Hardware Coprocessor Synthesis from an ANSI C Specification

Sumit Ahuja
Virginia Polytechnic and State University

Sandeep K. Shukla
Virginia Polytechnic and State University

Swathi T. Gurumani and Chad Spackman
CebaTech

Editor's note:
This article shows how design space exploration can be realized through high-level synthesis. It presents a case study of a hardware implementation of the Advanced Encryption Standard (AES) Rijndael algorithm. Starting from the algorithmic specification, the authors generate various architectures by using the C2R compiler.

—Philippe Coussy, Université de Bretagne-Sud

Advances in semiconductor technology, still governed by Moore's law, have enhanced our ability to create extremely large, complex microelectronic systems. SoCs let designers harness the increased number of transistors on a single chip and satisfy the ever-increasing demand for more functionalities in electronic systems. A growing trend in SoC design is to have heterogeneous processors on a single chip. In the embedded systems domain, a common performance enhancement strategy is to use dedicated coprocessors, to which the main processor can offload computationally intensive tasks, such as encryption, decryption, compression, and matrix multiplication. Until recently, software on the main processor handled such tasks. Consequently, there are many sophisticated software algorithms, which are often available in C-based implementations.

Reusing efficient, well-validated, and time-tested software algorithms via automatic hardware synthesis for quick architectural exploration and hardware generation can dramatically reduce time to market. Such a path to hardware would spare designers the excessive time required to develop coprocessors from scratch and would reduce the number of sources required to verify the hardware in the RTL simulation, which is quite slow. With this goal in mind, we present a methodology, based on the C2R (C to RTL) compiler, that uses existing C-based implementations of computationally intensive algorithms.[1] We also describe how to systematically perform architectural exploration at the C level and automatically synthesize efficient hardware coprocessors.

Several challenges must be met to achieve this goal. First, the C or C++ implementations target sequential implementation, but their corresponding hardware must be highly concurrent. This requires discovering concurrency opportunities from sequential implementations. Second, simple parallelization of the sequential code does not take into account resource constraints, so we must consider various alternatives for exploiting the available concurrency. Third, depending on the latency, area, and power requirements, there may be various ways to exploit the concurrency. Hence, we must make a choice through proper design space exploration. Fourth, once the proper architected form of the original C code is realized, we must verify that it has the exact same functionality as the original C code. Finally, we must find an effective synthesis solution to generate efficient RTL in Verilog or VHDL from this architected C code.

The process of identifying concurrency in the computation dictated by the given algorithm and then expressing it in concurrent C code using C2R

0740-7475/09/$26.00 © 2009 IEEE Copublished by the IEEE CS and the IEEE CASS

directives is called *restructuring* in C2R jargon. Restructuring imposes hardware architecture atop the computation. Of course, we can make many distinct choices of architectural changes, so an architectural exploration cycle is necessary to find the one that meets all the latency, area, and power constraints. We illustrate our entire methodology through a case study of an Advance Encryption Standard (AES) coprocessor synthesis to RTL Verilog starting from a C implementation of the AES. We also briefly introduce the C2R-based design methodology and discuss some advantages of using a C2R-based design flow. These advantages include quick functional verification and high-level power reduction via different granularities of a clock-gating technique. The main contributions of this article are as follows:

- exposition of the C2R methodology and compiler for coprocessor synthesis;
- demonstration of a case study for architectural exploration for meeting area and speed constraints using C2R; and
- explanation of how to effectively use C-based high-level synthesis (HLS) of coprocessors with the proper surrounding methodology. The "Overview of Related HLS Tools" sidebar discusses research in the area of high-level synthesis.

C2R methodology

In the C2R approach, hardware architecture is defined and explored in the untimed C source by coding the data path and parallelism needed. Using a small set of compiler directives, the compiler generates hardware parallelism, clocks, module ports, and other RTL features. For existing C code, typically only a small percentage of the code is modified during restructuring to synthesize the RTL for the baseline version. The designer has explicit control over the hardware architecture, and the C source code reflects the architecture for the hardware implementation. The restructured C source code can still be compiled using a standard C compiler. The compiler does not introduce any architectural changes during the compilation process; it only takes care of the microarchitecture and clocking.

Modules, processes, and interface functions provide the basic building blocks for defining hardware architecture in the restructured C source code. We briefly explain some of the features here. A C function that has been declared as a process generates a state machine in the RTL model. The C2R compiler can handle different types of hardware storage, including arbitrarily accessed flip-flop storage, internally instantiated RAM, and external memory. The compiler has built-in accessor functions for common on-chip single-port or multiport synchronous RAM. C code frequently includes pointers to indirectly reference a variable. To use the software concept of a pointer in hardware, the designer must associate (bind) the pointer with some specific memory.

Architectural decisions made in C source code largely determine the area and performance of the resulting RTL design. The compiler is generally optimized for performance, because it is the most critical factor for most IP development. There is typically a design-dependent area penalty when using the compiler, compared to what could be achieved with a hand-coded RTL design. But, in many cases, with highly complex algorithms in C, manual translation to the RTL is simply not a practical alternative. Thus, after minimal restructuring, the compiler can compile known functional C software to create hardware implementations of extremely complex algorithms.

C-based verification model

The significant advantage of having a functional software model is the reduction in execution time. C-based execution of the functional model will be several orders of magnitude faster than a corresponding RTL simulation. We can exploit this advantage by testing the model against as many test vectors as possible. Faster execution also helps in cores where the model can be reconfigured in several ways, depending on the design parameters. An extensive permutation of design configuration parameters might not be possible using hardware simulation; however, all possible combinations of various design parameters can be tested in a relatively short time using the functional model. The data collected using such runs can help in characterizing the relationship between the various design parameters. Besides the faster execution time, the C-based model can also exploit the tools and utilities available in the C development environments that have evolved over the years.

Code coverage. In the C2R-based design flow, the same restructured code generates the hardware and executes the functional model. Thus, code coverage performed in the C environment directly correlates

Overview of Related HLS Tools

High-level synthesis is an extensively researched area. Here, we present various HLS tools from industry and academia and discuss how they target high-level specifications for hardware synthesis. In industry, Bluespec,[1] Mentor Graphics,[2] Forte Design Systems,[3] and others provide tools to create synthesizable RTL. Examples of C-based synthesis tools in academia include GAUT[4] and Spark.[5]

Bluespec provides an HLS solution based on rule-based atomic transactions.[1] These transactions are represented using Bluespec System Verilog (BSV); the Bluespec compiler can generate synthesizable RTL from BSV. For verification, the BSV methodology can use either conventional simulation or a formal-verification-based approach that can be applied to the atomic transactional model of computation. Such verification methods are based on term-rewriting systems, the SPIN model checker, and so on. However, designers must learn a new language to develop hardware using Bluespec, and designing from scratch precludes the reuse of existing algorithm implementations.

Catapult C, from Mentor Graphics, takes ANSI C++ as the design input language.[2] This tool facilitates algorithmic development in a single-threaded native-C or native-C++ environment. The main advantage of such a tool is that, unlike structural languages such as VHDL, System Verilog, or even SystemC, the input language expresses only functional behavior. Catapult C lets users control the microarchitecture by applying constraints through Tcl scripts or using a GUI. The microarchitectural details (hardware interface, pipelining, and so on) are not encoded in the source but rather are determined during synthesis.

Cynthesizer, from Forte Design Systems, provides HLS capability for SystemC transaction-level models.[3] These TLMs are pin- and protocol-accurate SystemC models. Cynthesizer uses SystemC clocked thread models, with cycle-accurate communication models to implement the hardware. SystemC wait statements specify the design's cycle-accurate behavior for synthesis. On the verification and simulation side, for the synthesizable subset of SystemC, designers can use the C++ execution environment. This facilitates quick verification and generation of interface and hardware blocks to the RTL.

GAUT is an HLS tool based on C as the design language but mainly applicable for digital-signal processing (DSP) applications.[4] GAUT starts from a bit-accurate specification written in C or C++ and extracts the possible parallelism before going through the conventional stages such as binding, allocation, and scheduling. It has two mandatory synthesis constraints: throughput and clock period. HLS in GAUT uses dataflow graphs (DFGs) generated from GCC (GNU compiler collection) and library characterization of the target library for area, speed, and so forth, to generate the VHDL or cycle-accurate SystemC. This tool can also generate SystemC TLM simulation models for SoC platforms. Because of the specific emphasis on DSP synthesis, however, we could not use GAUT for coprocessor synthesis of the tasks we are concerned with.

Spark is an HLS tool that presents a high-level design methodology. This tool takes behavioral C as the input design language and generates RTL VHDL.[5] The main contributions of this methodology are inclusion of code motion and code transformation techniques at the compiler level to provide maximum parallelism for HLS, and HLS starting from the behavioral C input. Although previous tools were mainly for behavioral-C synthesis, the main thrust today is using C language for generating efficient hardware directly through synthesis.

C2R (C to RTL) is an HLS tool that accepts input specifications as ANSI C code, with minimal directives to provide concurrency to sequential C code.[6] A C2R-based design flow makes it possible to instrument and explore a system architecture in the source code as part of the restructuring process. The restructured C source is functionally equivalent to the desired hardware and also compilable by any native C compiler (such as GCC). Hence, the restructured source can be used for functional verification of the hardware in a native-C development environment. The tool internally creates a control dataflow graph (CDFG) and applies various scheduling and allocation algorithms to generate a Verilog RTL on the basis of the high-level specification.

References

1. G. Singh and S.K. Shukla, "Verifying Compiler Based Refinement of Bluespec Specifications Using the SPIN Model Checker," *Proc. 15th Int'l Workshop Model Checking Software* (SPIN 08), LNCS 5156, Springer-Verlag, 2008, pp. 250-269.
2. "Catapult C Synthesis," Mentor Graphics; http://www.mentor.com/products/esl/high_level_synthesis/catapult_synthesis.
3. "High-Level Design: Today's ESL Design Generation," Forte Design Systems; http://www.forteds.com/highleveldesign/index.asp.
4. "GAUT—High-Level Synthesis Tool from C to RTL," Université de Bretagne-Sud; http://www-labsticc.univ-ubs.fr/www-gaut.
5. "The SPARK High Level Synthesis Methodology," Univ. of California, San Diego; http://mesl.ucsd.edu/spark/methodology.shtml.
6. "C2R Compiler," CebaTech; http://cebatech.com/c2r_compiler.

with the statements executed in the hardware platform. We have used the GCOV (GNU coverage) tool in conjunction with GCC (GNU compiler coverage) to perform code coverage. We've also used this tool to analyze the source code of complex designs to determine code segments in which further architectural exploration and optimization would maximize the returns. For example, the tool provides information regarding the number and probabilities of each path being taken. Hence, in a multiple if-else block, the designer would know which data path to optimize, among multiple paths, to make the overall design efficient. The tool can also help identify untested segments of the restructured code, with test vectors then being generated to test corner-case code segments. Thus, this methodology enables the development and improvement of test suites, which can later be used during hardware simulation.

Functional coverage. Separate, independent functions that perform functional coverage on selected modules can be added to the restructured source code, and calls to these functions can be compiled only during the debug process. The coverage information is useful for determining whether all sections of the selected modules are being accessed. The GCOV tool, along with the functional-coverage code, can provide this information. Conversely, if the coverage information is fixed and known, it can also help characterize input test vectors so that all parts of the code are accessed.

High-level clock gating

Timing, area, and even power-aware decisions can be made at a high level. To control these parameters, intelligent scheduling and resource allocation schemes can be applied during HLS. Because clock-gating-based optimizations greatly reduce the design's total power consumption, knowing the power savings for the generated RTL after HLS can help the designer make power-aware design decisions effectively. Thus, enabling power reduction techniques for use at a high level can help designers make effective high-level decisions during code restructuring.

Clock gating can be enabled for different granularities at a high level, from coarse grained to fine grained, such as clock gating of a function or a register bank of a 32-bit to single-bit register. To facilitate clock gating in high-level descriptions or during HLS, we introduce a few directives such as clock gating (on/off) to control the gating's granularity. Once such transformations are applied, HLS can be enabled to use clock gating at various granularities.[2] Including these features in the design methodology is important for unifying the design flow around HLS rather than making HLS another part of such a flow. In our experiments, when area and timing results for the RTL were generated from HLS, lower-level implementations were usually similar. In many cases, HLS helps provide more reduction opportunities because of better visibility for the registers' enable (or gating) conditions.

Table 1. Percentage of power reduction compared to the original design without clock gating.

	Power reduction using clock gating (%)			
Benchmark	**1 bit**	**4-bit bank**	**8-bit bank**	**16-bit bank**
Fibonacci	16.32	16.41	16.32	NA
Caesar	37.78	38.22	38.22	NA
Viterbi	3.98	3.98	3.98	NA
AES	30.74	30.74	29.83	26.42
Gzip	16.73	16.39	14.96	12.29

Facilitating clock gating in HLS facilitates power-aware design exploration and leads to better-quality results for the RTL generated by the HLS tools. Table 1 represents the clock-gating-based power reduction that we applied to various register bank sizes: 1 bit, 4-bit bank, and so on. Each entry for the benchmarks represents how much power savings could be achieved with respect to the non-clock-gated RTL design. For example, for the AES design, up to 30% power savings were possible simply from doing clock gating. HLS took a few seconds to generate different RTLs.

We have compared the quality of generated RTL with respect to the RTL synthesis tool power compiler, as Table 2 shows. Our results for area, timing, and power were comparable to those obtained using the power compiler on the tested benchmarks. We used 180-nm technology libraries for our experiments.[3] For example, for the AES design compared to the power compiler, there was almost a 4% reduction in area, no impact on timing, and a 25% savings in power consumption. Our proposed algorithm performs control flow analysis of an enable or gating condition for clock-gating registers or register banks. This helps determine global enabling or gating opportunities to clock-gate the design for various

Table 2. Comparing our approach with RTL clock gating.

	Our results compared with RTL technique (%)*		
Benchmark	**Area**	**Timing**	**Power**
Fibonacci	89.70	100.00	94.89
Caesar	112.10	100.00	91.03
Viterbi	100.20	105.99	96.02
AES	95.77	100.00	74.58
Gzip	100.36	100.00	100.36

* Power compiler numbers for area, timing, and power are normalized to 100.

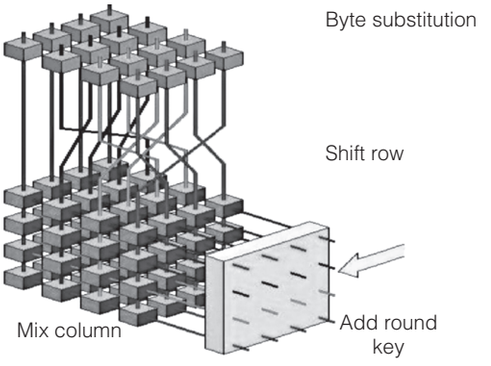

Figure 1. Four stages of Advanced Encryption Standard (AES) encryption (Source: © John Savard.)

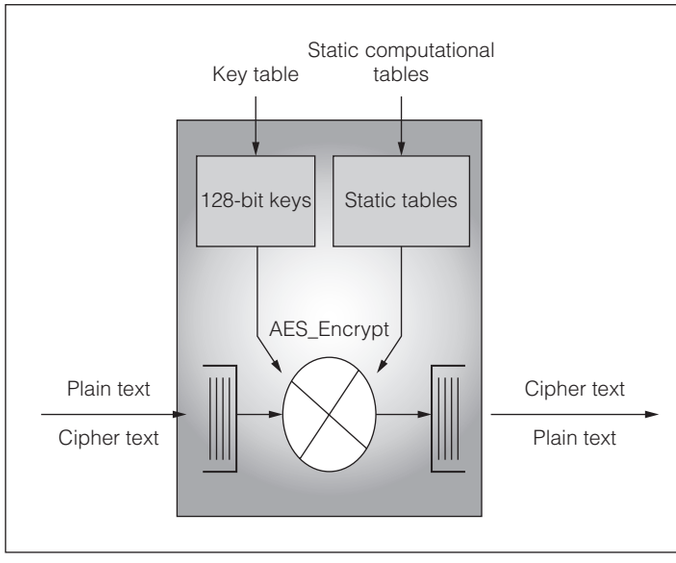

Figure 2. Block diagram of the AES algorithm.

granularities. Details of our comparison with the power-compiler and clock-gating methodology are available elsewhere.[2]

Advanced Encryption Standard

AES is a block cipher that works by encrypting groups of 128 message bits. Three different-length keys (128 bits, 192 bits, and 256 bits long) can be used in the AES encryption process. We selected the 128-bit key length for use in our case study. AES operates on a 4×4 array of bytes (16 bytes, or 128 bits), termed the *state*. The 128-bit algorithm consists of 10 steps called *rounds*. For encryption, each round of AES (except the last one) involves four stages:

- `AddRoundkey`. Each byte of the state is combined with the round key, and each round key is derived from the cipher key using a key schedule.
- `ByteSub`. This is a nonlinear substitution step in which each byte is replaced with another according to a lookup table (LUT).
- `ShiftRows`. In this transposition step, each row of the state shifts cyclically a certain number of positions.
- `MixColumns`. This mixing operation operates on the columns of the state, using a linear transformation to combine the four bytes in each column.

The final round replaces the `MixColumns` stage with another instance of `AddRoundkey`. Figure 1 captures these four stages. Each round of the AES algorithm requires access to four static tables, with each table being accessed four times, resulting in a total of 16 static table reads. In addition, there are four accesses to a key table, for a total of 20 memory accesses per round. This table-access characteristic of AES provides an opportunity for making several design trade-offs between performance (throughput and latency) and area. Figure 2 shows a simple block diagram of the AES algorithm, including the static tables and the dynamic 128-bit key table.

The C source code contains both *rolled* and *unrolled* versions of the algorithm, where the rolled version consists of a single *for* loop. Each pass through the *for* loop is one round of the encryption algorithm. We used the rolled version as the starting point for the baseline restructuring of the C code.

Architectural exploration

Here, we explain how to easily visualize various architectures in the C2R methodology. We explore

the baseline and three optimized versions of the hardware implementation of the AES Rijndael algorithm via the C source code restructuring process (the C source code is from http://www.efgh.com/software/rijndael.zip). The baseline represents the minimal restructuring needed to enable the C2R compiler to generate a synthesizable Verilog RTL design implementation. We chose an application IP (AES) of reasonable size and a good candidate with which to explain how to use C2R for quick design space exploration. Various other quick design space exploration examples using C2R for design IP blocks have been implemented, including gzip, gunzip, data encryption standard (DES) hardware blocks used for compression and decompression, and security-related applications.[1]

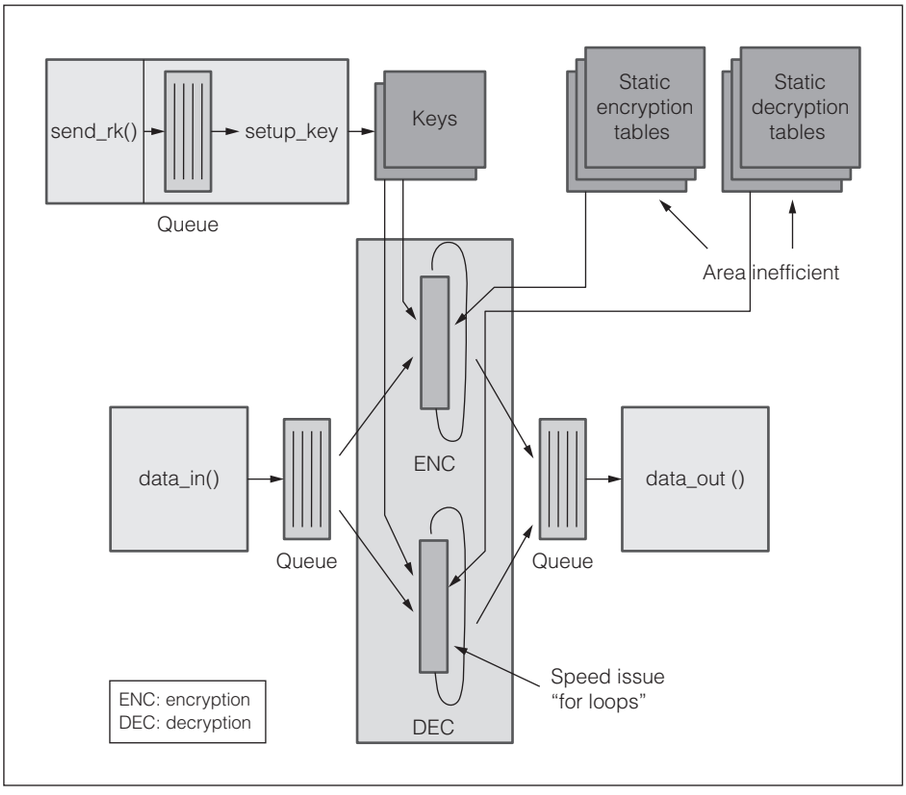

Figure 3. Baseline architecture for AES.

AES restructuring iterations

We investigated different architectural alternatives for AES. These alternatives can be generated by doing a little restructuring in the C code. These restructuring iterations include a baseline hardware architecture, a pipelined architecture, a block RAM (BRAM)-based architecture, and an architecture optimized for performance and area.

Baseline hardware architecture. Baseline restructuring, which is based on the rolled version of the AES algorithm, introduces three interface functions (shown as queues in Figure 3) to define the top-level module ports and data I/O queues of the hardware design for input data, output data, and round key (`rk`) input. The encrypt/decrypt C function is declared with the `c2r_process` directive, which causes this function to be realized as a finite-state machine (FSM) at the RTL. The interface functions (declarations and definitions) and process declarations represent the primary changes to the C source code for baseline restructuring. The other code changes basically consist of declarations of bit-width-specific data types for C variables to minimize hardware storage requirements. As Figure 3 shows, the hardware implementation contains encryption and decryption. Both these functions (FSMs in hardware) require interaction with static encryption, decryption, and key tables. The encryption function contains the *for* loop, which executes the 10 rounds in the algorithm. Figure 4 shows the list of function prototypes using C2R directives.

The Rijndael algorithm requires 10 large static arrays: five for encryption and five for decryption. By default, the static-table read-only storage compiles and synthesizes to wire and gates, such that the memories are simultaneously accessible in one clock cycle. The round-key data storage becomes flip-flops in the hardware, again allowing parallel, single-clock accesses for higher performance, but at the cost of area. With the rolled version of the code, even with

```
/function prototypes
extern uint8_t c2r_interface enc_data_in (uint8_t
  plaintxt, uint1_t encrypt);
extern uint8_t c2r_interface enc_data_out (void);
extern uint8_t c2r_interface send_rk (uint32_t rkkey);
extern void c2r_process encrypt (void);
```

Figure 4. C2R directives used for different functions.

Table 3. AES experiment results using C2R.

AES version	Frequency (MHz)	Throughput	Flip-flops	Slices, look-up tables	Block RAM (Mbytes)
Baseline	47.03	501 Mbps	4,890	50,784, 96,422	0
Pipelined	109.20	14 Gbps	4,662	27,760, 54,068	0
BRAM based	126.50	80.9 Mbps	5,425	3,679, 4,861	12
Speed and area optimized	144.00	18 Gbps	5,328	7,670, 14,588	80

parallel accesses to stored data, 12 clock cycles are still required to perform the 10 rounds of encryption, resulting in a throughput of 501 Mbps, as Table 3 shows. We obtained these numbers on a Xilinx 4-input, LUT-based Virtex IV FPGA.

Pipelined architecture. As Figure 3 shows, an obvious performance deficiency exists in the baseline architecture because a new 128-bit block of input data might not enter the *for* loop until the current block has completed all 10 rounds (note that there is no pipelining in this implementation). As previously discussed, there are 20 accesses to stored data in each round, so the number of clock cycles required to complete a round depends on how the storage is implemented in the hardware.

This iteration of the hardware architecture focuses on increasing the throughput. We used the unrolled version (the *for* loop unrolled) of the Rijndael algorithm as the starting point. The baseline suffered because a new block of data had to wait until the current block completed all 10 rounds. The logical solution is to introduce pipelining into the architecture by making each round a separate process. Figure 5 captures one such process, a 10-stage pipeline that produces a new block of encrypted (or decrypted) data every clock cycle. This new pipelined architecture delivers a throughput of 14 Gbps—about 28 times higher than the throughput of the baseline design.

```
void c2r_process enc_round1(void)
{
while (start_round1_flag == 0)
{
c2r_wait();
}
/* round 1: */
s10 = Te0[s0 >> 24] ^ Te1[(s1 >> 16) & 0xff] ^ Te2[(s2 >>
     8) & 0xff] ^ Te3[s3 & 0xff] ^ rk[4];
s11 = Te0[s1 >> 24] ^ Te1[(s2 >> 16) & 0xff] ^ Te2[(s3 >>
     8) & 0xff] ^ Te3[s0 & 0xff] ^ rk[5];
s12 = Te0[s2 >> 24] ^ Te1[(s3 >> 16) & 0xff] ^ Te2[(s0 >>
     8) & 0xff] ^ Te3[s1 & 0xff] ^ rk[6];
s13 = Te0[s3 >> 24] ^ Te1[(s0 >> 16) & 0xff] ^ Te2[(s1 >>
     8) & 0xff] ^ Te3[s2 & 0xff] ^ rk[7];
enc_run_flag = 0;
start_round2_flag = 1;
start_round1_flag = 0;
} // end enc_round1
```

Figure 5. Source code changes applied to the pipelined implementation.

```
void c2r_process enc_round1(void)
extern void c2r_foreign c2r_accessor c2r_memory
   c2r_useclock(clk) c2r_timing_model(C2R_SYNC) bram (
int c2r_width(fordepth(44)) *address,
uint1_t write_enable,
uint32_t write_data,
uint32_t read_data);
static uint32_t c2r_use(bram) rk[44]; //binds rk to bram
```

Figure 6. Memory accessor code to bind the round-key input to FPGA block RAM (BRAM).

BRAM-based architecture. Both the baseline and pipelined architectures allowed storage to be implemented in gates (FPGA LUTs) and flip-flops, resulting in fairly large areas, as illustrated by the FPGA slice and LUT counts in Table 3. These implementations might be suitable for an ASIC design, where gate and flip-flop storage would be quite efficient, but a more area- and LUT-efficient design is needed for an FPGA-based solution. To achieve a lower slice and LUT count, it is desirable to use on-chip BRAM resources in the FPGA. The way to accomplish this is via special accessor functions that define the interface to hardware memories. Such accessor functions at the C source code level instruct the compiler to generate Verilog RTL, automatically instantiating BRAM. Figure 6 shows a snippet of the code block

for the memory accessor. This dramatically reduces the number of slices and LUTs required, but the use of shared memories requires the full 20 clocks per round for data accesses. Table 3 captures the results.

Architecture optimized for performance and area. The goal for the final iteration was to achieve optimal efficiency: high-performance throughput with a reasonably small area (in terms of FPGA slices and LUTs). This architecture used the unrolled version of the AES algorithm, pipelined architecture, accessor functions, and a larger number of BRAMs to enable parallel access to stored data for each round. Instead of taking 20 clock cycles to access data, only 1 clock is required in this implementation. As Table 3 shows, the resulting throughput is 18 Gbps, and the LUT count is about one-fourth that of the pipelined architecture. (Additional reductions in LUT count can be realized by further increasing the number of BRAMs.)

The AES algorithm presented in this article illustrates the ease of adoption and productivity benefits of a C2R compiler and C-based hardware design flow for electronic-system level (ESL) design. Extensive ANSI C syntax support means that existing C code bases can be compiled with minimal changes to the source, and programmers can take advantage of the full power of the C language. It's possible to compile the input specification in ANSI C, including structures, unions, global variables, pointers to structures, complex arrays, multilevel indirection, and pointer arguments to function calls. Although theoretically such an architectural selection could occur automatically, in reality complete automation of concurrency opportunities is very difficult or is limited to a specific domain.[4] Our approach relies on the fact that architecture-level decisions are controlled by the architect, whereas the HLS tools handles microarchitectural control.

The four restructuring iterations discussed here were completed very quickly, and they had dramatically different results (throughput varying from 81 Mbps to 18 Gbps). These iterations show how different architectures can be explored at the system level through small changes to the C source code. The fact that the restructured ANSI C code can be compiled using GCC means that verification can be accomplished in a native-C environment, and the same source drives both architectural exploration and hardware implementation.

Thus, staying in the native-C environment for architectural exploration combined with compiler-based power-saving insertions can lead to the design of fast coprocessors, which can speed up computations by several orders of magnitude.[5] For a true ESL development methodology, it's important to work at different levels, such as the untimed C level—focusing on various objectives, ranging from verification to hardware development. Even if HLS cannot be used for all general-purpose hardware synthesis, certain kinds of hardware synthesis can be achieved with proper HLS and surrounding methodologies. ■

References

1. "C2R Compiler," CebaTech; http://cebatech.com/c2r_compiler.
2. S. Ahuja, W. Zhang, and S.K. Shukla, *A Methodology for Power Aware High-Level Synthesis of Co-Processors from Software Algorithms,* tech. report 2008-08, Virginia Polytechnic and State University, Fermat Lab; http://fermat.ece.vt.edu/Publications/pubs/techrep/techrep0808.pdf.
3. J.E. Stine et al., "A Framework for High-Level Synthesis of System-on-Chip Designs," *Proc. IEEE Int'l Conf. Microelectronic Systems Education,* IEEE CS Press, 2005, pp. 67-68.
4. "GAUT—High-Level Synthesis Tool from C to RTL," Université de Bretagne-Sud; http://www-labsticc.univ-ubs.fr/www-gaut.
5. E. Simpson et al., "VT Matrix Multiply Design for MEMOCODE '07," *Proc. 5th IEEE/ACM Int'l Conf. Formal Methods and Models for Codesign* (MEMOCODE 07), IEEE CS Press, pp. 95-96.

Sumit Ahuja is pursuing a PhD in electrical and computer engineering at Virginia Polytechnic and State University. His research interests include high-level synthesis, power estimation and reduction, and hardware-software codesign. He has an MEng in embedded systems design from the Advanced Learning and Research Institute in Lugano, Switzerland. He is a member of the IEEE.

Swathi T. Gurumani is part of the Hardware IP team at CebaTech, which generates complex IP blocks using the C2R synthesis tool. His research interests include hardware-software codesign, parallel processing, and high-level synthesis. He has a PhD in computer engineering from the University of Alabama

in Huntsville. He is a member of the Phi Kappa Phi and Eta Kappa Nu honor societies.

Chad Spackman is the CTO and cofounder of CebaTech. His technical interests include electronic system-level (ESL) methodology development and IP design. He has an MS in engineering from Penn State University.

Sandeep K. Shukla is an associate professor of electrical and computer engineering at Virginia Polytechnic and State University. His research interests focus on formal methods and their applications to system design. He has a PhD in computer science from the State University of New York at Albany. He is a senior member of the IEEE and the ACM.

■ Direct questions and comments about this article to Sumit Ahuja, FERMAT Lab, ECE Dept., 340 Whittemore Hall, Virginia Polytechnic and State University, Blacksburg, VA 24061; sahuja@vt.edu.

ADVERTISER INFORMATION

JULY/AUGUST 2009 • *DESIGN & TEST*