Datapath and ISA Customization for Soft VLIW Processors

Mazen A. R. Saghir, Mohamad El-Majzoub, and Patrick Akl
Department of Electrical and Computer Engineering
American University of Beirut
P.O. Box 11-0236 Riad El-Solh, Beirut, Lebanon
{mazen,mte01,pea02}@aub.edu.lb

Abstract

In this paper we examine the performance and area trade-offs resulting from customizing the datapath and instruction set architecture of a soft VLIW processor. In addition to describing the datapath and instruction set architecture of our processor, we describe a number of microarchitectural optimizations we used to reduce the area of the datapath. We also describe the tools we developed and used to customize, generate, implement, and program the processor. Our experimental results show that datapath and instruction set customization achieve high levels of performance, and that microarchitectural optimizations like selective data forwarding help keep FPGA resource utilization in check.

1. Introduction

FPGAs have long been used to implement specialized processors and application-specific hardware accelerators. With the proliferation of fast, high-density, and feature-rich FPGAs, these versatile devices are being transformed into powerful general-purpose computational platforms. Already, most medium- and high-end FPGAs feature one or more embedded microprocessor cores. This enables designers to partition their applications into software and hardware components by leveraging the tight coupling between the software programmable microprocessors and the logic fabric of the FPGA. Designers can also benefit from a wide selection of soft intellectual property cores that can be implemented in the FPGA and integrated with the microprocessors to form embedded systems on a reconfigurable chip.

Recently, FPGA vendors have begun providing soft microprocessor cores that can be implemented in the logic fabrics of their FPGAs [1][2]. This provides an additional level of flexibility since it allows designers to configure their processors with only the features needed for their applications. In most cases, designers can also create custom machine instructions that can be used to accelerate performance-critical code blocks. So far, commercial soft processors have remained simple, featuring datapaths and instruction set architectures that resemble early RISC processors. However, as the speeds and logic densities of FPGAs increase, and as more on-chip resources like hardware multipliers and memory blocks become available, it is becoming possible to implement more complex processor datapaths and instruction set architectures.

In this paper we present the results of a study we conducted to assess the impact of datapath and ISA customization on the performance and area of soft VLIW processors. In Section 2 we discuss related work and compare it with our own. In Section 3 we describe the datapath organization, instruction set architecture, support for custom computational units, and pipeline structure of our soft VLIW processor. Next, in Section 4, we describe a number of microarchitectural optimizations we used to reduce the area of our processors. In Section 5 we describe our development tools, and in Section 6 we present and discuss our results. Finally, in Section 7, we present our conclusions and describe future work.

2. Related Work

Due to their simple hardware implementations and support for instruction-level parallelism, VLIW architectures have been widely used to implement application-specific instruction processors (ASIPs) for embedded applications [3]. The datapaths and ISAs of these processors are typically customized to achieve the highest performance for a specific application or application domain [4][5]. However, once customized, it is usually very costly and time-consuming to re-customize these VLIW processors since they are often implemented as standalone processors or they are embedded in ASICs.

Over the past few years, configurable and customizable processors have also grown in significance due to the flexibility they offer designers in developing highly special-
trade-offs involved in designing soft processors that issue energy consumption across a set of benchmark applications. Theses tools to measure its area, execution performance, and instances of a simple RISC processor and used FPGA synthesis tools to measure its area, execution performance, and energy consumption across a set of benchmark applications. The results of this study provide several insights into the trade-offs involved in customizing soft processors that issue and execute single instructions. Our work builds upon these results and examines the effects of datapath and ISA customization on the performance and area of our soft VLIW datatypes capable of issuing and executing multiple machine operations.

Our VLIW processor is by no means the first to be implemented using FPGAs. Spyder [10] is an early example of a VLIW processor implemented using several FPGA devices and memory components. Although the logic densities of contemporary FPGAs have increased to the point where it is possible to implement an entire VLIW processor on a single device, many of the implementation techniques used by the designers of Spyder can still be used to reduce the overall area of the processor. In this section, we describe some of these techniques in more detail.

A more recent example of a VLIW processor implemented on an FPGA is the one developed by researchers at the University of Pittsburgh [11][12]. The processor consists of four 32-bit ALUs based on the instruction set architecture of the Altera NIOS-II soft processor [2]. The processor also contains a customizable hardware block that can be used to implement frequently executed loop kernels as custom hardware functions. The ALUs and the hardware block are interconnected through a single, multi-ported, 32 × 32-bit register file. All instructions are 128 bits long and consist of four, 32-bit fields. A profile-driven toolchain that partitions an application program into hardware and software components was also developed. Frequently executed loop kernels are processed by a behavioral synthesis module that synthesizes the code into logic gates that can be implemented on the processor’s hardware block. The Trimaran compiler [13] is also used to generate sequential NIOS-II machine operations, and a VLIW backend is used to schedule parallel operations into long instructions. Loop kernels that have been implemented in hardware are replaced by low-overhead calls to the corresponding hardware functions.

Our processor differs from the Pittsburgh processor in several ways. Our processor is configurable and the VHDL code describing its datapath is automatically generated. This enables us to quickly and easily explore a large architectural space. Our processor also uses heterogeneous functional units, which lead to more specialized datapath implementations. Our processor also uses distributed (clustered) register files whose fewer ports reduces the area and access time of individual register files and simplifies data forwarding [3]. Our processor also provides several alternatives for integrating custom hardware units into the datapath. Finally, our processor uses a retargettable toolchain, which obviates the need for developing a new set of tools every time the architecture changes.

3. Processor Architecture

In this section we describe the datapath, instruction set architecture, and pipeline organization of our processor.

3.1. Datapath organization

Figure 1 shows the datapath of our soft VLIW processor. The datapath can be configured to include any number of 16-, 32-, or 64-bit general-purpose functional units. These units execute a basic set of integer machine operations and include arithmetic and logic units (ALUs), multiply-accumulate units (MACs), address generation units (AGUs), and data memory units (DMUs). AGUs are used to compute data memory addresses while DMUs are used to execute memory-access operations. Every DMU is connected to a single-ported data memory bank that is implemented using an on-chip RAM block. The datapath may also include a number of custom computational units (CCUs), which execute user-defined machine operations that are used to customize and extend the basic ISA. Finally, the datapath includes a single control and branching unit (CTU) that performs data movement, branching, and other control-flow operations.

The datapath also includes up to three distributed register files, each with a configurable number of registers. Distributed register files use fewer read and write ports, and are therefore smaller and faster than a unified, multi-ported register file of the same aggregate size. The distributed register files include a data register file (DRF), an address register file (ARF), and an optional custom register file (CRF). The DRF stores data for the ALUs and MACs while the ARF stores data for the AGUs and supplies memory addresses to the DMUs. A CRF may also be added to the datapath.
3.3. Custom computational units

In our processor, ISA customization is achieved by adding custom machine operations to the base instruction set. These operations are executed on custom computational units (CCUs), which are user-designed combinational logic blocks that realize the custom operations in hardware. CCUs may be pipelined but do not have to be.

Figure 2 shows three different ways for adding a CCU to the datapath. The first (Figure 2(a)) extends the functionality of a regular functional unit by augmenting it with custom logic. This approach is used in some commercial soft processors like the Altera NIOS-II. Custom operations executing on such CCUs are constrained to only operate on a pair of operands, produce a single result, and have a latency of one processor clock cycle. Currently, the only functional units that can be augmented with custom logic are ALUs and AGUs. Another way for adding a CCU to the datapath is to cluster the CCU with other functional units that share a register file (Figure 2(b)). For example, a CCU can be clustered with ALUs and MACs sharing the DRF or AGUs sharing the ARF. This provides a tight coupling between the CCU and the other functional units, which enables the units to share and exchange data efficiently. However, this also requires additional register ports, which increase the size and latency of the register file. Finally, a CCU can be added to the datapath by connecting it to a dedicated CRF (Figure 2(c)). This approach provides a looser coupling between the CCU and other functional units in the datapath, and requires explicit machine operations to move data between the CRF and other register files.

3.4. Pipeline structure

Our base processor is designed around a four-stage pipeline that includes instruction fetch (IF), instruction decode (ID), execute (EX), and write-back (WB) stages.

In the IF stage, a long instruction is fetched from instruction memory. In the ID stage, the long instruction is decoded and its corresponding machine operations are dispatched for execution on the appropriate functional units.
However, branches and other control-flow operations are executed in this stage. Finally, register source operands are read from the corresponding register files and immediate operands are extracted from the instruction register during this stage.

During the execute stage (EX), operations dispatched in the ID stage are executed in parallel on available functional units. Since most operations have a latency of one processor clock cycle, they complete execution in this stage. However, some custom machine operations may require several clock cycles to execute. During this time, other operations can continue executing on other functional units. Although such out-of-order operation completion may lead to contention for register write ports and WAW hazards, these can easily be avoided through careful instruction scheduling. Contention for register write ports can also be eliminated by increasing the number of register write ports.

The results of various operations are written back to the corresponding register files in the WB stage. To enable overlapping the WB and ID stages, all registers are written during the first half of the processor clock cycle and read during the second half. To minimize the effect of RAW data hazards, data forwarding is used to bypass results between the WB and EX stages. The bypassing signal paths and control logic are automatically generated by the datapath generator.

4. Microarchitecture Optimizations

In this section we describe the microarchitectural optimizations we used in our datapath and their implications on the compiler or assembly programmer.

4.1. Hardware multipliers

Contemporary FPGAs provide on-chip hardware multipliers that are faster and smaller alternatives to implementing multipliers using the logic resources of the FPGA. We used hardware multipliers to implement the MAC functional units in our datapath. To support $32 \times 32$ multiplications, each MAC unit uses four hardware multipliers.

Hardware multipliers can also be used to implement fast and area-efficient shifters since shifting a value one bit position to the left is equivalent to multiplying it by two [15]. That is why we also used the hardware multipliers to implement arithmetic and logical shifters for the ALU functional unit. To support 32-bit shift operations, each shifter uses two hardware multipliers.

4.2. RAM blocks

RAM blocks are commonly used in FPGAs to implement a wide range of storage devices [16]. Memory banks and register files implemented using RAM blocks are faster and smaller than those implemented using the flip-flops found in FPGA logic blocks. For example, the Xilinx XC2V2000 Virtex-II FPGA we used for this study contains 56, 18 Kbit, dual-ported RAM blocks that can each be configured to store 32- or 64-bit words. The two ports on each RAM block can also be configured to serve as read, write, or read/write ports.

We used RAM blocks to implement a distributed data memory bank in our processor. Distributed data memory banks are commonly used in programmable DSPs since they provide a fast, low-cost, and high-bandwidth alternative to a large, unified, multiported data memory bank. By connecting every DMU in the datapath to a different RAM block, multiple data words can be accessed simultaneously. However, to benefit from this organization, data must be distributed among the various data banks in a manner that maximizes the exploitation of parallelism [17].

We also used RAM blocks to implement the multi-ported register files used in our datapath. Figure 3 shows how eight RAM blocks can be used to implement a register file with four read and two write ports. The key to supporting multiple register ports is to duplicate data across the various RAM blocks. In this example, the RAM blocks are divided into two banks with the upper four blocks used to implement even-numbered registers and the lower four blocks used to implement odd-numbered registers. Since data is duplicated across blocks of the same bank, multiple values can be read from different RAM blocks simultaneously. The output multiplexers also ensure that each read port can provide data from either register bank. RAM blocks corresponding to the same bank can also be updated with the same value simultaneously. This enables results destined for even and odd numbered registers to be written back simultaneously. To benefit from this register file organization, registers must be allocated in a manner that ensures results of different operations destined for the same register file target different register banks. In general, a register file with $M$ read ports and $N$ write ports requires $M \times N$ RAM blocks distributed across $N$ banks.

4.3. Selective data forwarding

Data forwarding is a well known technique for eliminating RAW data hazards in pipelined processors. In a VLIW datapath, the number of forwarding paths and the complexity of the forwarding logic grow in proportion to the number of functional units used in the datapath. When implemented in FPGAs, the area occupied by forwarding paths, multiplexers, and control logic can grow significantly. To reduce this area, we use selective data forwarding where only the forwarding paths actually needed to support a given application program are maintained. Although this makes for-
warding difficult for applications that have other bypassing needs, dependent operations in these applications can be scheduled in a way that eliminates RAW hazards. If dependencies cannot be eliminated through code scheduling, NOPs can be inserted between dependent operations, which degrades performance. Another solution exploits the field programmability of FPGAs to load processor configurations on a per application basis to benefit from selective data forwarding without affecting execution performance. In this case, the only overhead is that involved in reprogramming the FPGA.

5. Development Tools

Configurable processors require a flexible tool chain that enables designers to quickly configure, generate, implement, and program new processor designs. Figure 4 shows the various tools we developed for our VLIW processor and how they interact with each other. In this section we describe each of these tools.

5.1. Processor configuration file and PCF-Gen

The architectural and organizational parameters of our processor are captured in a processor configuration file (PCF) that is used to retarget both the datapath generator and the assembler. Among the parameters specified in the PCF are the processor’s data width, the number and type of its functional units, the set of operations supported by each functional unit, the sizes of the different register files, and the number and sizes of data memory banks.

The PCF is also used to describe CCUs, which are specified by a number of parameters that include CCU inputs and outputs, the number of CCU pipeline stages (if the CCU is pipelined), and the register file with which a CCU is associated. For each CCU, the PCF also specifies the mnemonics, inputs, outputs, latencies, and VHDL implementations of each of its custom operations. Currently, the VHDL implementation must be developed by the designer. However, we are developing a behavioral synthesis module that will automatically generate the VHDL code for CCUs based on an analysis of specific high-level programming constructs.

Figure 5 shows part of a PCF that describes an ALU that has been augmented with a custom machine operation. PCFs are written in XML, which is easily extensible and simplifies describing an instruction set architecture and a datapath that are continually evolving. XML files are also easy to parse, making it easy for our tools to extract relevant information about the processor.

Finally, PCFs are automatically generated by our processor configuration file generator (PCFGen), which provides a user-friendly graphical interface that greatly simplifies the
way architectural parameters are specified for a target processor.

5.2. DPGen

The datapath generator (DPGen) extracts relevant information from the PCF and generates VHDL code for the corresponding processor datapath. Based on the parameters specified in the PCF, DPGen generates behavioral VHDL code for the instruction decoder, control unit, branching unit, register files, functional units, and pipeline registers. If a CCU is specified in the PCF, DPGen also generates VHDL code for the additional logic and control and data paths needed to integrate the CCU with the rest of the datapath. Once the VHDL code for all datapath components has been generated, DPGen generates a top-level wrapper that instantiates and interconnects the various components. DPGen also generates a testbench that can be used to drive a functional simulation of the datapath.

5.3. Assembler

The instruction width and encoding of our VLIW processor vary with a number of architectural parameters that include the functional units and CCUs used in the datapath, the operations implemented in each functional unit and CCU, the size of the different register files, and the processor data width. To simplify the generation of binary code for such a variable instruction format, we developed a retargetable assembler that extracts relevant information about the processor’s ISA from the PCF and uses the information to generate long machine instructions in a binary format that matches the architecture of the target processor. Once configured, the assembler can translate assembly programs written for the target processor into binary code images that are used by DPGen to initialize the processor’s instruction memory bank.

5.4. ModelSim and ISE

In addition to our tools, we used Mentor Graphics ModelSim 6.0a to perform functional simulations of our processors to both validate their behavior and measure the execution cycle counts for a number of assembly language benchmarks we developed. We also used the Xilinx ISE tools to synthesize and implement the corresponding datapaths in a 190 MHz Xilinx XCV2000-FF896 Virtex-II FPGA. Finally, we used the various reports generated by the Xilinx tools to measure processor clock frequencies and FPGA resource utilization.

6. Results

In this section we present the results of different experiments that illustrate the performance and area tradeoffs resulting from datapath and ISA customization of various soft VLIW datapaths.

Since we still do not have a working high-level C compiler, we developed short assembly-language routines to assess the impact of specific architectural configurations on performance and area. To ensure our results reflect typical design trade-offs, we chose our kernel benchmarks from a range of embedded application domains. Table 1 shows the different kernel benchmarks we developed for this study.

In our results, we report the execution performance of a given benchmark as its wall clock execution time. This is computed as the product of the dynamic cycle count for the benchmark, obtained from the ModelSim behavioral simulation, and the processor clock cycle time, obtained from

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIB100</td>
<td>Computes the first 100 terms of the Fibonacci Series.</td>
</tr>
<tr>
<td>FACT100</td>
<td>Computes 100 factorial.</td>
</tr>
<tr>
<td>FIR256</td>
<td>Implements a 256-tap finite impulse response filter.</td>
</tr>
<tr>
<td>REV32</td>
<td>Implements a 32-bit reverse function.</td>
</tr>
<tr>
<td>SCB</td>
<td>Implements a bit scrambling function [18].</td>
</tr>
<tr>
<td>SAD16</td>
<td>Implements the $16 \times 1$ sum of absolute differences function [19].</td>
</tr>
</tbody>
</table>

Table 1. Kernel benchmarks.
the Xilinx XST synthesis report. We also report the area for a given datapath implementation in terms of FPGA slices and the number of RAM blocks and hardware multipliers used. These numbers are also obtained from the Xilinx XST synthesis report. Although slices are a crude measure of circuit area, they are, along with the number of RAM blocks and hardware multipliers used, an accurate measure of FPGA device utilization.

### 6.1. Base processor vs. Xilinx MicroBlaze

In our first experiment, we compared the performance and area of our base processor with the Xilinx MicroBlaze soft processor. Tables 2 and 3 summarize the performance and area of different implementations of our base processor (BP and BP+MAC) with those of the MicroBlaze (MB and MB+MUL) when executing two general-purpose computational kernels. To avoid giving our VLIW processor an unfair performance advantage, we made sure both kernels did not exploit any parallelism.

Our results show that our base processor is 1.42 to 2.23 times faster than the MicroBlaze when executing the FIB100 benchmark and 2.11 to 3.23 times faster when executing the FACT100 benchmark. The higher performance of our processor is mainly due to lower cycle counts, which are achieved as a result of low-overhead looping operations. These operations are part of the base instruction set and are used to eliminate test-and-branch operations in conditional loops. The higher performance is also due to faster processor clock frequencies, particularly when selective and no data forwarding are used (cf. columns labeled SEL and NO, respectively), which are due to the deeper pipeline of our processor. When full forwarding is used (cf. columns labeled FULL), longer delays in the datapath cause our processor to achieve clock frequencies that are comparable to the MicroBlaze. When no data forwarding is used, the cycle counts in our processors increase significantly due to additional NOP operations that must be inserted in the code to eliminate RAW data hazards.

Our results also show that our base processor uses 0.88 to 1.47 times the slices used by the MicroBlaze. When we add a multiplier unit to both processors, our processor uses 0.97 to 1.95 times the slices used by the MicroBlaze. When full forwarding is used, our processors use more slices due to additional forwarding logic and data paths. On the other hand, when selective or no data forwarding are used, our processors use fewer slices. However, since the MicroBlaze uses additional logic to implement bus interfaces and support interrupts, these results are slightly skewed in our favor. Finally, since our processors use distributed register files with a large number of read ports they use significantly more RAM blocks than the MicroBlaze.

### 6.2. Supporting ILP through datapath customization

In our second experiment, we studied the effects of customizing the datapath to support instruction-level parallelism. For this experiment, we used the FIR256 kernel because it exhibits high levels of parallelism, and we examined three processor implementations with progressively increasing support for parallelism: the base processor augmented with a MAC unit (BP+MAC); BP+MAC with a two-write-port DRF (BP+MAC+DRF2W); and a custom datapath that includes 1 ALU, 1 MAC, 2 AGUs, 2 DMUs, a three-write-port DRF, and a two-write-port ARF (FIR+DRF3W+ARF2W). Table 4 summaries our results.

Our results show that as support for parallelism increases in the datapath, performance increases accordingly. For example, with full forwarding, BP+MAC+DRF2W and FIR+DRF3W+ARF2W are 1.49 and 2.55 times faster than BP+MAC, respectively. This is mainly due to lower cycle counts, which are a direct consequence of exploiting higher levels of parallelism. On the other hand, supporting higher levels of parallelism increases datapath complexity, which reduces processor clock frequency. Still, the net result is an improvement in performance. Performance is fur-
Our results also show that supporting parallelism increases the number of FPGA resources used to implement the datapath. For example, when full forwarding is used, BP+MAC+DRF2W and FIR+DRF3W+ARF2W use 1.12 and 2.16 times as many slices as BP+MAC, respectively. Although the number of slices can be reduced significantly by using selective or no data forwarding, the savings must be weighed against the resulting levels of performance. When no data forwarding is used, the savings are achieved at the expense of degraded performance. Finally, the number of RAM blocks used in BP+MAC+DRF2W and FIR+DRF3W+ARF2W is 1.57 and 3.21 times the number used in BP+MAC, respectively. This is due to the increased number of read and write ports in both register files.

**6.3. Adding a custom operation to the AGU**

In our third experiment, we studied the effect of adding a bit-reverse operation to the base instruction set by augmenting the AGU with custom logic. We used the REV32 benchmark to compare the performance and area of our base processor (BP) with another implementation of BP that uses an augmented AGU (BP+REVAGU). Table 5 summarizes our results, which show that the custom operation in BP+REVAGU is 50.00 to 76.57 times faster than a software implementation executing on BP when full and no data forwarding are used, respectively. The higher performance is due to the significantly lower cycle count achieved by using the custom operation, and, in the case of no data forwarding, the higher processor clock frequency. Our results also show that the impact of adding the custom operation on FPGA resources is negligible. This is due to the simple implementation of the bit-reverse operation, which only requires a few additional slices. It is also due to the microarchitectural constraints on adding custom logic to a functional unit, which do not affect the number of register ports and keep the number of RAM blocks in the datapath constant.

**6.4. Adding multi-cycle CCUs to the datapath**

In our fourth experiment, we studied the effects of customizing the instruction set by adding two, non-pipelined, multi-cycle CCUs to the datapath. The SCB CCU implements a bit scrambling function used in WLAN OFDM modems [18]. It uses two 32-bit inputs, produces a single 32-bit output, and has a latency of eight clock cycles. On the other hand, the SAD16 CCU implements the 16 × 1 sum-of-absolute-differences function, which is used for motion estimation in MPEG4 video encoders [19]. It uses eight 32-bit inputs, produces a 32-bit output, and has a latency of four clock cycles. For each of these CCUs we created two processor implementations: one that connects the CCU to the DRF (BP+DRFSCB and BP+DRFSAD16) and another that connects the CCU to a dedicated 32 × 32 CRF (BP+CRFSCB and BP+CRFSAD16). Furthermore, for each implementation, we considered the cases where full and no data forwarding were used, respectively. We then compared these processors to our base processor when executing the SCB and SAD16 kernels, which are software implementations of the corresponding CCU functions. Table 6 summarizes our results.

Our results show that when connected to the DRF, the SCB CCU is 11.28 to 19.53 times faster than the BP, while
the SAD16 CCU is 23.61 to 43.06 times faster than the BP when full and no data forwarding are used, respectively. This is mainly due to the significantly lower cycle counts achieved when implementing the CCU functions in hardware. However, it is worth noting that in most cases the additional complexity resulting from introducing CCUs to the datapath also decreases the processor clock frequency. Still, the net effect is an improvement in performance. Our results also show that the number of FPGA slices used by BP+DRFSCB and BP+DRFSAD16 are 1.37 and 1.83 times those used by the BP when full forwarding is used, and that they consume more RAM blocks. This is due to the additional logic required to implement the CCUs, additional forwarding paths and logic, and the new DRF and the CRF, and the lower processor clock frequencies resulting from introducing the CCUs to the datapath. Our results also show that the number of FPGA slices used by BP+CRFSAD16 and BP+CRFSAD16 are 1.38 and 1.52 times those used by the BP when full forwarding is used, and that they also consume more RAM blocks. This, again, is due to the additional logic required to implement the CCUs, additional forwarding paths and logic, and the new CRF. When no data forwarding is used, BP+CRFSCB and BP+CRFSAD16 also use significantly less FPGA slices, which still amount to 1.45 and 1.72 times the number of slices used by the BP. The number of RAM blocks used by each processor also remains the same.

### 7. Conclusions and Future Work

In this paper, we described the architecture and microarchitecture of a customizable soft VLIW processor. We also described the tools we developed to customize, generate, and program this processor. We also examined the performance and area trade-offs achieved by customizing the processor’s datapath and instruction set architecture. The following points summarize our main results and conclusions.

1. As with hard processors, careful design of the instruction-set architecture and the pipeline has a significant impact on performance. Our results show that, without exploiting ILP, our soft processor is 1.42 to 3.23 times faster than the Xilinx MicroBlaze processor. This is mainly due to special machine operations in the instruction set (e.g. low-overhead looping) that reduce execution cycle counts and faster clock frequencies achieved using a deeper pipeline. Using on-chip resources, such as hardware multipliers and block memories, and implementing microarchitectural optimizations such as selective data forwarding also reduce the complexity and area of the datapath and contribute to increasing the processor clock frequency.

2. Customizing the datapath by duplicating functional units and increasing the number of register ports to support higher levels of parallelism achieves faster execution performance, particularly when coupled with selective data forwarding. However, this is achieved at the cost of using more logic resources and RAM blocks. Although resource utilization can be reduced by using selective or no data forwarding, these savings must be weighed against the levels of performance achieved.

3. Augmenting a functional unit with custom logic achieves significant improvements in performance while having little impact on FPGA resource utilization. The improvement in performance is mainly
due to implementing the CCU function in hardware while the limited resource utilization is due to the microarchitectural constraints on adding custom logic to a functional unit. Obviously, achievable levels of performance and actual resource utilization vary with the complexity of the custom logic.

4. Adding a CCU to the datapath achieves high levels of performance while using significant FPGA resources. The higher performance is chiefly due to implementing CCU functions in hardware, while the additional resource utilization is due to CCU logic, forwarding paths and logic, and additional registers or register ports. In general, connecting the CCU to the data register file achieves better performance than connecting it to a dedicated custom register file due to the lower overhead involved in transferring data between different register files. Again, the achievable levels of performance and actual resource utilization vary with the complexity of the CCU and the number of its inputs and outputs.

Now that we have shown that customizing the datapath and ISA of a soft VLIW processor achieves significant improvements in performance, and that using on-chip components and implementing microarchitectural optimizations reduces the utilization of FPGA resources, we are focusing on developing tools that can automate these tasks. We have started porting the GCC compiler to our base instruction set architecture, and are in the process of implementing standard optimizations for exploiting instruction-level parallelism – crucial for any VLIW processor – along with other, architecture-specific optimizations for supporting distributed data memory banks, banked register files, and selective data forwarding. Our compiler will also include a behavioral synthesis module that can create custom machine instructions by transforming specific programming constructs into hardware functions and leveraging available FPGA resources to implement these functions in customized CCUs. To guide our compiler, we are also developing performance and area models that can be used to make quick assessments about suitable design trade-offs.

References