The Microarchitecture of FPGA-Based Soft Processors

Peter Yiannacouras, Jonathan Rose, and J. Gregory Steffan
Department of Electrical and Computer Engineering
University of Toronto
10 King's College Road
Toronto, Canada
{yiannac,jayar,steffan}@eecg.utoronto.ca

ABSTRACT

As more embedded systems are built using FPGA platforms, there is an increasing need to support processors in FPGAs. One option is the *soft processor*, a programmable instruction processor implemented in the reconfigurable logic of the FPGA. Commercial soft processors have been widely deployed, and hence we are motivated to understand their microarchitecture. We must re-evaluate microarchiteture in the soft processor context because an FPGA platform is significantly different than an ASIC platform—for example, the relative speed of memory and logic is quite different in the two platforms, as is the area cost. In this paper we present an infrastructure for rapidly generating RTL models of soft processors, as well as a methodology for measuring their area, performance, and power. Using our automatically-generated soft processors we explore the microarchitecture trade-off space including: (i) hardware vs software multiplication support; (ii) shifter implementations; and (iii) pipeline depth, organization, and forwarding. For example, we find that a 3-stage pipeline has better wall-clock-time performance than deeper pipelines, despite lower clock frequency. We also compare our designs to Altera's NiosII commercial soft processor variations and find that our automatically generated designs span the design space while remaining very competitive.

Categories and Subject Descriptors

C.1.3 [**Processor Architectures**]: Other Architecture Styles—*Adaptable architectures*

General Terms

Measurement, Performance, Design

Keywords

Soft processor, FPGA, exploration, microarchitecture, RTL generation, application specific tradeoff, Nios, embedded processor, pipeline, ASIP, SPREE

1. INTRODUCTION

With the increasing cost and time-to-market of designing a state-of-the-art ASIC, an increasing number of embedded systems are being built using Field Programmable Gate Array (FPGA) platforms. Such designs often contain one or more embedded microprocessors which must also migrate to the FPGA platform to avoid the increased cost and latency of a multi-chip design. FPGA vendors have addressed this issue with two solutions: (i) incorporating one or more *hard processors* directly on the FPGA chip (eg., Xilinx Virtex II Pro and Altera Excalibur), and (ii) implementing one or more *soft processors* using the FPGA fabric itself (eg., Xilinx MicroBlaze and Altera Nios).

While FPGA-based hard processors can be fast, small, and relatively cheap, they have several drawbacks. First, the number of hard processors included in the FPGA chip may not match the number required by the application, leading to either too few or wasted hard processors. Second, the performance requirements of each processor in the application may not match those provided by the available FPGA-based hard processors (eg., a full hard processor is often overkill). Third, due to the fixed location of each FPGA-based hard processor, it can be difficult to route between the processors and the custom logic. Finally, inclusion of one or more hard processors specializes the FPGA chip, impacting the resulting yield and narrowing the customer base for that product.

While a soft processor cannot easily match the performance/area/power of a hard processor, soft processors do have several compelling advantages. Using a generic FPGA chip, a designer can implement the exact number of soft processors required by the application, and the CAD tools will automatically place them within the design to ease routing. Since it is implemented in configurable logic, a soft processor can be tuned by varying its implementation and complexity to match the exact requirements of an application. While these benefits have resulted in wide deployment of soft processors in FPGA-based embedded systems [27], the architecture of soft processors has yet to be studied in depth.

1.1 Understanding Soft Processor Microarchitecture

The microarchitecture of processors has been studied by many researchers and vendors for decades. However, the trade-offs for FPGA-based soft processors are significantly different than those implemented directly in transistors [25, 26]: for example, on-chip memories are often faster than the clock speed of a soft processor pipeline, and hard multipli-

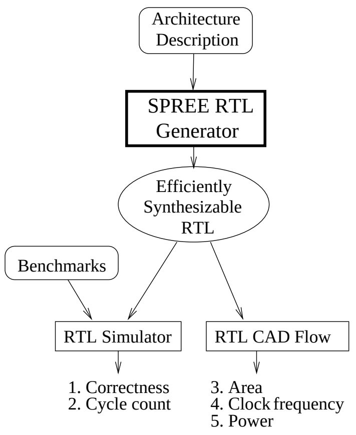

Figure 1: Overview of the SPREE system.

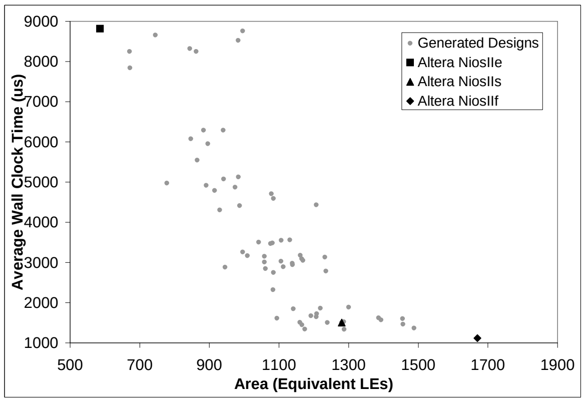

Figure 2: Comparison of our generated designs vs the three Altera Nios II variations.

ers are area-efficient and fast compared to other functions implemented in configurable logic. Furthermore, due to the difficulty in varying designs at the logic layout level, processor microarchitecture has traditionally been studied using high-level functional simulators that estimate area and performance. In contrast, FPGA CAD tools allow us to quickly and accurately measure the exact speed, area, and power of the final placed and routed design for any soft processor. Hence we have the compelling opportunity to develop a complete and accurate understanding of soft processor microarchitecture.

Our long-term research agenda is to be able to automatically navigate the soft processor design space, and to make intelligent application-specific architectural trade-offs based on a full understanding of soft processor microarchitecture. In this paper we describe our initial work comprised of the following three goals: (i) to build a system for automatically-generating soft processors with minimal input from the user; (ii) to develop a methodology for comparing soft processor architectures; (iii) to begin to populate and analyze the soft processor design space. We have developed the *Soft Processor Rapid Exploration Environment (SPREE)* (shown in Figure 1), a system which automatically generates an RTL-level description of a soft processor from text-based ISA and datapath descriptions—SPREE is described in detail in Section 2. We use FPGA CAD tools to accurately measure area, clock frequency, and power of the resulting RTL designs, and we also verify correctness and measure the cycle counts of several embedded benchmark applications on these designs. As a preview of the capabilities of our system, Figure 2 shows wall-clock-time vs area for our initial generated designs as well as for the three variations of the industrial Altera NiosII soft processor [7] (these results are described in detail in Section 4). Our designs successfully span the trade-off space between the NiosII variations, and a few designs even provide greater performance with less area than one of the NiosII variations.

1.2 Related Work

While industry architects have optimized commercial soft processors [25, 26], to the best of our knowledge a microarchitectural exploration of FPGA-based soft processors has never been conducted in the depth presented in this paper.

SPREE is a system for architecture exploration, of which there are numerous previously-proposed approaches that fall into two categories: *parametrized cores* and *architecture description languages* (ADLs). A parameterized core [3, 8, 15, 17, 24, 28] is designed at the RTL level allowing for certain aspects of the architecture to be adjusted. Few existing parameterized cores target FPGAs specifically, and all of them narrowly constrain the potential design space. Changing the ISA, timing, or control logic requires large-scale modification to the source code of the processor.

A multitude of architecture exploration environments have been proposed—a good summary of these is provided by Gries [16] and by Tomiyama [31]. The foundation of these environments is the ADL which completely specifies the design of the processor. The focus of these ADLs is to drive the creation of custom compilers, instruction set simulators, cycle accurate simulators, and tools for estimating area and power. Unfortunately these ADLs are often verbose and overly general (for our purposes). Furthermore, few ADLs provide a path to synthesis through RTL generation, and for those that do [20, 30, 32] the resulting RTL is often a very high-level description (for example, in SystemC), and therefore depends heavily on synthesis tools to optimize the design. In an FPGA, using different hardware resources results in large trade-offs—hence the soft processor designer needs direct control of these decisions.

There are two systems most closely related to SPREE: UNUM and PEAS-III. The UNUM [14] system automatically generates microprocessor implementations where users can seamlessly swap components without explicit changes to the control logic. The output of the system is a processor implemented in *Bluespec* [1], a behavioral synthesis language which can be translated to RTL. The drawback to this approach is that there is overhead to using the behavioral synthesis language, which also abstracts away implementation details that are essential for efficient FPGA synthesis.

The PEAS-III project [21] focuses on ISA design and hardware software co-design, and proposes a system which generates a synthesizable RTL description of a processor from a clock-based micro-operation description of each instruction. Although PEAS-III enables a broad range of exploration, it requires changes to the description of many instructions to produce a small structural change to the architecture. Instead of inferring the datapath from the micro-operation instruction descriptions, in SPREE we infer the micro-operations of each instruction from the dat-

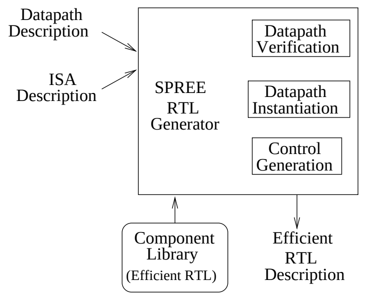

Figure 3: An overview of the SPREE RTL generator.

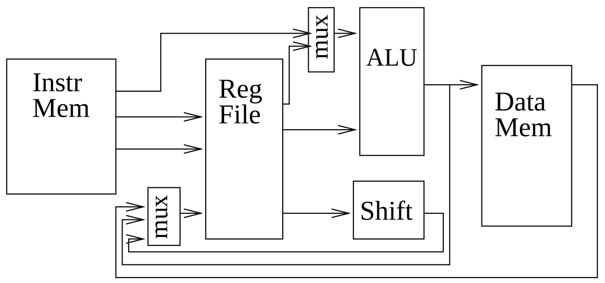

Figure 4: A datapath description shown as an interconnection of components.

apath, allowing the user to carefully design the datapath. PEAS-III was used [19] to conduct a synthesis-driven exploration which explored changing the multiply/divide unit to sequential (34-cycles), and then adding a MAC (multiply-accumulate) instruction. The results were compared for their area and clock frequency as reported by the synthesis tool.

Finally, there has recently been a surge of interest in using FPGAs as a platform for doing processor and system-level architectural studies [18]. However, the goal of such work is to overcome the long simulation times associated with software simulators of large and complex processors to enable cycle-accurate simulation.

1.3 Contributions

This paper makes the following three contributions. First, we present a methodology for comparing and measuring soft processors architectures. Second, we perform detailed benchmarking of a wide-variety of soft processor architectures, including accurate area, clock frequency, and energy measurements, compare our results to Altera's Nios II soft processor variations—this is what we believe is the first such study in this depth. Finally, we suggest architectural enhancements, component implementations, and potential compiler optimizations which are specific to FPGA-based soft processors.

2. OVERVIEW OF THE SPREE SYSTEM

The purpose of SPREE is to facilitate the rapid generation of RTL for a wide variety of soft processors, enabling us to thoroughly explore and understand the soft processor design space. As shown in Figure 3, SPREE takes as input a description of the target ISA and the desired datapath, verifies that the datapath supports the ISA, instantiates the datapath, and then generates the corresponding control logic. The output is a complete and synthesizable RTL description (in Verilog) of a soft processor. This section describes the SPREE system, further details of which are available online [33, 34].

It is important to note that for now we consider simple, in-order issue processors that use only on-chip memory and hence have no cache. The memory on the FPGA is faster than a typical processor implementation eliminating the need for exploring the memory hierarchy. Moreover, the largest FPGA devices have more than one megabyte of on chip memory which is adequate for many applications (in the future we plan to broaden our application base to those requiring off-chip RAM and caches). We also do not yet include support for dynamic branch prediction, exceptions, or operating systems. Finally, in this paper we do not modify the ISA (we restrict ourselves to a subset of MIPS-I) or the compiler, with the exception of evaluating software vs hardware support for multiplication (due to the large impact of this aspect on cycle time and area).

2.1 Input: The Architecture Description

The input to the SPREE system is the description of the desired processor, composed of textual descriptions of the target ISA and the processor datapath. The datapath is described as a graph of components from the Component Library. The functionality of each component and the required functionality of each instruction in the ISA are both described in a common language. The following describes each of these in more detail.

2.1.1 Describing the Datapath

The datapath is described by listing the set of components to use and the interconnection between their physical ports—an example of which is shown in Figure 4. A processor architect can therefore create any datapath that supports the specified ISA. This structural approach enables efficient synthesis, for example when pipelining: often a minor re-organization of the pipeline may be required to accommodate a high delay path through the circuit. Balancing logic delay to achieve maximum clock frequency depends critically on this ability to manually arrange the pipeline stages, a task that is beyond the retiming capabilities of modern synthesis tools. Without this ability we risk making incorrect conclusions based on poor implementations.

The datapath must also include certain control components when necessary, again using the case of a pipelined processor for example: pipeline registers, hazard detection units, and forwarding lines are available in the Component Library and must be used in an appropriate combination to ensure correct functionality of the processor. We hope to automate the insertion of these components in the future.

2.1.2 Selecting and Interchanging Components

The SPREE Component Library stores the RTL code, interface, and interface descriptions of every available processor component, for example: register files, shifters, and ALUs. When selected in the datapath description, a component is included in the resulting processor architecture. To evaluate different options for a given unit, a user can easily interchange components and regenerate the control logic.

```
Module alu_small {
   Input opA 32
   Input opB 32
   Output result 32
   Opcode opcode 2 {
       ADD 0 0
       SUB 1 0
       SLT 2 0
       }
}
```

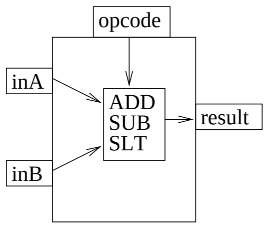

Figure 5: Sample component description for a simplified ALU. The ALU supports the GENOPs `ADD`, `SUB`, and `SLT`.

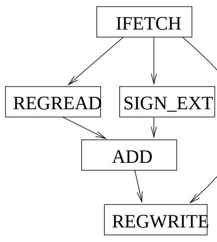

Figure 6: The MIPS ADDI instruction (addi rd,rs,IMM) shown as a dependence graph between GENOPs.

2.1.3 *Creating and Describing Custom Components*

The Component Library can be expanded to include custom components. Designers must provide the RTL description of the new component and describe its interface and functionality in a library entry. Figure 5 shows a simplified library entry for a small ALU. The interface is described by the `Module` line, which defines the name of the module, and by the names and bit-widths of the physical input and output ports which follow. The functionality of the component is described in the `Opcode` section which defines an opcode port (`opcode`). The fields inside the Opcode section describe the functionality of the component. Each line begins with the name of the supported operation and is proceeded by two integers: (i) the opcode port value that selects that operation, and (ii) the latency in cycles for the operation to complete (aynchronous components are denoted with a negative latency). For example, the ADD function of the simple ALU specified in figure 5 is selected by opcode 0 and has zero extra cycles of latency. The name of the supported operation comes from a set of *generic operations (GENOPS)* which form a common language for describing the behavior of a component and also the semantics of an instruction. Each GENOP is a small unit of functionality performed inside a typical microprocessor: examples of GENOPs include `ADD`, `XOR`, `PCWRITE`, `LOADBYTE`, and `REGREAD`. An opcode can support an arbitrary number of GENOPs, which allows for resource sharing, and a component can have an arbitrary number of opcode ports, which allows for parallellism.

2.1.4 *Describing the ISA*

Each instruction in the processor description is described in terms of a data dependence graph of GENOPs. An example of such a graph is shown in Figure 6 for a MIPS `Add-Immediate` instruction. In the graph, the nodes are GENOPs and the edges represent a flow of data from one GENOP to another. We institute the rule that *no GENOP can execute until all of its inputs are ready.* For a given instruction this graph shows the mandatory sequence of GENOPs, although the datapath will determine the exact timing.

2.2 Generating a Soft Processor

From the above inputs, SPREE generates a complete Verilog RTL model of the desired processor. As shown in Figure 3 and described below, SPREE generates the processor in three phases: (i) datapath verification, (ii) datapath instantiation, and (iii) control generation.

2.2.1 *Datapath Verification*

Since there are two separate inputs that describe the processor datapath and the ISA, SPREE must verify that the datapath indeed supports the ISA by ensuring that two conditions are met. First, the set of GENOPs supported by all of the components collectively must include each of the GENOPs required by the ISA. Second, the interconnection of these components must be such that the flow of data between GENOPs is analogous to the flow of data imposed by the ISA description. Both conditions can be met simply be ensuring that the GENOP graph describing each instruction in the ISA is a subgraph of the datapath GENOP graph. Note that by tracking which portions of the datapath GENOP graph remain unused (by any instruction), we can automatically trim unnecessary components and connections from the datapath.

2.2.2 *Datapath Instantiation*

From the input datapath description, we must generate an equivalent Verilog description. This task is relatively straight-forward since the connections between components are known from the datapath description. However, to simplify the input, SPREE allows physical ports to be driven by multiple sources and then automatically inserts the logic to multiplex between the sources, and generates the corresponding select logic during the control generation phase.

2.2.3 *Control Generation*

Once the datapath has been described and verified, SPREE automatically performs the laborious task of generating the logic to control the datapath's operation to correctly implement the ISA. The control logic provides two things to each component: what operation to perform (through *Opcodes*), and when to perform it (through *Enables*). From the datapath verification we know which operation is performed by each component for a given instruction, hence the opcode is simply decoded from the instruction. To distribute the opcode signals to multiple stages in the case of a pipelined processor, the control logic propagates the instruction word through every pipeline stage, and inserts necessary decode logic in each stage. The user can optionally locate the decode logic in the previous stage, which can have the effect of shortening a control-dominated critical path.

Enables are used to schedule operations. Generation of enable signals must take into account the datapath, asynchronous components which may take multiple cycles to complete, and hazard detection logic in the case of a pipelined processor. Generation of enable signals proceeds as follows: First the generator collects all timing information from each component. Next it analyzes the datapath and infers the

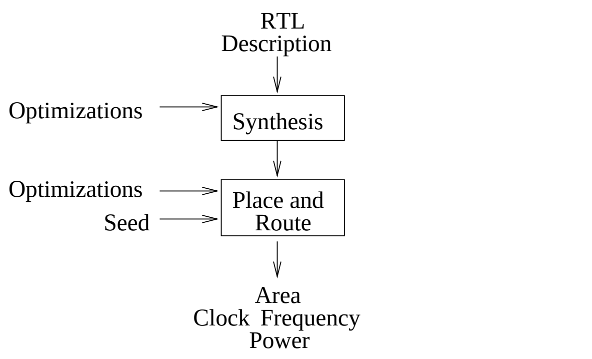

Figure 7: CAD flow overview. Optimizations and seed values add noise to the system affecting the final area, clock frequency, and power measurements, and hence must be carefully managed.

pipeline stage of each component. Within each pipeline stage, local stall signals are extracted and combined. Any stall signal is propagated to earlier stages so that all stages behind that which created the stall are also stalled— we refer to this propagation of stalls as the *stall network*. From the stall network, the enables are easily generated since a component is enabled if the instruction uses it and the component is not stalled. Similarly, the generator also generates *squash* signals: these are used to kill instructions and replace them with null operations, as in the case of a mis-speculated branch.

3. EXPERIMENTAL FRAMEWORK

Having described the design and implementation of SPREE in the previous section, we now describe our framework for measuring and comparing the soft processors it produces. We present a method for verifying the correctness of our soft processors, methods for employing FPGA CAD tools, a methodology for measuring and comparing soft processors (including a commercial soft processor), and the benchmark applications that we use to do so.

3.1 Processor Verification

SPREE verifies that the datapath is capable of executing the target ISA—however, we must also verify that the generated control logic and the complete system function correctly. We implement trace-based verification by using a cycle-accurate industrial RTL simulator (Modelsim [6]) that generates a trace of all writes to the register file and memory as it executes an application. We compare this trace to one generated by MINT [5] (a MIPS instruction set simulator) and ensure that the traces match. SPREE automatically generates test benches for trace dumping and creates debug signals to facilitate the debugging of pipelined processors.

3.2 FPGAs, CAD, and Soft Processors

While SPREE itself is indifferent to the target FPGA architecture, we have selected Altera's Stratix I [23] device for performing our FPGA-based exploration. The Components Library thus targets Stratix I FPGAs. We use Quartus II v4.2 CAD software for synthesis, technology mapping, placement and routing. We synthesize all designs to a Stratix EP1S40F780C5 device (a middle-sized device in the family, with the fastest speed grade) and extract and compare area, clock frequency, and power measurements as reported by Quartus.

It is important to understand that one must proceed carefully when using CAD tools to compare soft processors. Normally when an HDL design fails design constraints (as reported by the CAD software), there are three alternatives that avoid altering the design: (i) restructure the HDL code to encourage more efficient synthesis, (ii) use different optimization settings of the CAD tools, and (iii) perform seed sweeping—a technique which selects the best result among randomly-chosen starting placements. These three are design-independent techniques for coaxing a design into meeting specifications, and their existence illustrates the non-determinism inherent in combinatorial optimization applied in a practical context.

We have taken the following measures to counteract variation caused by the non-determinism caused by CAD tools: (i) we have coded our designs structurally to avoid the creation of inefficient logic from behavioral synthesis; (ii) we have experimented with optimization settings and ensured that our conclusions do not depend on them, and (iii) for the area and clock frequency of each soft processor design we determine the arithmetic mean across 10 seeds (different initial placements before placement and routing) so that we are 95% confident that our final reported value is within 2% of the true mean.

3.3 Metrics for Measuring Soft Processors

To measure area, performance, and power, we must decide on an appropriate set of specific metrics. For an FPGA, one typically measures area by counting the number of resources used. In Stratix, the main resource is the *Logic Element* (LE), where each LE is composed of a 4-input *lookup table* (LUT) and a flip flop. Other resources, such as the hardware multiplier block, and memory blocks can be converted into an equivalent number of LEs based on the relative areas of each in silicon. The relative area of these blocks was provided by Altera [13] Hence we report area in terms of *equivalent LEs*.

To measure performance, we have chosen to report the wall-clock-time for execution of a collection of benchmark applications, since reporting clock frequency or instructions-per-cycle (IPC) alone can be misleading. To be precise, we multiply the clock period (determined by the Quartus timing analyzer after routing) with the arithmetic mean of the cycles-per-instruction (CPI) across all benchmarks, and multiply that by the average number of instructions executed across all benchmarks. Averaging in this way prevents a long-running benchmark from biasing our results.

To measure power, we use Quartus' Power Play tool which produces a power measurement based on the switching activities of post-placed-and-routed nodes determined by simulating benchmark applications on a post-placed-and-routed netlist of a processor in Modelsim. We subtract out static power, and we also subtract the power of the I/O pins since this power dominates and is more dependent on how the processor interfaces to off-chip resources than its microarchitecture. For each benchmark, we measure the energy per instruction and report the arithmetic mean of these across the benchmark set.

Table 1: Benchmark applications evaluated.

Source	Benchmark	Modified	Dyn. Instr. Counts
MiBench [4]	BITCNTS	di	26,175
	CRC32	d	109,414
	QSORT*	d	42,754
	SHA	d	34,394
	STRINGSEARCH	d	88,937
	FFT*	di	242,339
	DIJKSTRA*	d	214,408
	PATRICIA	di	84,028
XiRisc [12]	BUBBLE_SORT		1,824
	CRC		14,353
	DES		1,516
	FFT*		1,901
	FIR*		822
	QUANT*		2,342
	IQUANT*		1,896
	TURBO		195,914
	VLC		17,860
Freescale [2]	DHRY*	i	47,564
RATES [9]	GOL	di	129,750
	DCT*	di	269,953

* Contains multiply
d Reduced data input set
i Reduced number of iterations

3.4 Comparing with Altera NiosII Variations

To ensure that our generated designs are indeed interesting and do not suffer from prohibitive overheads, we have selected Altera's NiosII family of processors for comparison. NiosII has three mostly-unparameterized variations: `NiosIIe`, a very small unpipelined 6-CPI processor with a serial shifter and software multiplication support; `NiosIIs`, a 5-stage pipeline with a multiplier-based barrel shifter, hardware multiplication, and an instruction cache; and `NiosIIf`, a large 6-stage pipeline with dynamic branch prediction, instruction and data caches, and an optional hardware divider.

We have taken several measures to ensure that comparison against the NiosII variations is as fair as possible. We have generated each of the Nios processors with memory systems identical to those of our designs: two 64KB blocks of RAM for separate instruction and data memory. We do not include caches in our measurements, though some logic required to support the caches will inevitably count towards the NiosII areas. The NiosII instruction set is very similar to the MIPS-I ISA with some minor modifications (for example, no branch delay slots)—hence NiosII and our generated processors are very similar in terms of ISA. NiosII supports exceptions and OS instructions, which are so far ignored by SPREE. Finally, like NiosII, we also use GCC as our compiler, though we did not modify any machine specific parameters nor alter the instruction scheduling. Despite these differences, we believe that comparisons between NiosII and our generated processors are relatively fair, and that we can be confident that our architectural conclusions are sound.

3.5 Benchmark Applications

We measure the performance of our soft processors using 20 embedded benchmark applications from four sources, (as summarized in Table 1): XiRisc [12], MiBench [4], RATES [9], and Freescale [2]. Some applications operate solely on integers, and others on floating point values (although for now we use only software floating point emulation); some are compute intensive, while others are control intensive. Table 1 also indicates any changes we have made to the application to support measurement, including reducing the size of the input data set to fit in on-chip memory (`d`), and decreasing the number of iterations executed in the main loop to reduce simulation times (`i`). Additionally, all file and other I/O were removed since we do not yet support an operating system.

4. EXPLORING SOFT PROCESSOR MICROARCHITECTURE

In this section we use SPREE to perform an initial investigation into the microarchitectural trade-offs for soft-processors. We first validate our infrastructure by showing that the generated designs are comparable to the highly optimized NiosII commercial soft processor variations. We then investigate in detail the following aspects of soft processor microarchitecture: 1) hardware vs software multiplication—the tradeoffs in containing hardware support for performing multiply instructions; 2) shifter implementations—how one should implement the shifter, since shifting logic can be expensive in FPGA fabrics; 3) Pipelining—we look at pipeline organization, measure different pipeline depths, and experiment with inter-stage forwarding logic. This small initial design space permits us to perform a very complete study, which will not be possible in the future when we support the variation of a greater number of archtectural features such as caches and branch predictors.

4.1 Comparison with NiosII Variations

As previewed earlier in Figure 2, we compare our generated designs to the three NiosII variations. Thus, there are three points in the space for NiosII, with `NiosIIe` being furthest left (smallest area, lowest performance), `NiosIIf` furthest right (largest area, highest performance), and `NiosIIs` in between. The figure shows that our generated designs span the design space, and that one of our generated designs even dominates the `NiosIIs`—hence we examine that processor in greater detail.

The processor of interest is an 80MHz 3-stage pipelined processor, which is 9% smaller and 11% faster in wall-clock-time than the `NiosIIs`, suggesting that the extra area used to deepen `NiosIIs`'s pipeline succeeded in increasing the frequency, but brought overall wall-clock-time down.[1] The generated processor has full inter-stage forwarding support and hence no data hazards, and suffers no branching penalty. The pipeline stalls only on load instructions (which must await the value being fetched from data memory) and on shift and multiply instructions (which complete in two cycles instead of one, since both are large functional units). The CPI of this processor is 1.36 whereas the CPIs of `NiosIIs` and `NiosIIf` are 2.36 and 1.97 respectively. However, this large gap in CPI is countered by a large gap in clock frequency: `NiosIIs` and `NiosIIf` achieve clock speeds of 120 MHz and 135 MHz respectively, while the generated processor has a clock of only 80MHz. These results demonstrate the importance of evaluating wall-clock-time over clock frequency or CPI alone, and that faster frequency is not always better. A similar conclusion was drawn for the original Nios

[1]A cynic might attribute this decision to the strong marketing influence of clock frequency.

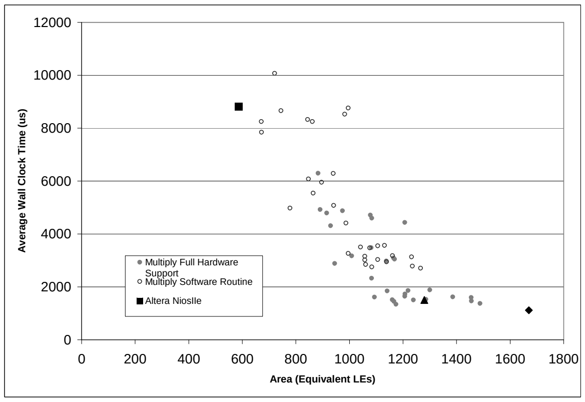

Figure 8: Average wall-clock-time vs area of processors with and without hardware multiplication support. The `NiosIIe` (top left) supports multiplication in software.

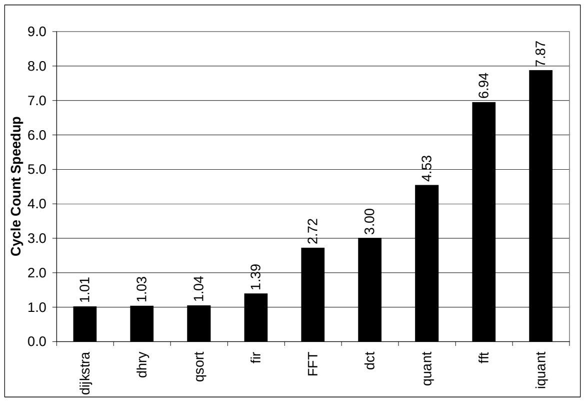

Figure 9: Cycle count speedup of hardware support for multiplication, for only those benchmarks that contain multiplies.

by Plavec [29], who matched the Nios wall clock time by targetting lower cycle counts in spite of slower clock frequencies.

Our smallest generated processor is within 15% of the area of `NiosIIe`, but is also 11% faster (in wall-clock-time). The area difference can be attributed to overhead in the generated designs compared to the hand optimized `NiosIIe`, knowing that overheads are more pronounced in a smaller-area design (600-700 LEs). Altera reports that `NiosIIe` typically requires 6 cycles per instruction, while our smallest processor typically requires 2-3 cycles per instruction. Although our design has less than half the CPI of the `NiosIIe`, our design also has half the clock frequency (82MHz for our design, 159 MHz for NiosIIe), reducing the CPI benefit to an 11% net win in wall-clock-time for our design.

Bearing in mind the differences between NiosII and our processors, it is not our goal to draw architectural conclusions from a comparison against NiosII. Rather, we see that the generator can indeed populate the design space while remaining relatively competitive with commercial, hand optimized soft processors.

4.2 The Impact of Hardware vs Software Multiplication

Whether multiplication is supported in hardware or software can greatly affect the area, performance, and power of a soft processor. For this reason, the `NiosIIe` has no hardware support while the other two Nios variations have full support. There may be many variations of multiplication support which trade off area for cycle time; we consider only full multiplication support using the dedicated multipliers in the FPGA.[2]

Figure 8 obviates the trade-off between area and wall-clock-time for multiplication support. In the figure we plot the NiosII variations, as well as a collection of our generated designs each with either full hardware support for multiplication or software-only multiplication. In terms of area, removing the multiplication saves 230 equivalent LEs, or approximately one fifth of the total area. However, in some of the designs, the multiplier is also used to perform shift operations as recommended by Metzgen [26], hence the multiplier itself is not actually removed even though it is no longer used for multiplies. For such designs the control logic, multiplexing, and the MIPS-I `HI` and `LO` registers used for storing the multiplication result are all removed, resulting in an area savings of approximately 80 equivalent LEs. In both cases the area savings is substantial, and depending on the desired application may be well worth any reduction in performance.

Figure 9 shows the impact of hardware support for multiplication on the number of cycles to execute each benchmark, but only for those benchmarks that use multiplication (Table 1). We see that some applications are sped up minimally while others benefit up to 8x from a hardware multiplier, proving that multiplication support is certainly an application-specific design decision. Software-only support for multiplication roughly doubles the total number of cycles required to execute the entire benchmark suite compared to hardware support. This increase translates directly into a wall-clock-time slowdown of a factor of two, since the clock frequency remains unimproved by the removal of the multiplication hardware.

The impact of multiplication support on energy is also very interesting. Experiments showed that the energy consumption of the dedicated multipliers and supporting logic is insignificant. This conclusion is expected because the switching activity of this path is relatively low, and because the dedicated multipliers are implemented efficiently at the transistor level. In fact, even with appreciable switching activities, the multipliers are often seen to make no contribution to power. The only effect of the hardware multiplication support is a decreased instruction count: while this reduces total energy, the energy consumed per instruction remains the same.

4.3 The Impact of Shifter Implementation

Shifters can be implemented very efficiently in an ASIC design. However, this is not true for FPGAs due to the relatively high cost of multiplexing logic [26]. We study three different shifter implementations: a *serial shifter*, implemented by using flip-flops as a shift register and requiring one cycle per bit shifted; a *LUT-based barrel shifter* implemented in LUTs; and a *multiplier-based barrel shifter* implemented using hard multipliers. We study the effects of using

[2]Hybrid implementations that we do not yet consider can also provide partial multiplication support in hardware.

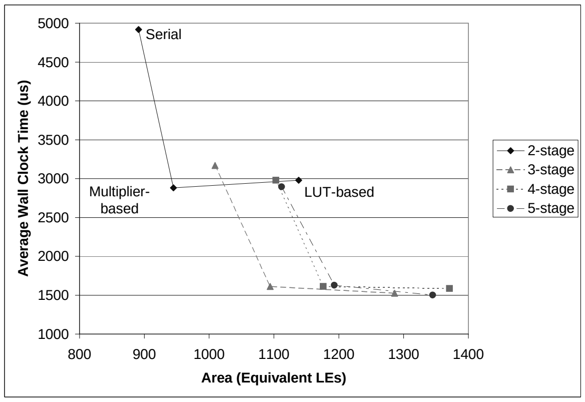

Figure 10: Average wall-clock-time vs area for different pipeline depths. In each series we have in order from left-to-right the 3 shifter implementations: Serial, Multiplier-based, LUT-based.

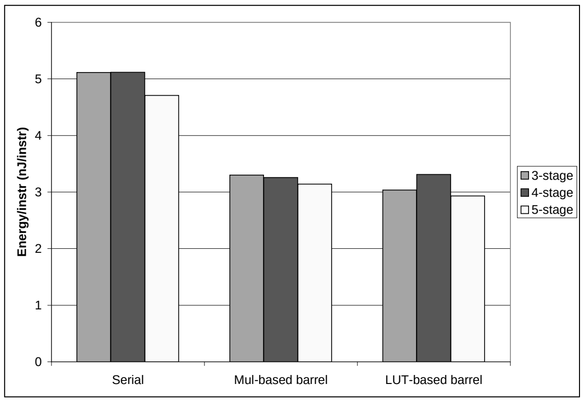

Figure 11: Energy per instruction across different pipeline depths and different shifter implementations

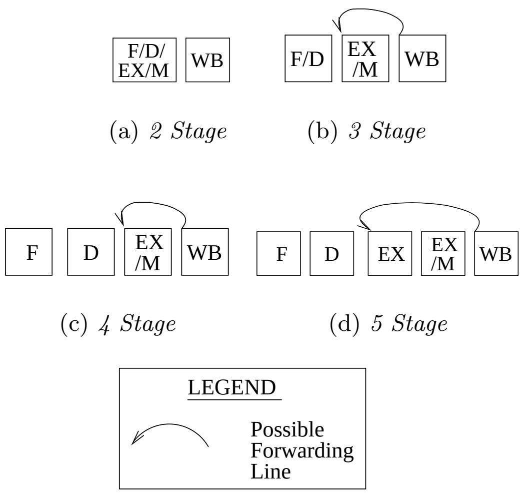

Figure 12: Processor pipeline organizations studied. Arrows indicate possible forwarding lines.

each of these shifter types over 4 different architectures, each with a different pipeline depth.

Figure 10 gives the wall-clock-time versus area tradeoff space for the different shifter implementations in the 4 architectures. It shows that the serial shifter is the smallest while the LUT-based barrel shifter is largest, on average 250 LEs larger than the serial shifter. In contrast, the multiplier-based shifter is only 64 LEs larger than the serial shifter: the multiplier is being shared for both shift and multiplication instructions, and the modest area increase is caused by the additional logic required to support shift operations in the multiplier.

The impact of each shifter type on wall-clock-time is also seen in Figure 10. On average, the performance of both the LUT-based and multiplier-based shifters are the same, because in all architectures the cycle counts are identical. The differences in wall-clock-time are caused only by slight variations in the clock frequency for different architectures. Thus, the multiplier-based shifter is superior to the LUT-based shifter since it is smaller yet yields the same performance. There is a definite trade-off between the multiplier-based shifter and serial shifter: the multiplier-based shifter is larger as discussed before—however, it yields an average speedup of 1.8x over the serial shifter.

In Figure 11 we show the energy per instruction for each of the shifter types with three different pipelines. Both the LUT-based and multiplier-based barrel shifters consume the same amount of energy, even though the LUT-based shifter is significantly larger in area. This is due to the increased switching activity in the multiplier and its tighter integration with the datapath (MIPS multiply instructions are written to dedicated registers while the shift result must be written directly to the register file). The processors with serial shifters consume more energy per instruction than those with barrel shifters because of the switching activity in the pipeline while the serial shifter is stalled (the stages from execute to writeback continue and are eventually filled with null operations). The shifter itself consumes significant energy as counters and comparators are toggled for every cycle of the shift, in addition to the shift register itself. Further energy overhead is caused by the SPREE Component Library which is yet to utilize power-aware features (even when not used, many functional units remain active). As the pipeline stalls for many cycles, these overheads accumulate and surpass that of a barrel shifter which would complete without stalling.

4.4 The Impact of Pipelining

We now use SPREE to study the impact of pipelining in soft processor architectures by generating processors with pipeline depths between 2 and 5 stages, the organizations of which are shown in Figure 12. A purely unpipelined processor, or 1-stage pipeline, is neglected since fetching the next instruction and writing a result to the register file can be pipelined for free, increasing the throughput of the system and decreasing the size of the control logic by a small margin. The area and average wall-clock-time measurements of the pipelines are shown in Figure 10. For each pipeline depth, we averaged the measurements across the three shifter implementations described in the previous section. For every pipeline, data hazards are prevented through interlocking, branches are statically predicted to be not-taken, and instructions after a taken branch are squashed.

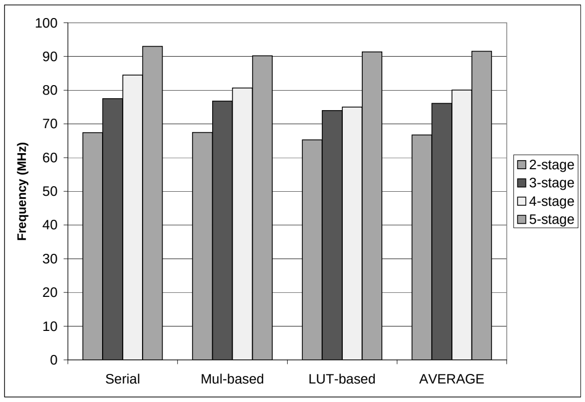

Figure 13: Clock frequency across different pipeline depths with different shifter implementations.

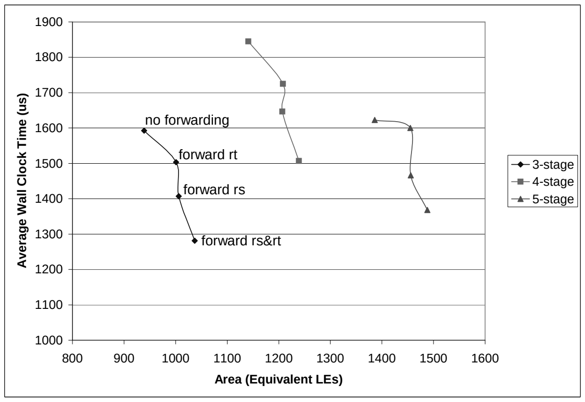

Figure 14: Average wall-clock-time vs area for different forwarding lines. As more forwarding is added, the processor moves right (more area) and down (faster wall-clock-time).

Figure 10 shows that area does indeed increase with the number of pipeline stages as expected, due to the addition of pipeline registers and data hazard detection logic. However, the 5-stage pipeline has only a small area increase over the 4-stage pipeline: for shorter pipelines, memory operations stall until they have completed, while in the 5-stage pipeline memory operations are contained within their own pipeline stage, eliminating the need for the corresponding stalling logic and saving some area for this design.

With respect to wall-clock-time, we see that deepening the pipeline improves performance over the shortest pipeline. The 2-stage pipeline has no branch penalty or data hazards. However, it suffers from reduced clock frequency and frequent stalls for multi-cycle operations: specifically, reading operands from the register file is multi-cycle because the register file is implemented using the synchronous RAMs in the FPGA, which inherently incur a cycle delay. Hence there is a large performance gain for increasing the pipeline depth from 2 to 3 stages. In the 3-stage pipeline we execute the operand fetch in parallel with the write back, which will cause stalls only on *read-after write* (RAW) hazards instead of on the fetch of every operand. Combined with the increase in clock frequency shown in Figure 13, this decrease in stalls leads to the 1.7x wall-clock-time speedup for the 3-stage pipeline over 2-stages.

While deciding the stage boundaries for our 3-stage pipeline was obvious and intuitive, deciding how to add a fourth pipeline stage was not. One can add a decode stage as shown in Figure 12(c), or further divide the execution stage. We implemented both pipelines for all three shifter types and observed that although the pipeline in Figure 12(c) is larger by 5%, its performance is 16% better. Hence there is an area-performance trade-off, proving that such trade-offs exist not only in pipeline depth, but also in pipeline organization.

While frequencies improve for the 4 and 5 stage pipelines, their cycle counts increase due to increased branch penalties and data hazards. The net effect on wall-clock-time, shown in Figure 10, shows that the performance of the 3, 4, and 5 stage pipelines improves only slightly. While the 3-stage pipeline seems the most attractive, it has the least opportunity for future performance improvements: for example, the cycle count increase suffered by the deeper pipelines can potentially be reduced by devoting additional area to branch prediction or more aggressive forwarding. Frequency improvements may also be possible with more careful placement of pipeline registers.

The energy per instruction of the three, four, and five stage pipelines can be seen in Figure 11. The energy consumption remains relatively consistent with a slight decrease as the pipeline depth increases (in spite of the extra area gained). We attribute this to decreased *glitching*[3] in the logic as more pipeline registers are added. The energy savings are diminished by the squashing associated with misspeculated branches and the previously-discussed overheads in stalling the pipeline.

4.4.1 The Impact of Inter-Stage Forwarding Lines

An important optimization of pipelined architectures is to include forwarding lines between stages to reduce stalls due to RAW hazards. We use SPREE to evaluate the benefits of adding forwarding lines to our pipelined designs. In all pipelines studied in this paper, there is only one pair of stages where forwarding is useful: from the writeback stage (WB) to the first execute stage (EX) (see Figure 12). Since the MIPS ISA can have two source operands (referred to as `rs` and `rt`) per instruction, there are four possible forwarding configurations for each of the pipelines: no forwarding, forwarding to operand `rs`, forwarding to operand `rt`, and forwarding to both `rs` and `rt`.

Figure 14 shows the effects of each forwarding configuration on wall-clock-time and area (note that points in the same series differ only in their amount of forwarding). While there is clearly an area penalty for including forwarding, it is consistently 65 LEs for any one forwarding line, and 100 LEs for two across the three different pipeline depths. In all cases the performance improvement is substantial, with more than 20% speedup for supporting both forwarding lines. An interesting observation is that there is clearly more wall-clock-time savings from one forwarding line than the other: forwarding operand `rs` results in a 12% speedup compared to only 5% for operand rt, while the area costs for each are the same. Also, the inclusion of forwarding did not decrease clock frequency significantly.

[3]Glitching refers to the spurious toggling of gate outputs often due to differing arrival times of the gate inputs.

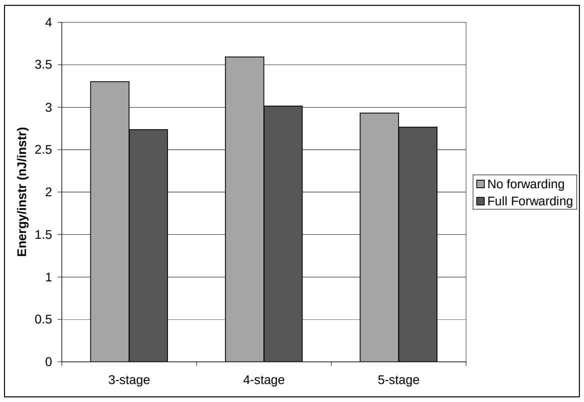

Figure 15: Energy per instruction for three pipelines each with no forwarding, and full forwarding (both `rs` and `rt`).

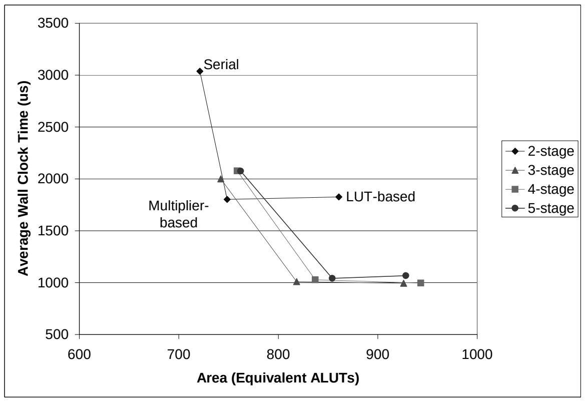

Figure 16: Average wall-clock-time vs area for different pipeline depths and shifters implemented on Stratix II.

The impact of forwarding lines on energy is shown in Figure 15. Energy is decreased by 15% (compared to no forwarding) when forwarding is present for both the `rs` and `rt` operands. This indicates that the energy consumption of the forwarding lines and associated control logic is considerably less than the energy consumed in the pipeline when instructions are stalled (without forwarding lines).

4.5 Device Independence of Exploration

The difference between ASIC and FPGA platforms is large enough that it serves as a one of the motivations for this work. However, FPGA devices differ among themselves: across device families and vendors the resources and routing architecture vary greatly. We have focused on a single FPGA device, the Altera Stratix, to enable efficient synthesis through device-specific optimizations. Our hypothesis, is that in spite of differences in FPGA architecture, the conclusions drawn about soft processor architecture will be transferable between many FPGA families. In the future, we plan to investigate this across a range of different FPGA families; fortunately, this only requires porting the Component Library since the Verilog generated by SPREE is generic. For now, we have migrated to Stratix II [11, 22], which is very similar to Stratix: the main differences are that Stratix II has a more advanced basic logic block, the ALM (Adaptive Logic Module), instead of the LE. One ALM can fit either one or two ALUTs (Adaptive Lookup Tables), which are used to measure area. We observed that there is some variation in the architectural conclusions, but that many of the conclusions still hold. For example, the graph in Figure 10 is nearly identical as seen in Figure 16, except that the LUT-based shifter is smaller in area as expected [10]. It is expected that the variation will be greater when migrating to another vendor.

5. CONCLUSIONS

As FPGA-based soft processors are adapted more widely in embedded processing, we are motivated to understand the architectural trade-offs to maximize their efficiency. We have presented SPREE, an infrastructure for rapidly generating soft processors, and have analyzed the performance, area, and power of a broad space of interesting designs. We have presented a rigorous method for comparing soft processors. We have also compared our generated processors to Altera's NiosII family of commercial soft processors and discovered a generated design which came within 15% of the smallest NiosII variation while outperforming it by 11%, while other generated processors both outperformed and were smaller than the standard NiosII variation.

Our initial exploration included varying support for multiplication, shifter implementations, pipeline depths and organization, and support for inter-stage forwarding. We have found that a multiplier-based shifter is often the best, and that pipelining increases area, decreases energy slightly, but does not always increase performance. We have observed that for a given pipeline depth, there still exist performance/area trade-offs for different placements of the pipeline stages. We have also quantified the effect of inter-stage forwarding, and observed that one operand benefits significantly more from forwarding than the other. These observations have guided us towards interesting future research directions.

5.1 Future Work

In the future, we plan to further validate the soft processor architectural conclusions drawn from SPREE by testing the fidelity of the conclusions across different FPGA devices and by implementing exception support to put SPREE generated soft processors in the same class as other commercial embedded processors. We also plan to broaden our architectural exploration space by including dynamic branch predictors, caches, more aggressive forwarding, VLIW datapaths, and other more advanced architectural features. In addition, we will explore compiler optimizations and hardware/software tradeoffs and include those in SPREE. Finally, we will research and adopt different exploration methods since our exhaustive eploration strategy is not applicable to a more broad architectural space.

6. REFERENCES

[1] Bluespec. http://www.bluespec.com.
[2] Dhrystone 2.1. http://www.freescale.com.
[3] LEON SPARC. http://www.gaisler.com.
[4] MiBench. http://www.eecs.umich.edu/mibench/.
[5] MINT simulation software. http://www.cs.rochester.edu/u/veenstra/.
[6] Modelsim. http://www.model.com.

[7] Nios II. http://www.altera.com/products/ip/processors/nios2.

[8] Opencores.org. http://www.opencores.org/.

[9] RATES - A Reconfigurable Architecture TEsting Suite. http://www.eecg.utoronto.ca/∼lesley/benchmarks/rates/.

[10] Stratix II - Design Building Block Performance. http://www.altera.com/products/devices/stratix2/features/architecture/st2-dbb_perf.html.

[11] Stratix II Device Handbook. http://www.altera.com/literature/lit-stx2.jsp.

[12] XiRisc. http://www.micro.deis.unibo.it/∼campi/XiRisc/.

[13] R. Cliff. Altera Corporation. Private Communication, 2005.

[14] N. Dave and M. Pellauer. UNUM: A General Microprocessor Framework Using Guarded Atomic Actions. In *Workshop on Architecture Research using FPGA Platforms in the 11th International Symposium on High-Performance Computer Architecture*. IEEE Computer Society, 2005.

[15] B. Fagin and J. Erickson. DartMIPS: A Case Study in Quantitative Analysis of Processor Design Tradeoffs Using FPGAs. In *Proceedings of the 1993 International Workshop on Field Programmable Logic and Applications*, Oxford, England, September 1993.

[16] M. Gries. Methods for Evaluating and Covering the Design Space during Early Design Development. Technical Report UCB/ERL M03/32, Electronics Research Lab, University of California at Berkeley, August 2003.

[17] M. Gschwind and D. Maurer. An Extendible MIPS-I Processor in VHDL for Hardware/Software Co-Design. In *Proc. of the European Design Automation Conference EURO-DAC '96 with EURO-VHDL '96*, pages 548–553, Los Alamitos, CA, September 1996. GI, IEEE Computer Society Press.

[18] International Symposium on High-Performance Computer Architecture. *Workshop on Architecture Research using FPGA Platforms*, San Francisco, California, 2005.

[19] M. Itoh, S. Higaki, J. Sato, A. Shiomi, Y. Takeuchi, A. Kitajima, and M. Imai. PEAS-III: An ASIP Design Environment, September 2000.

[20] A. Kejariwal, P. Mishra, J. Astrom, and N. Dutt. HDLGen: Architecture Description Language driven HDL Generation for Pipelined Processors. Technical Report CECS Technical Report 03-04, University of California, Irvine, 2003.

[21] A. Kitajima, M. Itoh, J. Sato, A. Shiomi, Y. Takeuchi, and M. Imai. Effectiveness of the ASIP design system PEAS-III in design of pipelined processors. In *ASP-DAC '01: Proceedings of the 2001 conference on Asia South Pacific design automation*, pages 649–654, New York, NY, USA, 2001. ACM Press.

[22] D. Lewis, E. Ahmed, G. Baeckler, V. Betz, M. Bourgeault, D. Cashman, D. Galloway, M. Hutton, C. Lane, A. Lee, P. Leventis, S. Marquardt, C. McClintock, K. Padalia, B. Pedersen, G. Powell, B. Ratchev, S. Reddy, J. Schleicher, K. Stevens, R. Yuan, R. Cliff, and J. Rose. The Stratix II logic and routing architecture. In *FPGA '05: Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, pages 14–20, New York, NY, USA, 2005. ACM Press.

[23] D. M. Lewis, V. Betz, D. Jefferson, A. Lee, C. Lane, P. Leventis, S. Marquardt, C. McClintock, B. Pedersen, G. Powell, S. Reddy, C. Wysocki, R. Cliff, and J. Rose. The stratix$^{\mathrm{tm}}$ routing and logic architecture. In *FPGA '03: Proceedings of the 2003 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, pages 12–20, 2003.

[24] A. Lodi, M. Toma, and F. Campi. A pipelined configurable gate array for embedded processors. In *FPGA '03: Proceedings of the 2003 ACM/SIGDA eleventh international symposium on Field programmable gate arrays*, pages 21–30, New York, NY, USA, 2003. ACM Press.

[25] P. Metzgen. A high performance 32-bit ALU for programmable logic. In *Proceeding of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*, pages 61–70. ACM Press, 2004.

[26] P. Metzgen. Optimizing a High-Performance 32-bit Processor for Programmable Logic. In *International Symposium on System-on-Chip*, 2004.

[27] K. Morris. Embedded Dilemma. http://www.fpgajournal.com/articles/embedded.htm, November 2003.

[28] S. Padmanabhan, J. Lockwood, R. Cytron, R. Chamberlain, and J. Fritts. Semi-automatic Microarchitecture Configuration of Soft-Core Systems. In *Workshop on Architecture Research using FPGA Platforms in the 11th International Symposium on High-Performance Computer Architecture*, 2005.

[29] F. Plavec. Soft-Core Processor Design. Master's thesis, University of Toronto, 2004.

[30] O. Schliebusch, A. Hoffmann, A. Nohl, G. Braun, and H. Meyr. Architecture Implementation Using the Machine Description Language LISA. In *ASP-DAC '02: Proceedings of the 2002 conference on Asia South Pacific design automation/VLSI Design*, page 239. IEEE Computer Society, 2002.

[31] H. Tomiyama, A. Halambi, P. Grun, N. Dutt, and A. Nicolau. Architecture Description Languages for Systems-on-Chip Design. In *The Sixth Asia Pacific Conference on Chip Design Language*, 1999.

[32] M. Vachharajani, N. Vachharajani, D. A. Penry, J. A. Blome, and D. I. August. Microarchitectural Exploration with Liberty. In *Proceedings of the 35th International Symposium on Microarchitecture*, November 2002.

[33] P. Yiannacouras. SPREE. http://www.eecg.utoronto.ca/∼yiannac/SPREE/.

[34] P. Yiannacouras. The Microarchitecture of FPGA-Based Soft Processors. Master's thesis, University of Toronto, In Prep.