FlexCore: Implementing an Exposed Datapath Processor

Magnus Själander
Florida State University
Computer Science Department
Tallahassee, FL 32306-4530, USA
msjaelander@fsu.edu

Per Larsson-Edefors
Chalmers University of Technology
Computer Science and Engineering Department
412 96 Gothenburg, Sweden
perla@chalmers.se

***Abstract*—The FlexCore processor is the resulting implementation of an exposed datapath approach conceptualized in the FlexSoC programme. By way of a crossbar switch interconnect, all execution units in a FlexCore datapath can potentially communicate, allowing the inherent hardware parallelism to be utilized. This interconnect enables configuration of a datapath to match an application domain, for example, by way of datapath accelerators. The baseline FlexCore is a general-purpose processor (GPP) and since all FlexCore configurations are extensions to the baseline, they offer GPP functionality as complement to the domain-specific functionality. This paper gives an overview of the implementation of complete FlexCore processors, accompanied with discussions on datapath interconnects, datapath extensions and instruction decompression.**

I. Introduction

Electronic systems are constrained in many dimensions: They must provide high enough performance under strict constraints on the energy expended, they must retain the flexibility associated with processors, and they must have support from an advanced development environment in which software can be developed and the design space can be explored. Processors are vital to electronic systems as they offer post-fabrication flexibility, but the generality of a processor is associated with overheads that cause inefficiencies [1].

The FlexCore processor was conceived within the FlexSoC programme, with the goal of combining high performance, energy efficiency, and flexibility [2]. In recent years, a complete HW/SW development environment has been developed [3] for FlexCore in order to provide a solid implementation methodology. The transport triggered architecture (TTA) [4] is another exposed datapath architecture. The TTA takes a more interconnect-centric approach where the compiler has control over a set of internal transport buses. The buses are used to transfer data between the units within the datapath and operations are performed on the data as a side effect to where the data is moved. The no instruction set computer (NISC) [5] and static pipelining [6] approach bear the closest resemblance to FlexCore in both concept and methodology. While FlexCore is a template of a processor architecture that can support a certain level of general-purpose functionality, NISC mainly targets narrower application domains. Furthermore, in NISC, information about the datapath interconnect is not considered at a fine-grained level but, reminiscent of high-level synthesis, a datapath and the resulting interconnect are generated from the application control-flow/data-graph and profiling information from application execution. Referring to the fact that the compiler statically determines the pipelining of each processor portion, the static pipelining approach explicitly targets general-purpose computing. This approach is not design-time configurable to meet the needs of a particular application domain, so datapath variations are not considered.

The FlexSoC research programme was supported by grants from the Swedish Foundation for Strategic Research (SSF) and the Swedish Research Council (VR).

Based on an exposed datapath, FlexCore is not a rigid architecture, but rather a template with the following key features:

- Exposed datapath: The datapath control signals are exposed to the compiler as a wide control word, called the native instruction set architecture (N-ISA). The exposed datapath allows the compiler to harness the datapath parallelism more efficiently than in a fixed ISA architecture.
- Rich interconnect: Since the ISA is not fixed, a FlexCore datapath can contain an arbitrary set of datapath units *and* an interconnect that allows for an arbitrary set of communication paths. An extensive datapath interconnect allows the compiler to work under the assumption that any two datapath units can communicate, thereby unleashing all inherent parallelism.
- Instruction decompression: An on-the-fly instruction decompressor translates shorter instructions stored in memory—the application-specific ISAs (AS-ISAs)—to N-ISA (Fig. 1). The efficacy of this scheme is contingent on the hypothesis that the AS-ISAs can be more densely coded than the N-ISA, that is, that the task running on the processor only requires a subset of the datapath resources for its execution at any particular time.
- Datapath extensions: Thanks to the exposed datapath and the rich interconnect, it is straightforward from a hardware point of view to extend the basic datapath with more execution units.

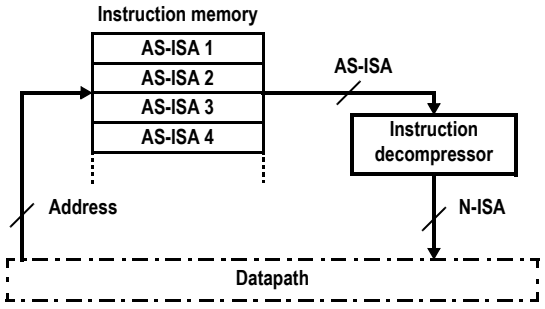

Fig. 1. Conceptual view of instruction decompression.

The following sections present parts of the FlexCore architecture (Sec. II), and the HW/SW development environment (Sec. III). Sec. IV describes the impact the interconnect configuration has on processor performance and energy. Sec. V presents a validation of a complete FlexCore system as well as evaluations of performance and energy for the combination of an exposed datapath and an instruction decompressor. Sec. VI describes studies done on datapath extensions, using accelerators, to reduce total processor energy.

II. Architecture

We will briefly review the FlexCore architecture, with emphasis on the datapath template, the instruction decompression, and the need for instruction scheduling across basic blocks.

978-1-4799-0103-6/13/$31.00 ©2013 IEEE

A. Datapath

The datapath template of FlexCore assumes an interconnect that potentially supports communication between any two datapath units. As shown in Fig. 2, an output of a particular unit (an output port) is connected to a register. The output of the register is routed to several input multiplexors, each driving an input of a unit (an input port).

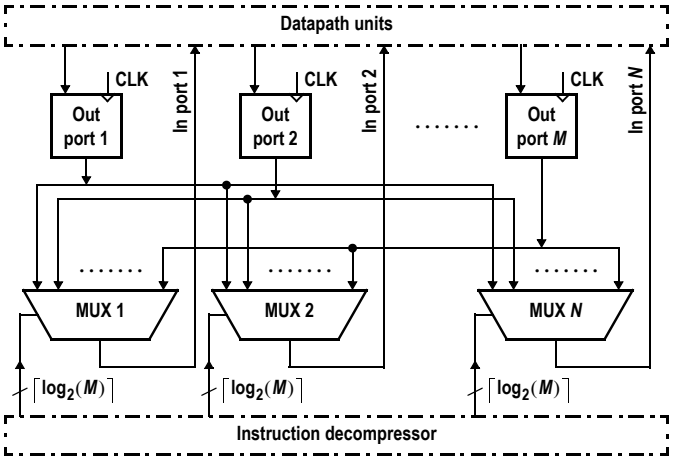

Fig. 2. In its most complex configuration the datapath interconnect acts as a crossbar switch, supporting communication from each of M output ports to each of N input ports.

Assuming M output ports, N input ports, and $N\lceil \log_2 M \rceil$ control signals, the interconnect template can at most support $N \cdot M$ communication paths. This is called the *full* interconnect configuration. While it supports a very high degree of parallelism, many of the supported paths are never used during application execution. An efficient configuration includes only the interconnect links that provide useful parallelism and, hence, reduce application execution time. Communication paths that are never used should clearly not be implemented, in order to limit the overhead, see Sec. IV.

As will be described in Sec. VI, the FlexCore datapath can be extended with several execution units, but the basic datapath consists of a very rudimentary set of units (Fig. 3). The two buffers are included to make this FlexCore datapath identical—with respect to datapath units—with the MIPS R2K datapath [7], a simple and well-known in-order five-stage pipeline architecture.

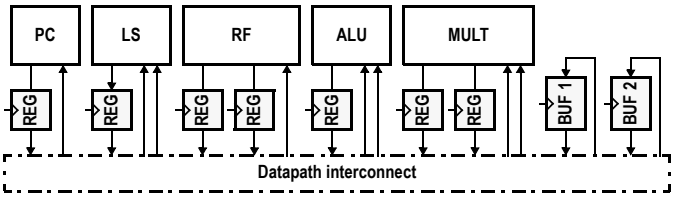

Fig. 3. Datapath consisting of a program counter (PC), a load/store unit (LS), a 32-entry register file (RF), an arithmetic logic unit (ALU), two buffers and an integer multiplier (MULT). The shown datapath is referred to as the *baseline*.

The datapath operations can be expressed as register-transfer notations (RTN), where an operation can be performed on output port registers of the various datapath units. The output port from where a value is being read represents the address of the interconnect multiplexor, while the control signals to a specific datapath unit represent the operation. Concatenating the RTN instructions of all datapath units for one clock cycle creates one N-ISA.

B. Instruction Decompressor

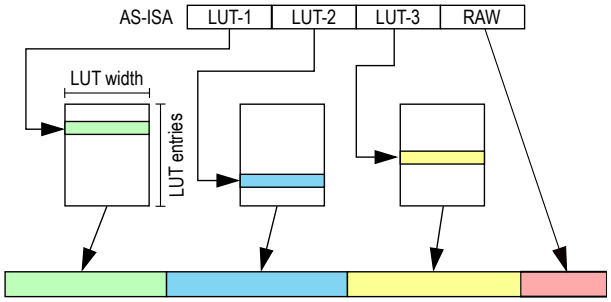

Fig. 4. Illustration of AS-ISA to N-ISA decompression with look-up tables.

Fig. 4 shows an instruction decompressing technique based on partitioned look-up tables (LUTs) that was suggested in an earlier work [8], but for the implementation that will be evaluated in Sec. V-A the technique has been further developed into a process of several steps: **1.** Each effect of the N-ISA (e.g., ALU opcode, register file read/write addresses, and interconnect addresses) is specified in a configuration file that is automatically generated for each FlexCore configuration. Effects can be manually grouped or marked to not be stored in a LUT, so-called RAW effects. This helps to reduce the search space as many small effects can otherwise cause a large number of potential solutions. RAW effects are simply passed directly from the AS-ISA to the N-ISA and are not considered by the following steps. **2.** The process starts by combining N-ISA effects into all possible permutations within configurable bit-width limits of a LUT. The bit-width limits are enforced to further reduce the total number of possible LUTs and, thus, the search space. **3.** All N-ISA instructions of an application are inspected to identify all unique bit-patterns for each LUT. For each LUT, the maximum number of entries required across all applications are also recorded. **4.** Given the set of all LUTs, all permutations of these that form a valid N-ISA are created; these are the possible solutions. Each bit of the N-ISA is represented in one and only one LUT. **5.** All possible solutions are processed to calculate the width of the compressed AS-ISA and the total bits for the LUTs, across all applications. We impose the requirement that AS-ISAs should be equal or shorter than 64 bits, and then we find the least number of LUT bits. **6.** Given this configuration, AS-ISAs can be generated by converting bit-patterns in the N-ISA to the corresponding address in a LUT.

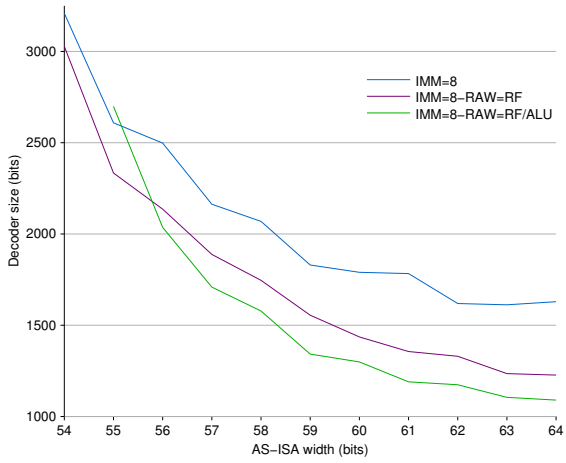

Fig. 5. Decompressor size relative AS-ISA size for different configurations.

Fig. 5 shows the required number of LUT bits for different AS-ISA widths, when different RAW effects are selected. The shown results are for a FlexCore with a 60-link interconnect identified in our earlier work [3], see Sec. IV. The results are generated using the EEMBC autocorrelation (Autcor), convolutional encoder (Conven), fast Fourier transform (FFT), and Viterbi benchmark [9]. All results are for a decompressor in which the 32-bit immediate has been split in an 8-bit RAW part and a 24-bit part that is stored in a LUT (IMM=8). For the data set called RAW=RF, the register file read addresses are represented as RAW effects that are not stored in any LUT. The data set called RAW=RF/ALU has, in addition, the interconnect addresses for the two input operands to the ALU represented as RAW effects. The explanation that the configurations with RAW effects yield a smaller decompressor size is that the selected RAW effects have a high entropy as the register file and ALU are the datapath units that have the highest utilization. High entropy effects are difficult to compress and, thus, require many LUT bits to be stored. The figure shows that the LUT size levels off at around 64 bits, not yielding any improvement in decompressor size for larger AS-ISA sizes. The smallest AS-ISA obtained is 54 bits wide, requiring almost three times the LUT bit count of the smallest decompressor configuration.

During the execution of one application, the LUT bit-pattern can be changed by using certain reload instructions, leading to a reduced decompressor size. However, due to lack of compiler support for reload instructions, the results in Fig. 5 are limited to the decompressor being configured once per application.

C. Inter-Block Scheduling

One of the intrinsic disadvantages of exposed datapaths is that an instruction only controls the datapath in the current cycle. In contrast, in conventional pipelined general-purpose processors, control signals are delayed by pipeline stages and follow a hardcoded pipeline path, hence instructions from different basic blocks can exist inside the pipeline simultaneously.

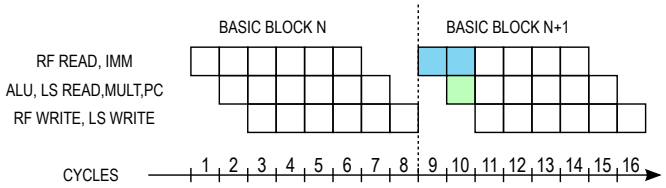

Fig. 6. Independently scheduled basic blocks.

When considering N-ISA scheduling for a single basic block, the first cycle (Fig. 6) is limited to acquire data (e.g., through an RF read or immediate value). Only in the second cycle, data are available to perform operations upon. Similarly, if a single basic block is considered when scheduling the last cycle (cycle 8 in Fig. 6), the only operations that can be performed are an RF write and a store to memory. To perform any other operation would have no effect as its resulting data would not be consumed by any other instruction. The independent scheduling of two consecutive basic blocks, and the inefficiency that this causes, is illustrated in Fig. 6.

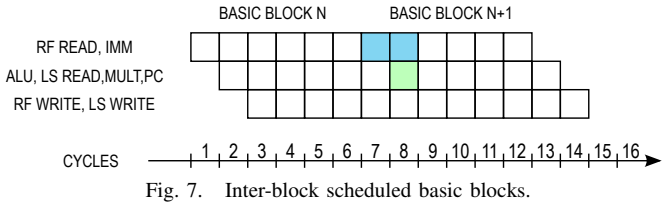

Fig. 7. Inter-block scheduled basic blocks.

In order to improve the scheduling of basic blocks, it is desirable to merge instructions from one block with all its predecessor blocks. Fig. 7 shows when the two first instructions of block $N+1$ have been merged with the two last instructions of block N. This type of inter-block optimization can have a large impact on the cycle count, but there are limitations: Due to resource conflicts, it is not always possible to move instructions to the end of preceding basic blocks. Furthermore, this optimization can not be performed if the branch or jump target is not statically known. While the environment in Sec. III currently does not support inter-block optimization, the evaluation in Sec. V-A uses optimized schedules.

III. Processor Development Environment

In designing a FlexCore, one datapath configuration is initially assumed. Subsequently the compiler creates a schedule and the simulator generates different statistics. As part of the design space exploration phase, the designer may iteratively want to apply changes to the configuration, to improve different parameters such as execution time, power per cycle, or timing. When the proper configuration has been identified, implementation commences.

The processor development environment consists of a complex toolchain ranging from software development to circuit implementation. Since some tools have been developed within the research group, some are commercially available and some publicly available, a system of hierarchical makefiles that utilizes scripts greatly helps in the development process. The development environment shown in Fig. 8 has a software toolchain, with processor configuration, compilation, simulation and design space exploration tools, and a hardware toolchain for the implementation of a complete processor system, including data and instruction caches, and interface logic.

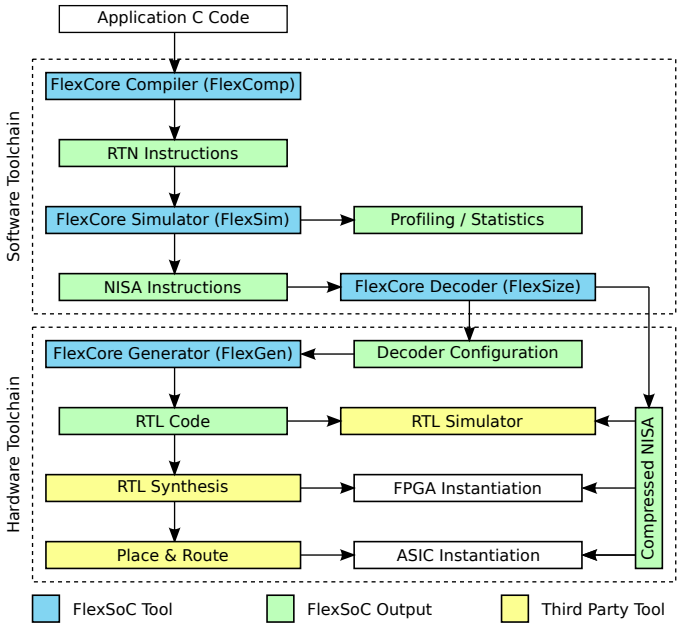

Fig. 8. Organization of the HW/SW development environment.

The software toolchain handles the configuration and software verification of a FlexCore processor [3]. The compiler generates static schedules of FlexCore assembly instructions (the RTN instructions) for a particular datapath configuration. The compiler generates RTN instructions that can be simulated in the FlexSim simulator. The simulator also acts as the assembler to generate the N-ISA code that is used for VHDL simulation and as input to our hardware platform.

The hardware toolchain has a frontend that generates a top-level VHDL module and an associated testbench for a particular FlexCore configuration. All datapath units (e.g., the ALU) have been implemented and verified in advance, and their RTL code blocks reside in a code repository. The generated VHDL code can be synthesized to FPGAs for functional validation, or placed and routed in an ASIC technology for accurate timing, power, and area estimation.

With the exception of third-party, commercial tools, the complete development environment can be downloaded from the FlexSoC website [10]. Also, an open-source VHDL model of a level-1 cache system [11] was recently made available on the FlexSoC website.

IV. Datapath Interconnect Design Exploration

For the baseline FlexCore datapath configuration (Fig. 3), the full interconnect configuration for $M = 9$ output and $N = 10$ input ports supports 90 communication paths (Sec. II-A). Since the extreme level of parallelism that it offers cannot be exploited, the use of the full interconnect creates a significant area and power overhead, in terms of large multiplexors, many wires, and excessive number of N-ISA bits.

a) FFT evaluation: Before the toolchain described in Sec. III was completed, a manual schedule was developed for the EEMBC FFT benchmark to ascertain what could be an energy-efficient domain-specific interconnect configuration. While the power overhead of the FlexCore with a full interconnect was 23% as compared to the reference processor, it turns out that the overhead of the 42-link interconnect configuration that is tailored to the FFT application was only 4% [12]. Since the execution time was significantly reduced, the total energy gain was 29%. It should be noted that among the 42 communication paths selected, we made sure to have support for all general-purpose paths that exist in the reference processor.

b) 60-link configuration: Although it always can execute general-purpose code, typically a FlexCore processor is optimized with respect to an application domain. With the help of the development environment outlined in Sec. III, it is possible to explore the design space and identify an interconnect that can service several different application domains. Using such an interconnect somewhat defeats the purpose of domain customization, but it makes for an interesting study on limitations. A 60-link interconnect was shown to be the most energy-efficient alternative when considering nine complete EEMBC benchmarks [3]. Since there is a large variation in the computing characteristics of the benchmarks, we clearly no longer tap the strength of FlexCore, however, still we can identify a 15% cycle count reduction and a 17% energy reduction in comparison to the reference. We also note that since the benchmarks are diverse, the performance and energy efficiency of different interconnect configurations, except the one of the reference, do not vary significantly.

c) Interconnect optimizer: Based on the environment in Sec. III, an interconnect optimizer was developed [13]. The optimizer invokes the toolchain to analyze the selected interconnect configuration. The resulting energy is considered and based on a genetic algorithm the optimizer makes updates to the interconnect configuration and reruns the toolchain. Preserving general-purpose functionality of the baseline datapath is a key FlexCore feature. However, to study limitations, the optimizer was used for the EEMBC Autcor benchmark to identify an optimal interconnect configuration of 51 links that offers a further 29% improvement in energy efficiency over the best configuration that was compliant with the general-purpose datapath reference.

It should be noted that all evaluations in this section were based on the datapath only, thus, the impact of the instruction decompressor needs to be considered. The following section will address more or less complete FlexCore systems.

V. Implementation, Verification and Evaluation

This section deals with ASIC and FPGA implementations of the baseline FlexCore. The first subsection presents an ASIC evaluation of the performance and energy efficiency of the FlexCore datapath and decompressor, while the other subsection deals with the functional validation of a complete FlexCore processor system, including caches, on an FPGA.

For both evaluations, FlexCore is assessed using the processor that served as an inspiration for the baseline implementation as reference, that is, MIPS R2K (see Sec. II-A). The two processors use identical datapaths units, but different interconnects; MIPS R2K has 33 links [2], while FlexCore uses the 60-link interconnect identified to yield the highest datapath energy efficiency for nine EEMBC benchmarks (see Sec. IV).

A. FlexCore Processor Netlist Evaluation

We implemented the datapath and the instruction decompressor using Synopsys Design Compiler and PrimeTime for a commercial 65-nm cell library. Similarly, a MIPS R2K processor was implemented [14]. Netlists were generated for a common timing constraint of 2.2 ns, at 125°C, 1.1 V and the worst-case process corner. Since the instruction decoder of the MIPS R2K reference is less complex than the implemented FlexCore instruction decompressor (Sec. II-B), the area of the MIPS processor becomes 31% smaller. Due to compiler restrictions, the LUTs are loaded only once per applications. However, the LUT implementation supports reloading of individual entries and uses clock gating to reduce power (Fig. 9).

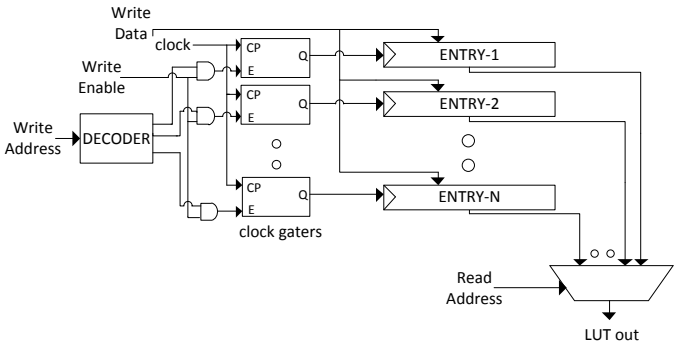

Fig. 9. LUT implementation.

Three different EEMBC benchmarks were used for this evaluation. Accounting for LUT load cycles, still the FlexCore is 35%, 30%, and 18% faster than MIPS for Autcor, FFT, and Conven, respectively [14]. The result for Conven is indicative of the fact that the C code of this benchmark has short loops and little instruction-level parallelism to offer. Power and energy values were obtained at the nominal process corner, using 25°C and 1.2 V, by annotating the application simulation's switching activity files to the processor netlists. This evaluation showed that the FlexCore expends 24%, 12%, and 6% less energy than MIPS for Autcor, FFT, and Conven, respectively [14].

B. FlexCore System Validation

Fig. 10 shows a complete FlexCore system—to the right of the synchronizer—and the testbed—to the left of the synchronizer. To architect a system at minimal effort, IP components were used as far as possible. A number of the system components, like the AMBA AHB bus [15], were drawn from the GRLIB IP library [16].

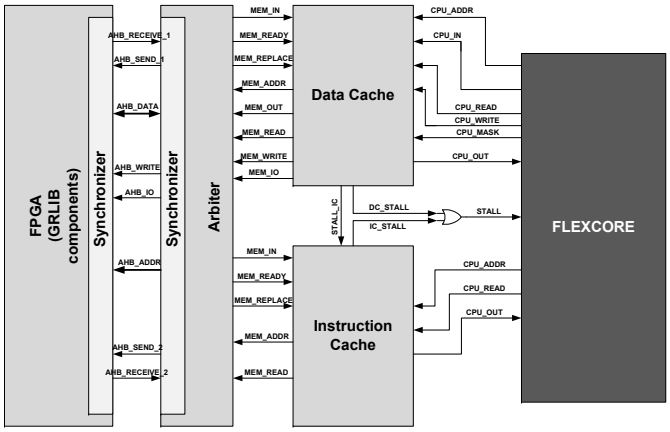

Fig. 10. Block diagram of the FPGA FlexCore system.

To validate the function of the system, we used a bus interface arbiter to test an FPGA version of the FlexCore processor when embedded inside the FPGA testbed. The bus interface implements a handshaking protocol to ensure reliable data transfers. The data bus at the interface is responsible for data directed to both the instruction and data cache, and also outbound data from the data cache. The bus interface logic on either side handles conflict resolution when necessary. Due to the restrictions imposed by the interface and dependencies between the caches, the data cache gets priority in conflict resolution and can stall the instruction cache.

The testbed was an ML-402 FPGA board that has 32 MB of onboard memory coupled to a Xilinx Virtex-4 SX-35 FPGA, which is large enough to fit the complete design. The FPGA version of the FlexCore processor was thoroughly tested for functionality. The complete FlexCore and the GRLIB components of the testbed together use 14,733 out of the available 30,720 4-input LUTs, 42 kB memory, and 3,952 out of the available 30,720 slice registers. A host of EEMBC applications—Autcor, FFT, bit manipulation, etc.—were tested and the FPGA FlexCore ran them all successfully to completion, thus, validating the functionality of the processor system.

VI. Datapath Accelerators

One potential benefit of exposing the datapath to the compiler is that special-purpose datapath hardware can be added to efficiently support the needs of an application domain. Assume there are recurring pieces of code that are too complex to be efficiently executed in basic execution units such as ALUs. In this situation we may extend the datapath by adding more execution units—*datapath accelerators*—to the FlexCore datapath. The energy reductions enabled by these accelerators are due to three reasons: 1) they speed up execution and shorten the time during which the processor is dissipating power, 2) they localize communication avoiding the switched capacitance on long buses, and 3) they use, to some extent, local memory that reduces datapath interconnect transfers.

Integrating an accelerator entails adding datapath circuits, extending the interconnect, and adding N-ISA control bits. Thus, the accelerated datapath will be larger and more power dissipating than the baseline datapath. Clearly, the overhead of accelerator integration has to be minimized. The key to energy efficiency is that the accelerators are used frequently enough to more than compensate for the overhead they present to the system. We use the term *lightweight* to emphasize that the accelerator circuits must be of a very limited complexity and size to be useful for integration in an exposed datapath. Still, constrained by complexity, as much as possible the accelerators should be *versatile* in the sense that they can be widely used.

Using the FlexCore framework, several projects have addressed the implementation and integration of lightweight and versatile accelerators in exposed datapaths, and the design tradeoffs involved in using such accelerators. In Sec. VI-A we describe an accelerator that can switch between the default 32-bit operation and a narrow-width mode in which two 16-bit operations are done in parallel and without any interference between the operations. The second accelerator in this review (Sec. VI-B) has a relatively simple core functionality. This allows for the use of several parallel circuits, each of which supports one standard of the function and which can be invoked by the compiler. Third, in Sec. VI-C, we target an application that has such challenging memory requirements that the algorithmic kernel must be divided into one accelerator part and one general-purpose part. In Sec. VI-D, finally, we show how the expressiveness of the datapath control word can be used to shut off unused portions of execution units.

A. Double-Throughput Acceleration

As far as DSP filter applications, these are often limited to a 16-bit dynamic range and thus the 32-bit datapath is not fully utilized. To more efficiently use the datapath resources in an exposed datapath architecture, a 32-bit multiply-accumulate (MAC) accelerator that optionally can execute two independent 16-bit MAC operations simultaneously was introduced [17]. This *double-throughput MAC* (DTMAC) unit has a circuit complexity similar to a conventional 32-bit multiplier. Since the accumulated results remain inside the accelerator, the DTMAC unit is a power-efficient solution as there is no need for datapath communication to handle the accumulation.

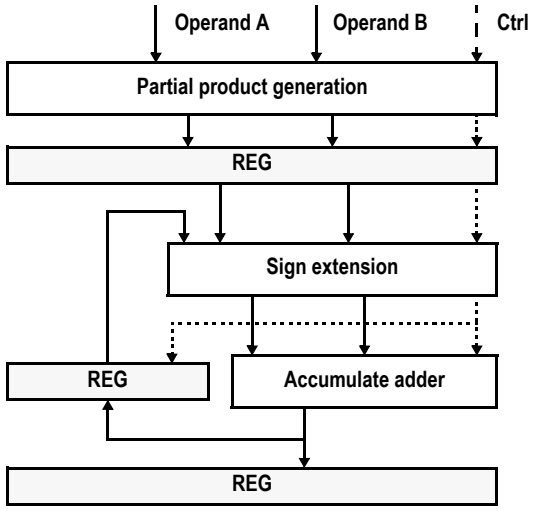

Fig. 11. Simplified block schematic of the DTMAC unit. A new way to handle MAC sign extension enables the use of only one adder (the accumulate adder), instead of using one each in the multiplier and the accumulator which is the conventional way. The control signal (Ctrl) controls what mode is used; for example, full 32-bit mode or 2x16-bit double-throughput mode.

Since the DTMAC unit also can handle multiplications, the multiplier in the baseline datapath in Fig. 3 can be replaced with a DTMAC unit. The datapath area overhead of this replacement is less than 2%. Since the interconnect ports are not affected by the replacement, the overall overhead is very limited.

Table I shows evaluation results for three different processor datapaths (no instruction decompressor is considered here): The reference general-purpose processor (GPP) datapath uses a FlexCore that is restricted to the MIPS R2K interconnect, while the two FlexCore datapaths both use full interconnect configuration, see Sec. IV.

TABLE I
EVALUATION OF DTMAC ACCELERATION FOR TWO EEMBC BENCHMARKS

	Autcor		FFT	
	Execution time (μs)	Energy (nJ)	Execution time (μs)	Energy (nJ)
GPP w/ MULT	3.51	22.89	127.93	835.11
FlexCore w/ MULT	2.32	14.60	84.41	530.96
FlexCore w/ DTMAC	0.80	5.84	70.20	510.35

Since the compiler can exploit the parallelism inherent in the exposed datapath, the two FlexCore datapaths show significant execution time improvements. The DTMAC accelerator further improves the performance by offering efficient parallel execution of the MAC operation. Since the DTMAC hardware overhead is limited, the power overhead is low and the shorter execution time can be turned into a substantial energy reduction.

The selection of the two EEMBC benchmarks was made in order to show the range of performance improvements that are possible. The impact of using DTMAC instead of MULT is more significant in Autcor than in FFT, because Autcor has many parallel 16-bit multiply-accumulate operations, while FFT contains quite few 16-bit multiply-accumulate operations in sequence.

B. Error Detection Using Cyclic Redundancy Checking

Cyclic redundancy checks (CRCs) are routinely done to ensure that data have not been compromised. Different applications have different CRC needs and this is reflected in the key polynomials being used, for example, CRC5 ($p(x) = x^5 + x^2 + 1$) is used in USB interfaces, CRC8 ($p(x) = x^8 + x^2 + x + 1$) is used in ATM protocols, and CRC16 ($p(x) = x^{16} + x^{12} + x^5 + 1$) is used in XMODEM and X25 protocols. Running CRC on a general-purpose datapath rather than in a dedicated circuit is ineffective in terms of execution time and, thus, in energy. Since a dedicated CRC circuit for one standard is very small and entails insignificant overheads, we can save energy by integrating a CRC accelerator in an exposed datapath.

The challenge of introducing a CRC accelerator is to find a circuit that can handle different key polynomials in an efficient manner. Fig. 12 shows a lightweight accelerator that was built to handle not only CRC5, CRC8 and CRC16 (see above), but also CRC32 which is used in the IEEE 802.3 standard of local area networks [18].

The evaluation, which was done using the PowerStone CRC16 benchmark [19] on the FlexCore datapath, showed that the addition of the CRC accelerator in Fig. 12 improved the datapath performance and energy efficiency by more than seven times as compared to the baseline datapath (Fig. 3). The overheads of integrating this multi-mode CRC accelerator were limited; the area grew by 7%, while the timing was unaffected.

C. Error Correction Using Viterbi Decoding

Forward-error correction (FEC) is computationally challenging and thus FEC accelerators are vital [20]. Standalone accelerator circuits tend to be large, since the memory requirement increases exponentially with the error correction capability. An exposed datapath offers an interesting compromise solution using a mix of hardware acceleration—in a lightweight, memory-limited accelerator—and software execution—in the general-purpose portion of the datapath. To address acceleration of more complex algorithms, we recently described a Viterbi accelerator [21]. To complement the view of the previous two accelerator descriptions, we leave out all implementation details and instead focus on datapath integration and overall application execution.

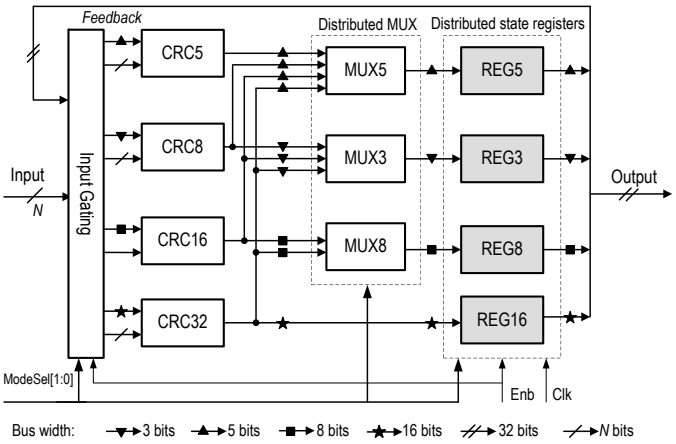

Fig. 12. Block schematic of a CRC accelerator that can support four different CRC standards. The accelerator mode control signals (ModeSel) are exposed to the compiler, which can select which CRC standard to use. The switching power of unused CRC circuits is negligible, since the inputs are not propagated unless needed. As in the case of the DTMAC unit, the local data feedback reduces power dissipation.

The Viterbi accelerator in Fig. 13 has its output registers embedded. The input data arrive only from the register file (RF) and the load/store (LS) units, which simplifies the input switchbox; an example on how the FlexCore interconnect can be configured based on application understanding. One 32-bit word goes directly from the RF unit to the Viterbi unit, while a 2-way multiplexer selects either one 32-bit word from the RF unit (In A) or the 32-bit word from the LS unit (In B). The output of the accelerator only needs to be routed to the RF and the LS units. Thus, the input switchboxes for the RF and the LS units need to accept one more data input.

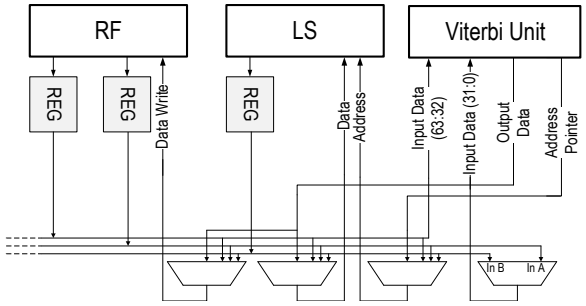

Fig. 13. Integration of the Viterbi accelerator into the FlexCore datapath shown in Fig. 3.

For evaluation we used the EEMBC Viterbi benchmark, which employs a so-called soft-decision algorithm. Fig. 14 illustrates the sequence of execution of this benchmark, with emphasis on the most computation-intensive parts of the benchmark. The evaluation showed that by integrating the Viterbi accelerator, the datapath becomes 20% larger, which directly impacts power dissipation (a 38% increase). However, thanks to the improved performance, the reduced execution time (90% for the total benchmark or 99% for the benchmark kernel) yields an 87% energy reduction for the whole benchmark [21].

The overall design tradeoff is to balance the datapath resource so that it matches the application domain. Due to its complexity, the introduced Viterbi accelerator increases the datapath area significantly, but this is an acceptable overhead in the event the processor will run FEC applications frequently. The next subsection will show that, unlike area overheads, power dissipation overheads—both active and static—can be reduced by using the appropriate circuits.

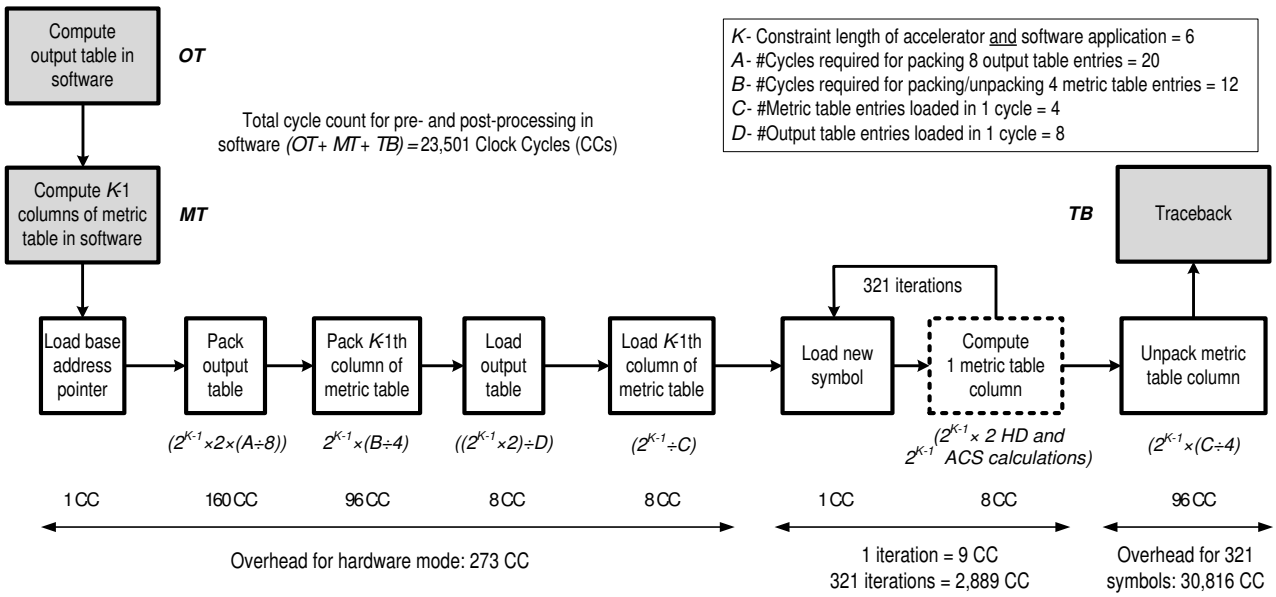

Fig. 14. Example sequence of steps while using a Viterbi accelerator. The gray boxes represent portions of the code that are performed in software, in the general-purpose datapath portion, while transparent boxes represent portions that are accelerated. The execution time for each application phase is given in clock cycles (CC). The constraint length parameter K represents how well the code can correct errors (the higher, the better). K is equal to 6 for the EEMBC benchmark used. HD stands for Hamming distance calculation, whereas ACS means add-compare-select.

D. Power Gating of Idle Circuits

Operand isolation is used for the CRC unit in Fig. 12 to reduce the switching power of unused circuit portions. Although switching remains the dominating mechanism for power dissipation, the leakage power component is significant in advanced CMOS processes, especially in high-performance process technologies (low threshold voltages) at elevated temperature. Unless the power supply is shut off, an accelerator that is not used will still dissipate precious static power. If the complex accelerator is idle for a longer time, the accumulated static energy that is dissipated can become a major problem.

In contrast to operand isolation, introducing power-gating circuits on an accelerator is a very invasive design step. For example, the area increases significantly due to the special circuits and extra power supply wiring that need to be added to the physical implementation. This ultimately increases the switching power and makes it important to also consider the transitions between idle and active modes when minimizing the total processor energy. A scheme based on an exposed datapath offers some advantages in terms of mitigating the static power overhead. Since the compiler is directly controlling the datapath, it is possible to statically identify if an accelerator is idle for such a long time that it is worthwhile to shut off power.

It is not controversial to claim that it is effective to use power gating for large accelerator blocks that are unused during distinct execution phases when the current applications have no need for acceleration. The question we have pursued rather is; can it be effective to implement power gating on a more fine-grain level, shutting down datapath execution units.

We demonstrated that introducing power gating on the 32-bit multiplier can save total energy even when the multiplier is not idle for the entire duration of the application [22]. The power gating circuit infrastructure is shown in Fig. 15 and it is clear that power gating at execution-unit level has a big impact on the physical implementation. Still, for the two EEMBC benchmarks analyzed, shutting down the multiplier when it is idle can yield an overall energy saving. The actual saving depends on how common the multiplication operation

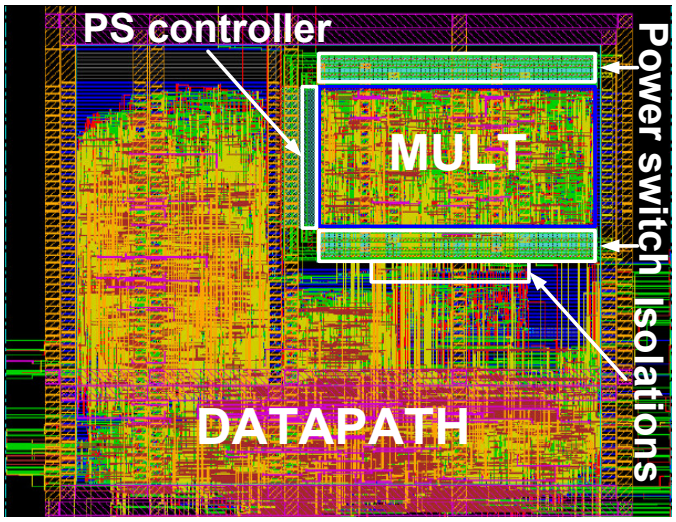

Fig. 15. Physical view after place and route of a 65-nm FlexCore datapath with a power-gated multiplier.

is. For the Autcor benchmark, the multiplier is only used initially as shown in the execution profile in Fig. 16a. Since the multiplier can be shut down for a relatively long time, the total datapath energy saving can be as high as 14.2%, assuming a high-performance process technology. In contrast, in the FFT benchmark the multiplier is used more frequently (see Fig. 16b) and thus the saving is limited to 8%.

Based on earlier studies [23], [24] where the foundation of the techniques was laid, the use of power gating to reduce leakage power for narrow-width operands in integer arithmetic was comprehensively evaluated using an automated design flow for power gating [25]. Among the results from experiments in a 45-nm process technology, power gating of a 32-bit multiplier is demonstrated to yield an 11.6x static leakage energy reduction per 8x8-bit operation, at a performance penalty of 6.7%. Note that these numbers pertain to the arithmetic unit in isolation, not to the processor.

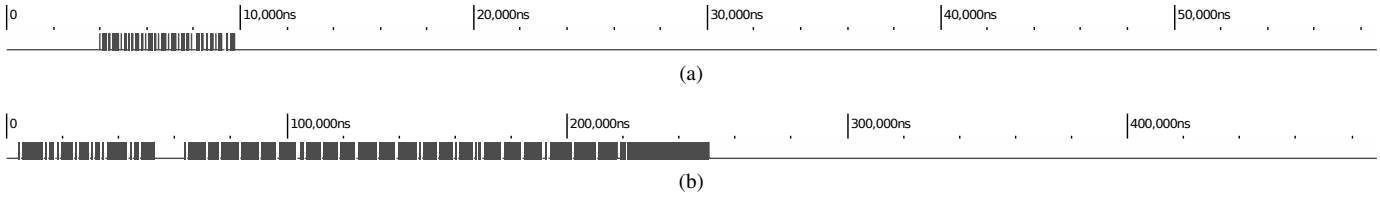

Fig. 16. Profiles of multiplier use for (a) Autcor and (b) FFT benchmark.

VII. Conclusion

The presented FlexCore processor implementations have demonstrated our approach to using exposed datapaths for energy-efficient computing. Our processor development environment enables co-implementation and co-verification of software and hardware to realize and evaluate novel combinations of concepts such as exposed datapath control, rich interconnect, on-the-fly instruction decompression, and datapath extensions. The evaluations we provide show that the FlexCore approach intrinsically offers interesting possibilities. While the instruction decompression incurs an area overhead, it allows for a static schedule that can make efficient use of all hardware resources and it allows for domain-specific datapath extensions that can accelerate applications substantially. Although the FlexCore implementation work has come to explore many areas, there is still scope for further performance and energy improvements. As far as future work, many improvements rely on compiler modifications: For example, making the compiler support AS-ISA reload instructions (Sec. II-B) could reduce the size of the decompressor bit-pattern by at least three times [8].

VIII. Acknowledgement

We want to acknowledge the work of many individuals that have made contributions to the FlexSoC research programme: H. Ali, S. Alipour, M. W. Azhar, K. K. Ansari, A. Bardizbanyan, M. Björk, M. Brinck, A. R. Buzdar, T. Carlqvist, D. Eckerbert, K. Eklund, J. Elahi, J. Ferry, N. Frolov, P. Gammie, P. Gavin, B. Goel, E. der Hagopian, S. M. Hassan, B. Hidaji, T. T. Hoang, J. Hughes, A. Ibrahim, M. Islam, K. Jeppson, U. Jälmbrant, D. Knyaginin, J. Lidman, S. A. McKee, J. Mårts, B. Nasri, M. Pellauer, C. Prabhushankar, E. Ryman, V. Saljooghi, V. Saseendran, T. Schilling, M. Sheeran, D. Siaudinis, P. Stenström, K. P. Subramaniyan, L. Svensson, M. Thuresson, A. Vijayshekar, D. Whalley and T. Yang.

References

[1] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz, "Understanding sources of inefficiency in general-purpose chips," in *27th Annual Int. Symp. on Computer Architecture*, 2010, pp. 37–47.

[2] M. Thuresson, M. Själander, M. Björk, L. Svensson, P. Larsson-Edefors, and P. Stenstrom, "FlexCore: Utilizing exposed datapath control for efficient computing," *J. of Signal Processing Systems*, vol. 57, no. 1, pp. 5–19, 2009.

[3] T. T. Hoang, U. Jälmbrant, E. der Hagopian, K. P. Subramaniyan, M. Själander, and P. Larsson-Edefors, "Design space exploration for an embedded processor with flexible datapath interconnect," in *Proc. IEEE Int. Conf. on Application-specific Systems Architectures and Processors*, Jul. 2010, pp. 55–62.

[4] H. Corporaal and M. Arnold, "Using transport triggered architectures for embedded processor design," *Integrated Computer-Aided Engineering*, vol. 5, no. 1, pp. 19–38, 1998.

[5] B. Gorjiara and D. Gajski, "Automatic architecture refinement techniques for customizing processing elements," in *45th ACM/IEEE Design Automation Conf.*, Jun. 2008, pp. 379–384.

[6] I. Finlayson, G. Uh, D. Whalley, and G. Tyson, "An overview of static pipelining," *IEEE Computer Architecture Letters*, vol. 11, no. 1, pp. 17–20, Jun. 2012.

[7] D. A. Patterson and J. L. Hennessy, *Computer Organization & Design, The Hardware/Software Interface*, 2nd ed. Morgan Kaufman Publishers Inc., 1998.

[8] M. Thuresson, M. Själander, and P. Stenstrom, "A flexible code compression scheme using partitioned look-up tables," in *Proc. 4th Int. Conf. on High Performance Embedded Architectures and Compilers*, 2009, pp. 95–109.

[9] Embedded Microprocessor Benchmark Consortium. [Online]. Available: http://www.eembc.org

[10] The FlexSoC Project. [Online]. Available: http://www.flexsoc.org

[11] V. Saljooghi, A. Bardizbanyan, M. Själander, and P. Larsson-Edefors, "Configurable RTL model for level-1 caches," in *Proc. IEEE NORCHIP Conf.*, Nov. 2012.

[12] M. Själander, P. Larsson-Edefors, and M. Björk, "A flexible datapath interconnect for embedded applications," in *Proc. IEEE Computer Society Annual Symp. on VLSI*, May 2007, pp. 15–20.

[13] B. Hidaji, S. Alipour, K. P. Subramaniyan, and P. Larsson-Edefors, "Application-specific energy optimization of general-purpose datapath interconnect," in *Proc. IEEE Computer Society Annual Symp. on VLSI*, Jul. 2011, pp. 301–306.

[14] A. Bardizbanyan, M. Själander, and P. Larsson-Edefors, "Reconfigurable instruction decoding for a wide-control-word processor," in *Proc. IEEE Int. Symp. on Parallel and Distributed Processing*, May 2011, pp. 322–325.

[15] ARM Limited, *AMBA Design Kit Technical Reference Manual, revision r3p0*.

[16] Aeroflex Gaisler, *GRLIB IP Core User's Manual, Version 1.1.0 B4104*. [Online]. Available: http://gaisler.com/products/grlib/grip.pdf

[17] T. T. Hoang, M. Själander, and P. Larsson-Edefors, "Double throughput multiply-accumulate unit for FlexCore processor enhancements," in *Proc. IEEE Int. Symp. on Parallel and Distributed Processing*, May 2009.

[18] M. W. Azhar, T. T. Hoang, and P. Larsson-Edefors, "Cyclic redundancy checking (CRC) accelerator for the FlexCore processor," in *Proc. Euromicro Conf. on Digital System Design: Architectures, Methods and Tools*, Sep. 2010, pp. 675–680.

[19] J. Scott, L. H. Lee, A. Chin, J. Arends, and W. Moyer, "Designing the low-power M•CORE™ architecture," in *IEEE Power Driven Microarchitecture Workshop*, Jun. 1998, pp. 145–150.

[20] M. A. Anders, S. K. Mathew, S. K. Hsu, R. K. Krishnamurthy, and S. Borkar, "A 1.9 Gb/s 358 mW 16-256 state reconfigurable Viterbi accelerator in 90 nm CMOS," *IEEE J. Solid-State Circuits*, vol. 43, no. 1, pp. 214–222, Jan. 2008.

[21] M. W. Azhar, M. Själander, H. Ali, A. Vijayashekar, T. T. Hoang, K. K. Ansari, and P. Larsson-Edefors, "Viterbi accelerator for embedded processor datapaths," in *Proc. IEEE Int. Conf. on Application-specific Systems Architectures and Processors*, Jul. 2012, pp. 133–140.

[22] T. T. Hoang, V. Saseendran, D. Siaudinis, and P. Larsson-Edefors, "Power gating multiplier of embedded processor datapath," in *Proc. Conf. on Ph.D. Research in Microelectronics and Electronics*, Jul. 2011, pp. 41–44.

[23] M. Själander and P. Larsson-Edefors, "Multiplication acceleration through twin precision," *IEEE Trans. on Very Large Scale Integrated Systems*, vol. 17, pp. 1233–1246, Sep. 2009.

[24] M. Själander, M. Draždžiulis, P. Larsson-Edefors, and H. Eriksson, "A low-leakage twin-precision multiplier using reconfigurable power gating," in *Proc. IEEE Int. Symp. on Circuits and Systems*, May 2005, pp. 1654–1657.

[25] T. T. Hoang and P. Larsson-Edefors, "Data-width-driven power gating of integer arithmetic circuits," in *Proc. IEEE Computer Society Annual Symp. on VLSI*, Aug. 2012, pp. 237–242.