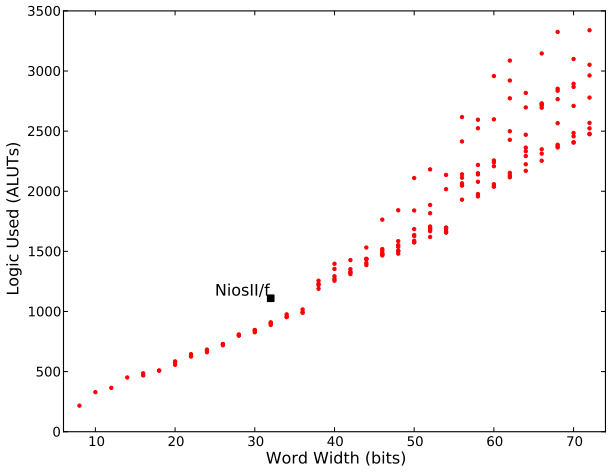


Figure 13: Area (ALUTs, excluding BRAMs and DSP blocks) vs word widths ranging from 8 to 72 bits, for an 8-stage Octavo design. For each width we plot the points representing the possible addressable memory depths, maximally ranging from 2 to 32,768 words.

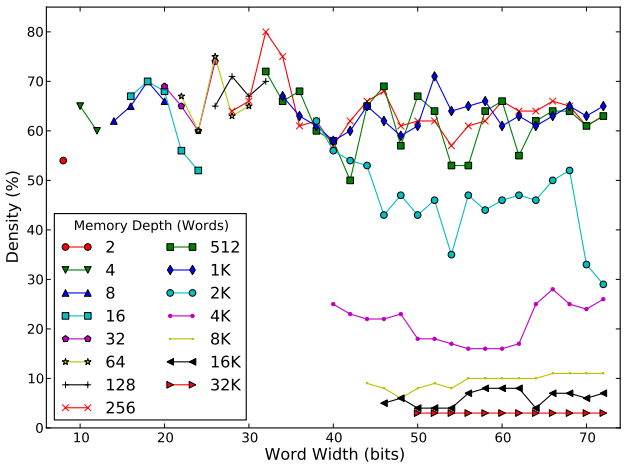


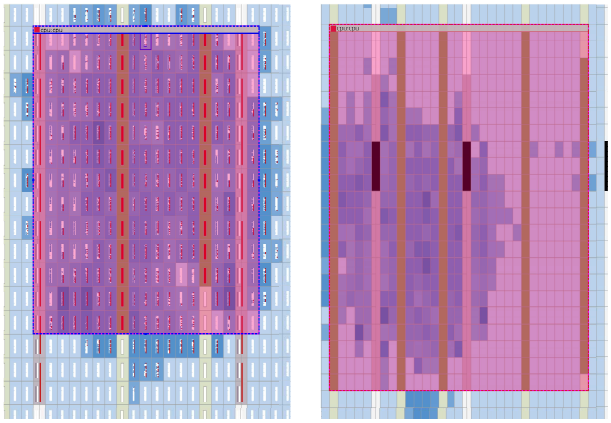
Figure 14: Density, measured as the percentage of ALUTs in actual use within the rectangular area containing an 8-stage Octavo instance, over word widths ranging from 8 to 72 bits and plotted for each addressable memory depth ranging from 2 to 32,768 words. BRAMs and DSP blocks do not count towards ALUT count.

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.

10.3 Density

Our final experiments seek to find if some Octavo configurations are "denser" than others, leaving fewer ALUTs, BRAMs, or DSP blocks unused within their rectangular area. Figure 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 address-



(a) 1024 Words (b) 4096 Words

Figure 15: Physical layout of an 8-stage, 72-bit wide Octavo instance with (a) 1024 and (b) 4096 memory words. 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.

able 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.

Figures 15(a) and 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 colored rectangular area contains only the ALUTs used by Octavo. Any outside ALUTs belong to a test harness and are ignored. The columns contain the DSP blocks which implement the Multiplier, and the BRAMs for the Memory. The remaining small block denote groups of ALUTs, with shade indicating the relative number of ALUTs in use 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, and prevents the FPGA CAD tools from placing and routing multiple Octavo instances (or other modules) in parallel, lengthening the design cycle. Further work may lead us to create vector/SIMD versions of Octavo to reclaim unused resources.

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.

11. CONCLUSIONS

In this paper 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 32,768 words), and does so 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 linearly with word width, increasing 11.4x in area 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 rectangular area to contain them than would compactly contain the processor logic.

12. FURTHER WORK

Our FPGA-centric architecture approach led us to Octavo, a fast but unconventional architecture. We will next attempt to push more standard processor features back into Octavo to determine whether a high F_{max} can be maintained with more conventional architecture support. For example, we will attempt to provide some support for indirect memory access and possibly eliminate the need for code synthesis and non-re-entrant code. We will also investigate the possibility of allowing fewer threads than pipeline stages via cheap methods for hazard detection and thread scheduling [13]. Beyond a single Octavo datapath, other important avenues of research include scaling Octavo to have multiple datapaths with vector/SIMD support, and to have interconnect, communication, and synchronization between multiple cores. We will also work towards connecting to a data parallel and highly-threaded high-level programming model such as OpenCL. Finally, we hope to explore the applicability of Octavo and its descendants to other FPGA devices.

13. ACKNOWLEDGMENTS

Thanks to Altera and NSERC for financial support, the reviewers for useful feedback and Joel Emer, Intel, for insightful comments.

14. REFERENCES

[1] ALTERA. DC and Switching Characteristics for Stratix IV Devices. http://www.altera.com/literature/hb/stratix-iv/stx4_siv54001.pdf, June 2011. Version 5.1, Accessed Dec. 2011.

[2] ALTERA. Nios II Performance Benchmarks. http://www.altera.com/literature/ds/ds_nios2_perf.pdf, June 2011. Version 7.0.

[3] ALTERA. Nios II Processor. http://www.altera.com/devices/processor/nios2/ni2-index.html, October 2011. Accessed December 2011.

[4] ANJAM, F., NADEEM, M., AND WONG, S. A VLIW softcore processor with dynamically adjustable issue-slots. In *Field-Programmable Technology (FPT), 2010 International Conference on* (dec. 2010).

[5] CHOU, C., SEVERANCE, A., BRANT, A., LIU, Z., SANT, S., AND LEMIEUX, G. VEGAS: Soft Vector Processor with Scratchpad Memory. In *ACM/IEEE International Symposium on Field-Programmable Gate Arrays* (February 2011).

[6] 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. 2* (June 2009).

[7] 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).

[8] 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* (April 2006).

[9] 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.

[10] 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* (2005).

[11] KNUTH, D. E. *The Art of Computer Programming, Volume 1 (3rd ed.): Fundamental Algorithms*. 1997.

[12] LABRECQUE, M., AND STEFFAN, J. Improving Pipelined Soft Processors with Multithreading. In *International Conference on Field Programmable Logic and Applications* (Aug. 2007).

[13] LABRECQUE, M., AND STEFFAN, J. G. Fast Critical Sections via Thread Scheduling for FPGA-based Multithreaded Processors. In *in International Conference on Field Programmable Logic and Applications* (2009).

[14] MOUSSALI, R., GHANEM, N., AND SAGHIR, M. Microarchitectural Enhancements for Configurable Multi-Threaded Soft Processors. In *International Conference on Field Programmable Logic and Applications* (Aug. 2007).

[15] MOUSSALI, R., GHANEM, N., AND SAGHIR, M. A. R. Supporting multithreading in configurable soft processor cores. In *CASES '07: Proceedings of the 2007 international conference on Compilers, Architecture, and Synthesis for Embedded Systems* (2007), ACM.

[16] SAGHIR, M. A. R., EL-MAJZOUB, M., AND AKL, P. Datapath and ISA Customization for Soft VLIW Processors. In *ReConFig 2006: IEEE International Conference on Reconfigurable Computing and FPGAs* (Sept. 2006).

[17] XILINX. MicroBlaze Soft Processor. http://www.xilinx.com/microblaze, October 2011. Accessed December 2011.

[18] YIANNACOURAS, P., STEFFAN, J. G., AND ROSE, J. VESPA: portable, scalable, and flexible FPGA-based vector processors. In *Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems* (2008), CASES '08.

[19] 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. 2* (June 2009).