A Coarse-Grained Array Accelerator for Software-Defined Radio Baseband Processing

A shrinking energy budget for mobile devices and increasingly complex communication standards make architecture development for software-defined radio very challenging. Coarse-grained array accelerators are strong candidates for achieving both high performance and low power. The C-programmable hybrid CGA-SIMD accelerator presented here targets emerging broadband cellular and wireless LAN standards, achieving up to 100-Mbps throughput with an average power consumption of 220 mW.

Bruno Bougard
IMEC

Bjorn De Sutter
Ghent University

Diederik Verkest
Liesbet Van der Perre
Rudy Lauwereins
IMEC

Wireless technology is considered a key enabler of future consumer products and services. To cover the extensive range of applications, future handheld devices must support a wide variety of wireless communication standards concurrently. The growing number of air interfaces makes traditional implementations based on the integration of multiple specific radios and baseband ICs cost-ineffective. By contrast, software-defined radios (SDRs) achieve flexibility and cost-efficiency by deploying baseband processing on programmable or reconfigurable processors.[1] Researchers in academia and industry have already proposed several SDR platforms, of which most support current wireless standards such as W-CDMA (UMTS)—Wideband Code Division Multiple Access (Universal Mobile Telecommunications System)—IEEE 802.11 b/g, and IEEE 802.16.[1–5]

However, a major challenge remains in implementing emerging multicarrier and multi-antenna standards while maintaining cost-effectiveness: Compared to the current wireless standards, standards such as IEEE 802.11 n and LTE (Long Term Evolution) represent a tenfold increase in complexity and in required throughput. Technology scaling will no longer suffice to sustain the complexity increase. Instead, we must revise architectures to achieve the required performance at energy budgets that are acceptable for handheld integration (about 300 mW). This revision must take into account the key characteristics of wireless baseband processing: Most of the computation time is spent in inner loops (also known as kernels) that feature high data-level parallelism (DLP) and high instruction-level parallelism (ILP), corresponding to simple control flow.

This article presents the design, implementation, and performance evaluation of a C-programmable hybrid coarse-grained array, single-instruction, multiple-data (CGA-

SIMD) SDR accelerator. This accelerator exploits the high ILP available in SDR kernels, combined with simple and effective DLP support. Its programming flow is fully integrated with that of the main CPU. A unified compiler[6] maps the sequential, nonkernel ANSI C code onto the main CPU, while mapping the loops from that same ANSI C code onto the accelerator. The result is almost as energy efficient as wide SIMD (vector) architectures, without those architectures' limited flexibility and programming burden.

Existing SDR architectures

Some existing SDR platforms rely heavily on wide SIMD to exploit DLP in kernels with limited instruction fetch overhead. Examples are NXP's EVP16 processor[2] and the SODA platform from the University of Michigan, Ann Arbor.[3] A SODA processor further reduces power consumption by eliminating all hardware support for variable vector widths. Instead, the programmer must know all vector widths and write specific assembly code for them manually. A major disadvantage of these architectures is the lack of compiler support, which stems from the fact that separate wide SIMD data paths are very complex targets to program. Furthermore, how well future standards will map onto these wide SIMD architectures with limited flexibility remains an open question.

Wide VLIW architectures offer more flexibility because each (parallel) data operation is programmed separately. The VLIW-like architectures Sandblaster[1] and HiveFlex[7] come with full compiler support. Their flexibility potentially also makes them better targets for future standards. However, these architectures inherently consume more energy than wide-SIMD architectures. HiveFlex tries to limit the overhead by combining a narrow nonloop VLIW mode with a wide loop VLIW mode. Still, the loop mode does not exploit kernels' dataflow-like character very well—unlike wide SIMD data paths. It is therefore unclear to what levels of performance the HiveFlex or Sandbridge can scale, and what energy efficiency they can obtain at higher clock speeds.

Coarse-grained reconfigurable array (CGA) architectures exploit the dataflow dominance.[5,8,9] Although this class of processors offers more parallel resources, these processors are typically only programmable at the assembly level. Furthermore, because the CGA accelerator is usually loosely coupled to the main CPU (as is, for example, MorphoSys[9]), its programming is complicated by the need for an explicit data-passing interface at design time. Also, this interface can involve a significant data-passing overhead at runtime.

Our accelerator, an instance of the Adres (architecture for dynamically reconfigurable embedded systems) architecture template,[10] aims to combine the advantages of all the aforementioned approaches while trading off their drawbacks. We achieve high performance using a CGA-like ILP architecture that is coupled very tightly to a main CPU. Leveraging the DRESC (dynamically reconfigurable embedded system compiler) ANSI C compiler framework,[6] we keep the ease of programming fundamentally at the same level as that of the Sandblaster framework. Meanwhile, the limited but effective SIMD support helps maintain the energy-efficiency in the same range as the EVP and SODA implementations. Furthermore, we are confident that our architecture's flexibility scales well to future standards such as LTE, thus delaying the need for complicated many-core programming.

Accelerator architecture

Figure 1 depicts the top-level architecture of the proposed accelerator. It consists of 16 interconnected 64-bit functional units (FUs) connected to many small, distributed register files, and to a larger register that the accelerator shares with a closely coupled main CPU.

Operation mode

The (single physical) shared register file consists of 64 64-bit registers that can pass data from the main CPU, which executes nonaccelerated code, to the CGA and back. Because the main CPU and the accelerator also share a level-1 scratchpad memory and its interface, invoking the accelerator does not require high data-passing overhead: The data is already in place in either the shared

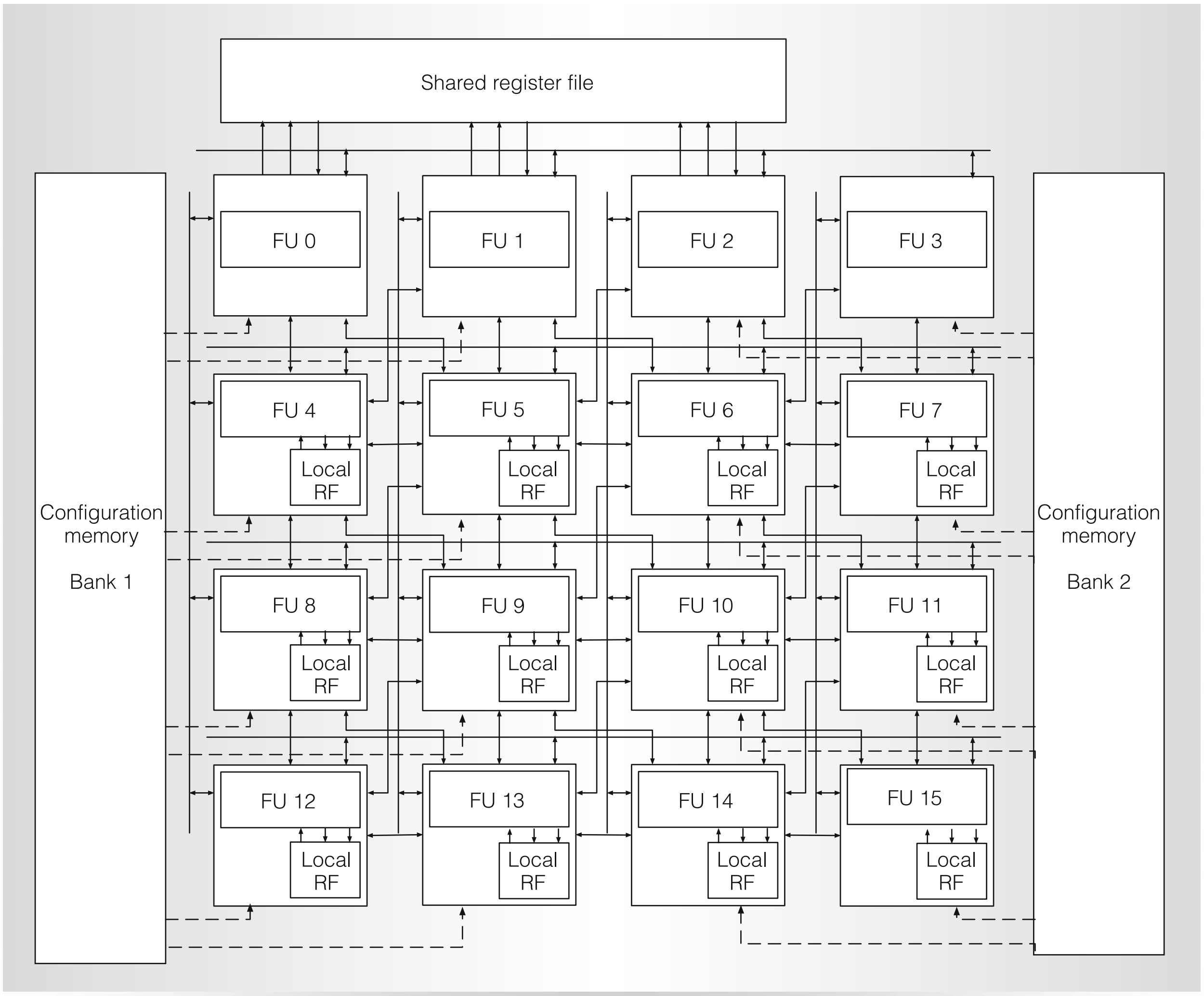

Figure 1. Accelerator architecture: 16 functional units (FUs) are connected to one shared and 12 local register files (RFs) through a heterogeneous, dynamically reconfigurable interconnect, for which the configurations are fetched from the configuration memory banks, together with the code to be executed.

memory or in the shared register file. On our SDR platform, the entire switch requires only two clock cycles. Moreover, because the accelerator executes as a real coprocessor—that is, not overlapping in time with the main VLIW CPU—the programming model is the simple sequential-code model. Our ANSI C compiler fully automatically generates statically scheduled code for the main CPU and for the accelerator, and inserts the necessary code for invoking the accelerator.

Figure 2a outlines a sequential C code fragment containing one application kernel in the form of a *for* loop. The compiler partitions the code into five parts. *Preloop code* and *postloop code* are compiled into binary code for the main CPU. This code will be stored in the main CPU's instruction memory, and will be fetched through an instruction cache as shown in Figure 2b. To invoke the CGA, the compiler also inserts loop invocation code between the pre- and postloop code. Furthermore, the compiler generates software-pipelined code[11] for the loop by partitioning its loop body into pipeline stages—in this case, three stages that take two each to execute. All three stages are allocated in the accelerator's configuration memory, as Figure 2b shows.

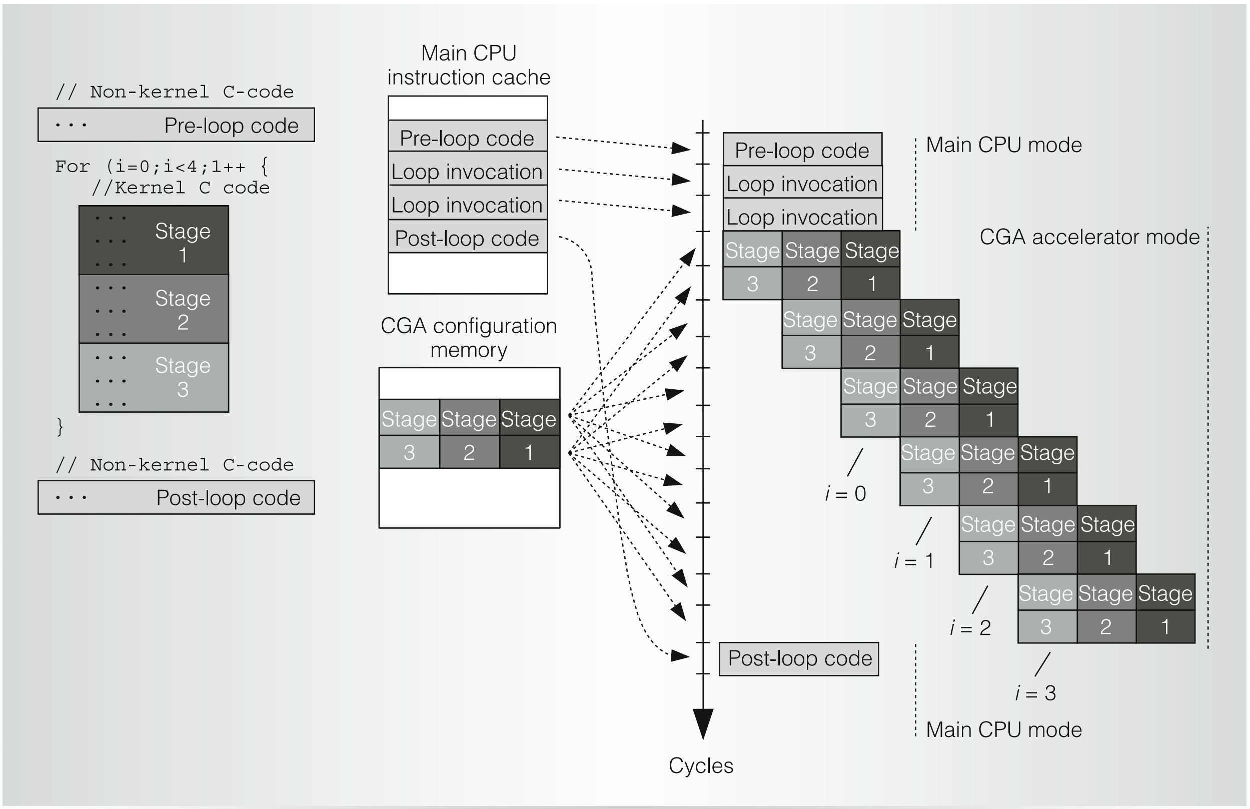

Figure 2. Accelerator operating mode: source code, code layout in memories, and execution trace.

Figure 2c illustrates the execution of the example code fragment. First, the main CPU executes the preloop code, after which it invokes the accelerator, which then takes over control.

Using the configuration memory addresses it got from the main CPU through the invocation code, the accelerator iteratively fetches configurations—that is, statically scheduled code—from the CGA configuration memory and executes them on the array. The dotted arrows in Figure 2 depict the fetching of these configurations. When the loop exit condition is triggered, the accelerator returns control to the main CPU.

As the trace in Figure 2 makes clear, four full iterations of the original loop have been executed (for $i = 0, 1, 2,$ and 3). The original loop schedule, which required 3×2 cycles for the three stages of each loop iteration, would have required 24 cycles ($3 \times 2 \times 4$) to execute the whole loop. By contrast, the CGA code starts a new iteration of the software-pipelined loop after every two cycles. As a result, the full execution of the loop takes only 12 cycles on the accelerator. Moreover, this acceleration is achieved with a very simple configuration code-fetching mechanism, which is similar to L0 loop buffering.[12] This mechanism ensures that the configuration fetching consumes relatively little energy, even with configuration words several hundred bits wide. This is possible because the loop bodies compiled for the accelerator feature no control flow. For loops that contain a control flow, such as conditional statements, the compiler inserts predication to convert the control flow into dataflow.[13]

Core

The 16 64-bit core FUs perform the loop body computations. These predicated FUs can perform all regular arithmetic and logic operations, as well as instructions to generate the predicates. They can also perform several special instructions that implement four-way 16-bit SIMD operations, such as parallel shifting and parallel subtraction and addition. All of these basic operations have a latency of one cycle. In addition, all FUs can also execute 16-bit integer signed and unsigned multiplications, as well as two-way SIMD multiplications of 2×16-bit complex fixed-point numbers. All multiplications have three-cycle latency. Additionally, one FU can execute a 24-bit division with a latency of eight cycles. All multicycle operations are fully pipelined.

Three of the 16 FUs are connected to the shared register file through two read ports and one write port each. These ports are shared with the issue slots of the main CPU. The other FUs each have a local 64-bit register file that features two read ports, one write port, and four rotating registers to support software pipelining. In the shared register file, only the top 32 of the 64 registers are rotating registers. Because of their smaller size and reduced number of ports, the local registers are far less power-hungry than the shared register file.

The register files and FUs in the CGA are connected via a dense 64-bit-wide interconnect network, as Figure 1 shows. Basically, the FUs are connected to their local register files and to neighboring FUs. This lets data flow through the accelerator efficiently. A multiplexer sits in front of each source operand port of each FU and in front of each read port of a register file that has more than one incoming connection. Each multiplexer is programmed explicitly every cycle by selection bits that are fetched from configuration memories. Besides the multiplexer selection bits, the configurations also include opcodes to be executed on the FUs, and addresses to be set at the register file ports.

The compiler generates all configurations.[6] This compiler relies on *intrinsics* (function calls to built-in functions that encapsulate complex instructions) in the ANSI C code to program the SIMD operations. Apart from this way of being programmed at the source-code level, the SIMD operations execute just like any other operation, on exactly the same data path. This lets them be scheduled as, and in between, the other regular operations, greatly facilitating the generation of mixed regular code and SIMD code. On processors that have a separate SIMD data path—which has the advantage of allowing a wider SIMD datapath—code generation is typically much more difficult, and hence often not as automated as in our flow.

Memory interface

To access the streaming data handled by the accelerated kernels, the accelerator core shares a level-1 scratchpad memory with the main CPU. This shared memory must offer sufficient bandwidth to feed the computations being performed on the 16 FUs. In addition, memory power consumption must be minimized. In our implementation, we meet the throughput requirement by equipping four of the 16 FUs with load- and store-capable units. These four load-store units connect to four single-ported, 32-bit, interleaved memory banks through a crossbar. We chose four units and four banks because the required peak memory bandwidth for many important kernels is between 3 and 4 accesses per cycle (sustained throughout the steady-state phase of software-pipelined loops), and because making the number of memory banks a power of two facilitates the interleaving.

Obviously, having multiple single-ported banks can result in bank conflicts if different load-store units want to access the same bank at the same time. On architectures with blocking loads, such as our accelerator, the simplest method to resolve such conflicts consists of stalling the processor until all requests to a bank have been handled. However, with four concurrent load-store units, the chance of having conflicting accesses is high, so this approach could result in serious performance degradation. For important kernels such as fast Fourier transforms (FFTs), we measured slowdowns of up to 50 percent. To avoid such a slowdown, we have designed an alternative conflict resolution scheme that relies on longer load instruction latencies.

Longer load instruction latencies are most often not detrimental to performance in static software-pipelined schedules. Consider the trace in Figure 2 again. Even if the schedule length of one iteration in the original loop would become 33 percent longer because load operations are given a

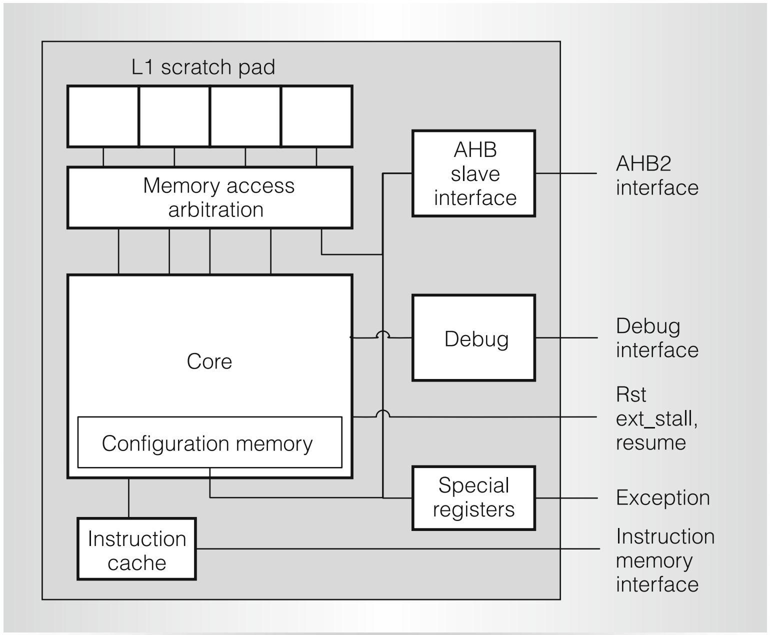

Figure 3. Processor top-level architecture.

higher latency—meaning we would need four pipeline stages instead of three—the length of the software-pipelined loop trace of 12 cycles would increase to only 14 cycles (for the additional stage of the last iteration), an increase of only 17 percent.

We have designed a memory interface in which load operations have a latency of seven cycles instead of the lowest obtainable latency of four cycles in our CGA design. The three additional cycles serve to buffer issued memory accesses until they can actually be performed—that is, when their respective banks become available—and to store already-loaded values before they are fed back to the CGA load-store units. As such, up to three accesses to the same bank can be issued in the same cycle (or be outstanding) without requiring the processor to stall. This reduces the bank-conflict stall overhead for the aforementioned FFT benchmarks to below 10 percent, while still enabling the use of single-ported memory banks to decrease the power consumption.

In the main CPU mode, when the code is not software pipelined and the chance of access conflict is lower (only three load-store units are present on the main CPU), the buffers are disabled. This enables the main CPU to execute memory accesses with the shorter latency of five cycles.

Implementation

To prove the viability of our tightly coupled SDR accelerator concept, we have developed a processor prototype consisting of a simple VLIW CPU coupled with the proposed accelerator. We designed this processor to serve as a slave in multicore SDR platforms.[14]

Accelerator integration

Figure 3 depicts the top-level block diagram of the prototype processor. The main CPU in this processor is a three-issue VLIW. The processor has an asynchronous reset, a single external system clock, and a half-speed (AMBA) bus clock. Instruction flow and dataflow are separated (Harvard architecture). A direct-mapped instruction cache is implemented for the VLIW CPU with a dedicated 128-bit-wide instruction memory interface. The level-1 data memory is as we described earlier, with four 32-bit-wide banks that can each store 16 Kbytes of data. These banks can also be accessed externally through an AMBA2-compatible slave bus interface (which connects to the memory interface just like an additional load-store unit in the processor). The CGA configuration memories (128 configurations) and special registers are also mapped to the AMBA bus interface via a 32-bit internal bus. This way, the CGA configuration memories and the level-1 scratchpad data memories can be accessed via direct-memory-access (DMA) transfers.

In addition, the processor has a level-sensitive control interface with configurable external endianness and AMBA high-performance bus (AHB) priority settings (which means there is configurable priority between core and bus interface to access the memory), exception signaling, and external *stall* and *resume* input signals. Because of the large state, the accelerator is noninterruptible. However, the external *stall* and *resume* signals provide an interface to work as a slave in a multiprocessor platform. The *stall* signal stops the processor while maintaining its state (for example, to implement flow control at the SoC level). Internally, a special *stop* instruction can be issued to set the processor in an internal sleep state, from which it can recover by asserting the *resume* signal. The data scratchpad and special register bank remain accessible through the AHB interface in sleep mode.

Process and library selection

The architecture we have described is implemented to reach a 400-MHz clock rate in worst-case conditions when implemented in 90-nm technology. Hence, with 16 units × four-way SIMD × 400 MHz, it delivers up to 25.6 giga-ops (16-bit ops), sufficient to implement 2 × 2 20-MHz multiple-input multiple-output, orthogonal-frequency division multiplex (MIMO-OFDM) at 100 Mbps or more.[15] To achieve such a clock frequency at maximum energy efficiency, we have selected a general-purpose process. Although it is leakier, the general-purpose process has a better power-delay product than the low-power process usually considered for embedded applications. Our implementation tackles leakage in operation mode with different threshold voltages for critical and noncritical paths (using what is known as a multi-VT design) and, in standby, with third-party, substrate-biased standard-cell library and memory macros.

Power-aware design

We wrote the register-transfer-level (RTL) descriptions of the FUs and multiported register files to enable automated fine-grained clock gating during synthesis. It turned out that 95 percent of the flip-flops were clock-gated with the appropriate activation signal. Furthermore, we manually implemented operand isolation in the FUs to avoid bit toggling in unused operators. Finally, we inserted scan test support and memory BIST logic.

We used the resulting netlist as input for physical design with Cadence SOC Encounter. Macros are placed at the periphery, as Figure 4 shows. Standard-cell placement was then optimized, followed by clock tree synthesis and final place and route. After parasitic extraction from the resulting layout, we checked timing with Synopsys PrimeTime and estimated power with Synopsys PrimePower.

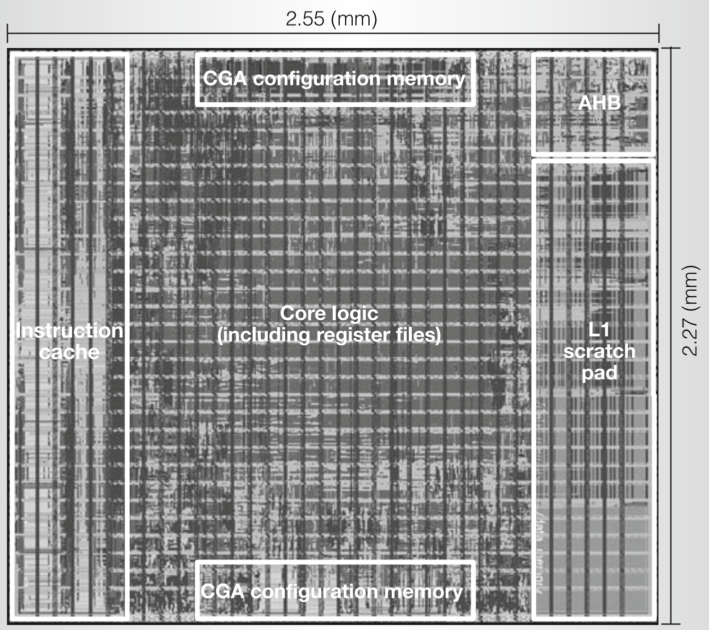

Figure 4. Processor layout and main building blocks.

Design results

The final layout achieves a timing of less than 2.5 ns in worst-case conditions, which enables the operation of the processor at 400 MHz. The critical path is located in the execution stage of the FUs implementing

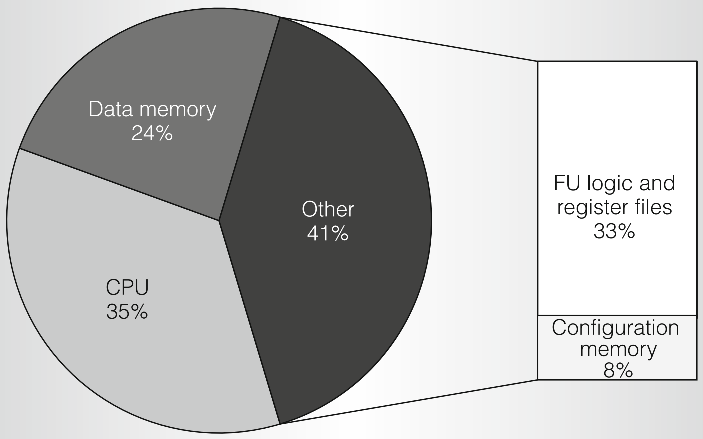

Figure 5. Prototype processor area breakdown.

Table 1. Benchmark descriptions.

Shorthand	Description	Throughput
11 n 64QAM Tx	IEEE 802.11 n transmitter with two-antenna space-division multiplexing, 64-QAM-OFDM in 20 MHz	108 Mbps
11 n 64QAM Rx	IEEE 802.11 n receiver with two-antenna space-division multiplexing, 64-QAM-OFDM in 20 MHz	108 Mbps
11 g 64QAM Tx	IEEE 802.11 g single-antenna transmitter, 64-QAM-OFDM in 20 MHz	54 Mbps
11 g 64QAM Rx	IEEE 802.11 g single-antenna receiver, 64-QAM-OFDM in 20 MHz	54 Mbps
LTE Tx	3GPP-LTE single-antenna transmitter, 16-QAM in 5 MHz	Up to 18 Mbps

QAM: quadrature amplitude modulation

the pipelined multiplier. The prototype die area is 5.79 mm^2, including level-1 data, instruction, and configuration memories. Figure 5 gives a more detailed area breakdown. The accelerator occupies 41 percent of the area (2.37 mm^2); FU logic and register files take 33 percent.

Performance evaluation

We dimensioned the processor to enable the execution of next-generation broadband cellular communication and wireless LAN standards. To evaluate its performance and power consumption, we selected a set of benchmarks corresponding to transmitter and receiver baseband processing in the IEEE 802.11 n and 3GPP-LTE standards. Table 1 presents some details of these benchmarks, which we implemented to meet the real-time latency constraints of the respective protocols.

For each benchmark, Table 2 presents the portion of the execution time spent in accelerated mode as well as the instructions per cycle (IPC) ratio obtained in that mode. The average IPC over the different benchmarks is 10.18. This number includes regular and SIMD operations, each counted as one operation. On average, about 45 percent of the operations are SIMD operations. The resulting utilization of the accelerator's FUs is 10.18/16 = 63.65 percent. This figure is particularly high considering that it is obtained with compiled ANSI C code.

Furthermore, the prototype power consumption was estimated on the basis of a gate-level simulation. The netlist resulting from the physical design, back-annotated with capacitance and parasitics information extracted from the final layout, was simulated with Mentor Graphics ModelSim. From such simulation, we generated accurate gate-level activity profiles. We used Synopsys PrimePower to evaluate the CPU and accelerator power consumption based on the activity profile and the layout information. Table 3 lists the results. Peak CPU and accelerator powers are for the typical design corner (V = 1 V, nominal process, T = 25 °C). Leakage is extrapolated to typical leakage corner (V = 1 V, nominal process, T = 65 °C). In each case, we added the data memory hierarchy power consumption. Table 4 presents the average estimated power consumption for executing the different benchmarks in real time.

Table 2. Accelerator performance.

Benchmark	Time in accelerator mode (%)	IPC
11 n 64QAM Tx	64	10.75
11 n 64QAM Rx	56	9.99
11 g 64QAM Tx	53	11.05
11 g 64QAM Rx	56	10.37
LTE Tx	99	8.76

Table 3. Processor power consumption.

Component	Active power, typical (mW)	Leakage power, typical (mW)	Leakage power, T = 65 °C (mW)
CPU + data memory	75	12.5	25
Accelerator + data memory	310	12.5	25

Figure 6 further breaks down the accelerator active power. A major fraction, 38 percent, goes to the interconnect subsystem, which includes buffers, multiplexers, and pipeline registers between the FUs. The FUs themselves, the configuration memories (CMEMs), and the data memories (DMEMs) consume 29 percent, 14 percent, and 11 percent. The shared and distributed register files consume 6 and 2 percent.

Early experiments on prototype SoC samples taped out in March 2008 confirm that, in practice, the estimated frequency and power consumption numbers are reached.

This article presented the design of a hybrid CGA-SIMD accelerator dedicated to software-defined radio baseband processing. The performance and power-efficiency obtained with our prototype—comprising the accelerator coupled to a basic VLIW CPU and a multibanked data memory hierarchy—demonstrates that our compiler-supported CGA-SIMD approach is practically viable for current and future wireless standards. We know of no other architectures that combine our CGA-SIMD's level of performance, power-efficiency, and C programmability. As both the presented design and the supporting compiler are early research prototypes, many enhancements can still be made, both to the hardware and to the software support. A very interesting direction for ongoing and future research is the exploitation of thread-level parallelism (TLP) in CGA-SIMD architectures, in addition to the already available support for ILP and DLP. MICRO

Table 4. Average power consumption per benchmark.

Benchmark	Average power consumption (mW)
11 n 64QAM Tx	250
11 n 64QAM Rx	232
11 g 64QAM Tx	225
11 g 64QAM Rx	232
LTE Tx	333

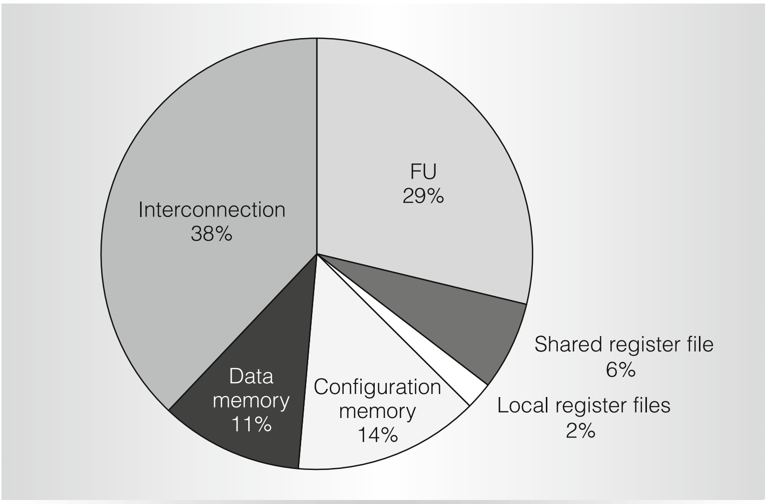

Figure 6. Power consumption breakdown in acceleration mode.

References

1. J. Glossner et al., ''The Sandbridge SB3011 Platform,'' *Eurasip J. Embedded Systems*, vol. 2007, article ID 56467; http://www.hindawi.com/getarticle.aspx?doi=10.1155/2007/56467.
2. K. van Berkel et al., ''Vector Processing as an Enabler for Software-Defined Radio in Handheld Devices,'' *Eurasip J. Applied Signal Processing*, vol. 2005, no. 16, Sept. 2005, pp. 2613-2625.
3. Y. Lin et al., ''SODA: A Low-Power Architecture for Software Radio,'' *Proc. 33rd Ann. Int'l Symp. Computer Architecture*, IEEE CS Press, 2006, pp. 89-101.
4. U. Ramacher, ''Software-Defined Radio Prospects for Multistandard Mobile Phones,'' *Computer*, vol. 40, no. 10, Oct. 2007, pp. 62-69.
5. D. Arditti Ilitzky et al., ''Architecture for the Scalable Communications Core's Network on Chip,'' *IEEE Micro*, vol. 27, no. 5, Sept./Oct. 2007, pp. 62-74.
6. B. Mei et al., ''Exploiting Loop-Level Parallelism on Coarse-Grained Reconfigurable Architectures Using Modulo Scheduling,'' *Proc. IEE Computers and Digital Techniques*, vol. 150, no. 5, 22 Sept. 2003, pp. 255-261.
7. *HiveFlex CSP 2000 Series, Programmable OFDM Communication Signal Processor*, SiliconHive; http://www.siliconhive.com.
8. A. Lodi et al., ''XiSystem: A XiRisc-Based SoC with Reconfigurable IO Module,'' *IEEE J. Solid-State Circuits*, vol. 41, no. 1, Jan. 2006, pp. 85-96.

9. H. Singh et al., ''MorphoSys: An Integrated Reconfigurable System for Data-Parallel and Computation-Intensive Applications,'' *IEEE Trans. Computers*, vol. 49, no. 5, May 2000, pp. 465-481.
10. B. Mei et al., ''Architecture Exploration for a Reconfigurable Architecture Template,'' *IEEE Design & Test*, vol. 22, no. 2, Mar.-Apr. 2005, pp. 90-101.
11. B. Ramakrishna Rau, ''Iterative Modulo Scheduling: An Algorithm for Software Pipelining Loops,'' *Proc. 27th Ann. Int'l Symp. Microarchitecture* (MICRO 94), ACM Press, 1994, pp. 63-74.
12. E. Gibert, J. Sánchez, and A. González, ''Flexible Compiler-Managed L0 Buffers for Clustered VLIW Processors,'' *Proc. 36th Ann. IEEE/ACM Int'l Symp. Microarchitecture*, IEEE CS Press, 2003, pp. 315-325.
13. S.A. Mahlke et al., ''Effective Compiler Support for Predicated Execution Using the Hyperblock,'' *Proc. 25th Ann. Int'l Symp. Microarchitecture* (MICRO 92), IEEE CS Press, 1992, pp. 45-54.
14. L. Van der Perre et al., ''Architectures and Circuits for Software Defined Radios: Scaling and Scalability for Low Cost and Low Energy,'' *Proc. Int'l Solid-State Circuits Conf.* (ISSCC 07), IEEE Press, 2007, pp. 568-589.
15. B. Bougard et al., ''Energy Efficient Software Defined Radio Solutions for MIMO-Based Broadband Communication,'' *Proc. European Signal Processing Conf.*, European Assoc. for Signal Processing, 2007; http://www.eurasip.org/Proceedings/Eusipco/Eusipco2007/Papers/c1l-a03.pdf.

Bruno Bougard is a senior researcher at IMEC, Belgium's Interuniversity Micro-Electronic Centre, which performs research and development, ahead of industrial needs by three to 10 years, in microelectronics, nanotechnology, enabling design methods, and technologies for information and communications technology systems. His research interests focus on energy-efficient circuits and systems for broadband wireless communication, specifically energy-aware software-defined and cognitive radio. He has an MSc in electrical engineering from the Polytechnic University of Mons, Belgium, and a PhD in electrical engineering from Katholieke Universiteit Leuven, Belgium.

Bjorn De Sutter led the architecture and compilation team at IMEC until early 2008, where he completed the work described in this article. He now holds a research position at Ghent University. His compiler research has focused on whole-program optimization, program compaction, binary rewriting, and code generation techniques for reconfigurable architectures. He has an MSc and a PhD in computer science from Ghent University, Belgium.

Diederik Verkest is a member of IMEC's VLSI Design Methodology Group and is currently in charge of IMEC's research on design technology for nomadic embedded systems. He is also a professor at Vrije Universiteit Brussel and at Katholieke Universiteit Leuven. He has an MSc degree and a PhD in applied sciences from Katholieke Universiteit Leuven.

Liesbet Van der Perre is the scientific director of the IMEC Wireless Research Group, comprising teams of researchers in the fields of digital baseband solutions, RF front ends, cross-layer optimization, mixed-signal design technologies, and ultralow-power radios. She has an MSc and a PhD in electrical engineering from Katholieke Universiteit Leuven.

Rudy Lauwereins is vice president of IMEC, where he leads the Nomadic Embedded Systems Division. He is also a professor at Katholieke Universiteit Leuven. His current research interests include many system-level aspects of digital design technology. These range from multicore software mapping, over the exploitation of 3D stacking technology, to system-level solutions to process variability. He has an MEng and a PhD in electrical engineering from Katholieke Universiteit Leuven.

Direct questions and comments about this article to Liesbet Van der Perre, IMEC, Kapeldreef 75, B-3001 Leuven, Belgium; Liesbet.VanderPerre@imec.be.