High-Throughput Programmable Cryptocoprocessor

A loosely coupled cryptocoprocessor based on the Advanced Encryption Standard combines high throughput with programmability. Using domain-specific instructions and design principles such as control hierarchy and block pipelining, the security engine supports Internet Protocol security and other networking applications.

Alireza Hodjat and Ingrid Verbauwhede
University of California, Los Angeles

High-speed Internet Protocol security (IPsec) applications require high throughput and flexible security engines. Virtual private networks, for example, require a throughput of over 2 gigabits per second. IPsec uses the Advanced Encryption Standard[1] algorithm in various operation modes.[2] Most security applications combine AES and block ciphers in general with different operation modes because the straightforward electronic code book (ECB) mode is vulnerable to statistical attacks.[3] The US National Institute of Standards and Technology recommends block cipher modes of operation,[4] which, in addition to ECB, include cipher block chaining (CBC), counter, cipher feedback (CFB), output feedback (OFB), and CCM, a new mode that combines the counter and CBC-MAC (message authentication code) modes. CCM only requires the encryption algorithm and can generate encrypted and authenticated data simultaneously.[5] As the "Related Work on Programmable Security Engines" sidebar mentions, no current systems support all four modes: ECB, CBC, counter, and CCM.

Recent Internet Society Request for Comments (RFC) efforts propose combining AES with block cipher modes, such as AES in counter mode with IPsec[6] and AES in XCBC-MAC with IPsec.[7] Other researchers use AES in counter and CCM modes for IPsec.[8] Standard proposals tend to change, but these changes are usually limited to initialization, setup, key management, and so on. Combining programmability with high throughput supports a wide range of current and future standards for security applications.

A high-speed CPU is one way to implement security primitives. However, factors such as memory bandwidth and cache misses prevent the CPU from achieving multi-Gbps throughput. The "AES/Rijndael: Speed" Web site (http://www.tcs.hut.fi/~helger/aes/rijndael.html) reports AES throughput on various CPUs at over 1 GHz. Optimized C code compiled with gcc (GNU Compiler Collection) 3.0.2 achieves only 861 Mbps on a 2.25-

Published by the IEEE Computer Society

Related Work on Programmable Security Engines

Ravi et al. present a system-level design methodology for programmable security processor platforms.[1] It uses Tensilica's Xtensa processor[2] and includes customized instructions, which improve performance from less than one Mbps to several tens of Mbps. In the instruction set extension approach, which Barat, Lauwereins, and Deconinck refer to as a *tightly coupled processing scheme*, the main CPU (Xtensa) is customized for a specific domain by adding a new functional unit to its pipeline.[3] Custom instructions flow through the pipeline and the new functional unit decodes and executes them. Our approach differs in that we use loosely coupled, independent coprocessors in conjunction with a main embedded processor core. These programmable coprocessors are designed for specific domains and attached to the main processor on a dedicated interface.

A typical embedded system contains multiple tasks that might need acceleration—for example, network protocol processing in the networking domain, image or speech processing in the digital signal processing (DSP) domain, and authentication and privacy protection in the security domain. Figure A1 shows the stream of data samples that typically flow from the DSP unit to the security unit and continue to the networking unit.

continued on p. 36

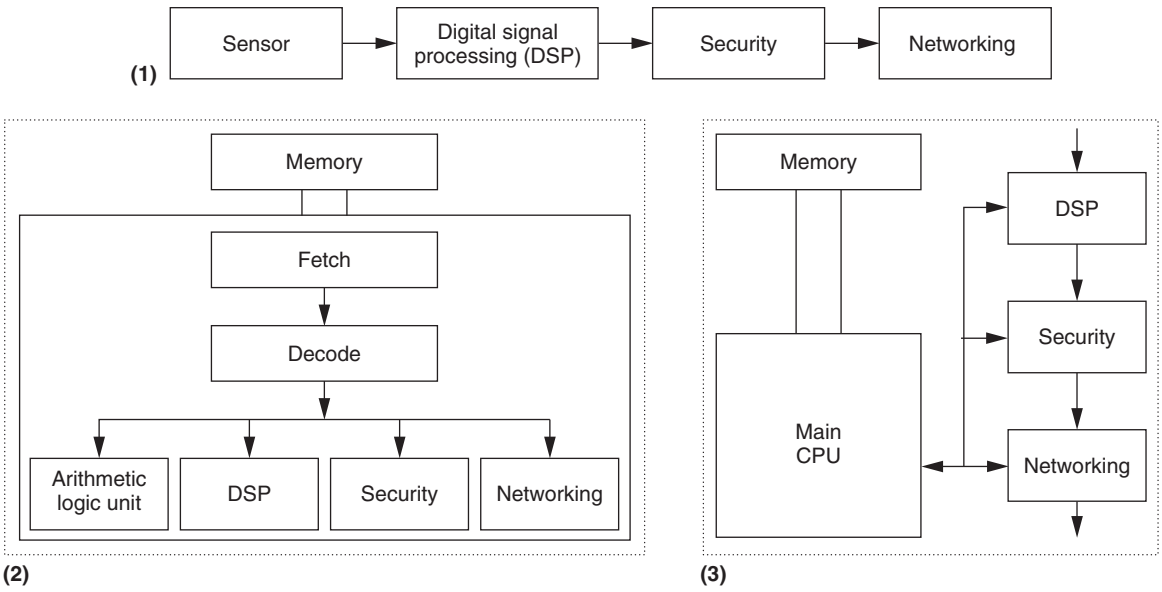


Figure A. Embedded security system: Typical data sample stream from the DSP unit to the security and networking units (1); system design using a tightly coupled instruction set extension (2); and system-level view of our design (3).

GHz AMD Athlon. A hand-optimized assembly code of the AES algorithm achieves up to 718 Mbps on a 1.33-GHz Pentium III and up to 1,436 Mbps on a 3.06-GHz Pentium IV. The CPUs achieved these throughputs in ideal circumstances; the AES was the only algorithm running, so there was no overhead for other tasks.

We have developed a high-throughput, programmable cryptocoprocessor that runs the AES algorithm in different operation modes for IPsec applications. Instead of using multi-GHz CPUs, we use domain-specific processors to obtain the required throughput. Domain specialization helps close the gap between performance and programmability. The cryptocoprocessor achieves a maximum throughput of 3.43 Gbps at a 295-MHz clock frequency using 0.18-micron CMOS technology. The instruction set includes initialization, key setup, and AES encryption for different operation modes. Block pipeline instructions allow AES to run in ECB, CBC-MAC, counter, and CCM modes in 11 clock cycles per 128-bit block without loss in throughput compared to an AES without a mode of operation.

Architecture

The cryptocoprocessor architecture consists of three modules. These are input module,

continued from p. 35

Figure A2 shows how these systems can be designed using a tightly coupled instruction set extension. The processor is customizable for each domain by adding functional units to the pipeline. This way, the corresponding functional units decode and execute the domain instructions. Figure A3 is a system-level view of our design. Programmable coprocessors meet the throughput requirements for each domain. The main embedded processor programs each domain-specific coprocessor. Thus, the embedded processor exercises control while data is transferred between coprocessors.

Another related system is the CryptoManiac, a coprocessor for cryptographic workloads.[4] Its domain-specific processor performs cryptographic functions on the data path through its processing elements. The processing elements support various cryptographic algorithms, thus creating some overhead. In the AES algorithm, CryptoManiac performs 624 Mbps at 390-MHz clock frequency.

Recent publications report ASIC implementations of the AES algorithm.[5–8] Hifn's storage security processor,[6] for example, uses 0.13-micron technology, achieving 2 Gbps at 133 MHz, and can be used in the counter and CBC modes. Carlson et al.[7] present another implementation in 0.13-micron technology that operates only in ECB mode and achieves 2.18 Gbps at 500 MHz. Satoh et al.[8] report a compact AES implementation in 0.11-micron technology that runs at 2.6 Gbps at 224 MHz for the ECB and CBC modes. None of these cases support all four modes: ECB, CBC, counter, and CCM.

References

1. S. Ravi et al., "System Design Methodologies for a Wireless Security Processing Platform," *Proc. 39th Design Automation Conf.* (DAC 02), ACM Press, 2002, pp. 777-782.
2. *Xtensa Application-Specific Microprocessor Solutions—Overview Handbook*. Tensilica, 2001.
3. F. Barat, R. Lauwereins, and G. Deconinck, "Reconfigurable Instruction Set Processors from a HW/SW Perspective," *IEEE Trans. Software Eng.*, vol. 28, no. 9, Sept. 2002, pp. 847-862.
4. L. Wu, C. Weaver, and T. Austin, "CryptoManiac: A Fast Flexible Architecture for Secure Communication," Proc. 28th Int'l Symp. Computer Architecture (ISCA-01), IEEE CS Press, 2001, pp. 110-119.
5. T. Ichikawa et al., "Hardware Evaluation of the AES Finalists," Proc. 3rd AES Candidate Conf., ACM Press, 2000, pp. 279-285.
6. "Hifn HIPP III 4300 Storage Security Processor," http://www.hifn.com/products/4300.html.
7. D. Carlson et al., "A High-Performance SSL IPsec Protocol Aware Security Processor," Proc. Int'l Solid-State Circuits Conf. (ISSCC 03), UEEE Press, 2003, pp. 142-483.
8. A. Satoh et al., "A Compact Rijndael Hardware Architecture with S-Box Optimization," Proc. AsiaCrypt 2001, LNCS 2248, Springer-Varlag, 2001, pp. 239-254.

output module, and the encryption module, which includes the AES core.

AES core

Figure 1 shows the AES core's architecture. It implements the 128-bit key, 128-bit data version of the AES algorithm and performs encryption in 11 cycles, with one round of the algorithm executing in one clock cycle. AES-128 execution takes 10 rounds, leaving one clock cycle for the initial key-addition phase.

We optimized the AES core for speed, with a goal of minimizing delay for one round. The substitute-phase (S-boxes) is implemented using lookup tables; all other steps in each round are XOR chains. Other alternatives for implementing S-boxes exist,[9,10] but we found that the straightforward implementation—lookup tables—is fastest.[11] Therefore, the core performs each round in a single clock cycle optimized for minimum combinational delay.

Cryptocoprocessor

As Figure 2 shows, the cryptocoprocessor includes

- the input and output modules, which perform handshaking to read the input and write the encrypted data;
- the encryption module, which contains logic to run AES in the ECB, CBC-MAC, counter, and CCM modes; and
- the top controller, which issues commands to the other three modules.

The cryptocoprocessor includes four 32-bit I/O interfaces. The input and output modules can read or write a 128-bit block of data using two of the interfaces—one for data input and one for output—asynchronously. The other two interfaces are synchronous; the main CPU core and the coprocessor use them—one as input and the other as output—for data communication.

Memory-mapped interface with host CPU. Figure 3a shows how the cryptocoprocessor attaches to a CPU core through the memory-mapped interface. Four registers connect the host CPU to the cryptocoprocessor: instruction, configuration, 32-bit input, and 32-bit output. The host CPU can read or write to these registers

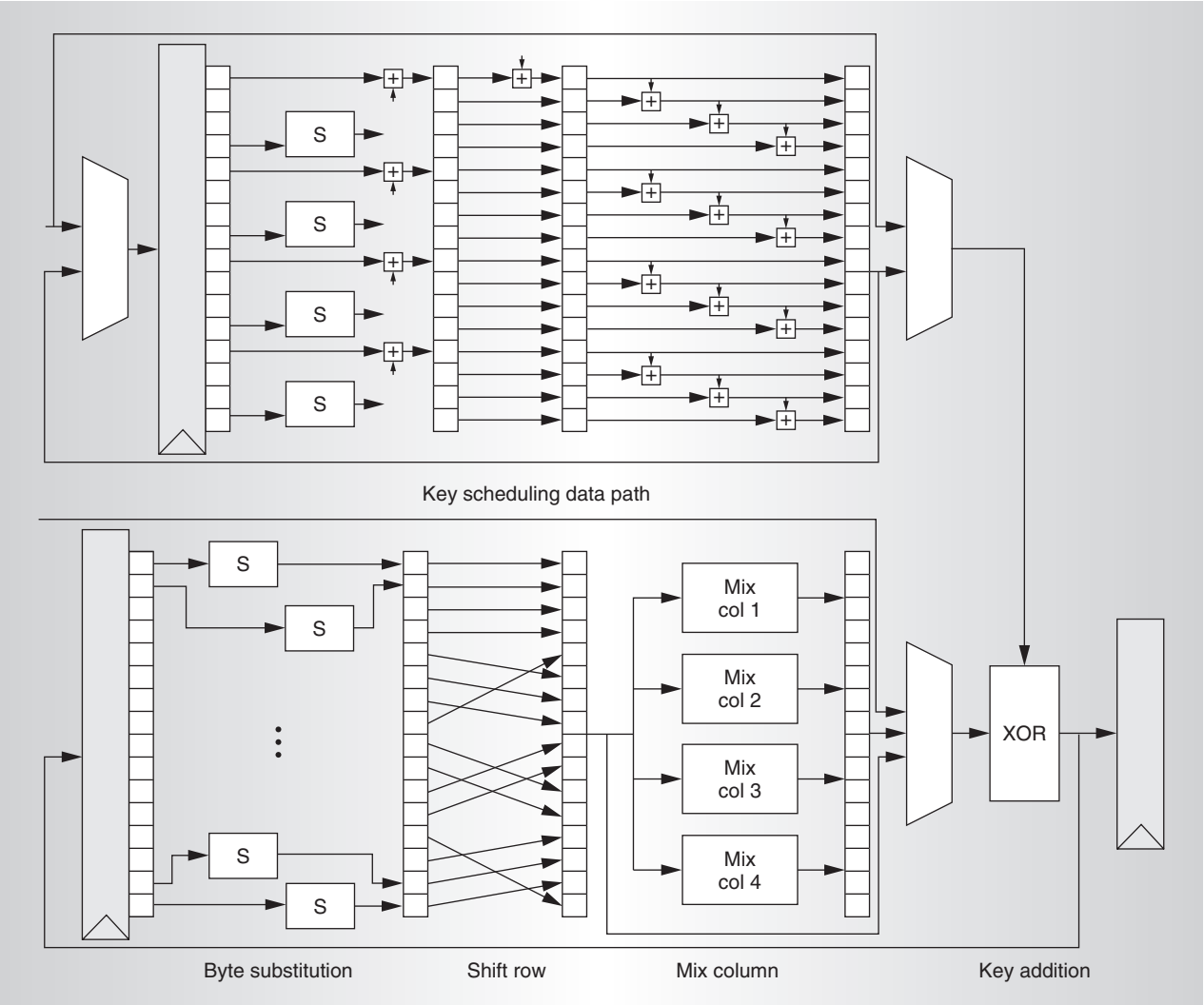


Figure 1. Advanced Encryption Standard core architecture.

by accessing different memory locations. The memory-mapped interface decodes the memory addresses and updates the register values. The main CPU can therefore easily program the cryptocoprocessor.

The CPU programs the coprocessor through the 8-bit instruction and configuration registers. Moreover, the main CPU core and the coprocessor use the 32-bit input and output registers for data communication. The main CPU uses these registers for key setup and initialization of the CBC-MAC, counter, and CCM operation modes. Therefore, it is possible to change the key and initial vector values in the software.

Asynchronous I/O interfaces. The input and output interfaces use two handshaking signals to read and write a 128-bit data block asynchronously in multiple clock cycles. Figure 3b shows how the cryptocoprocessor connects to the input and output modules through these interfaces. The modules work independently of the cryptocoprocessor and the main CPU host and can be programmed through the CPU core's memory-mapped interface. The modules produce data for the cryptocoprocessor and use the coprocessor's encrypted output.

Design principles

Several design principles let the cryptocoprocessor achieve the required throughput for IPsec and other networking applications.

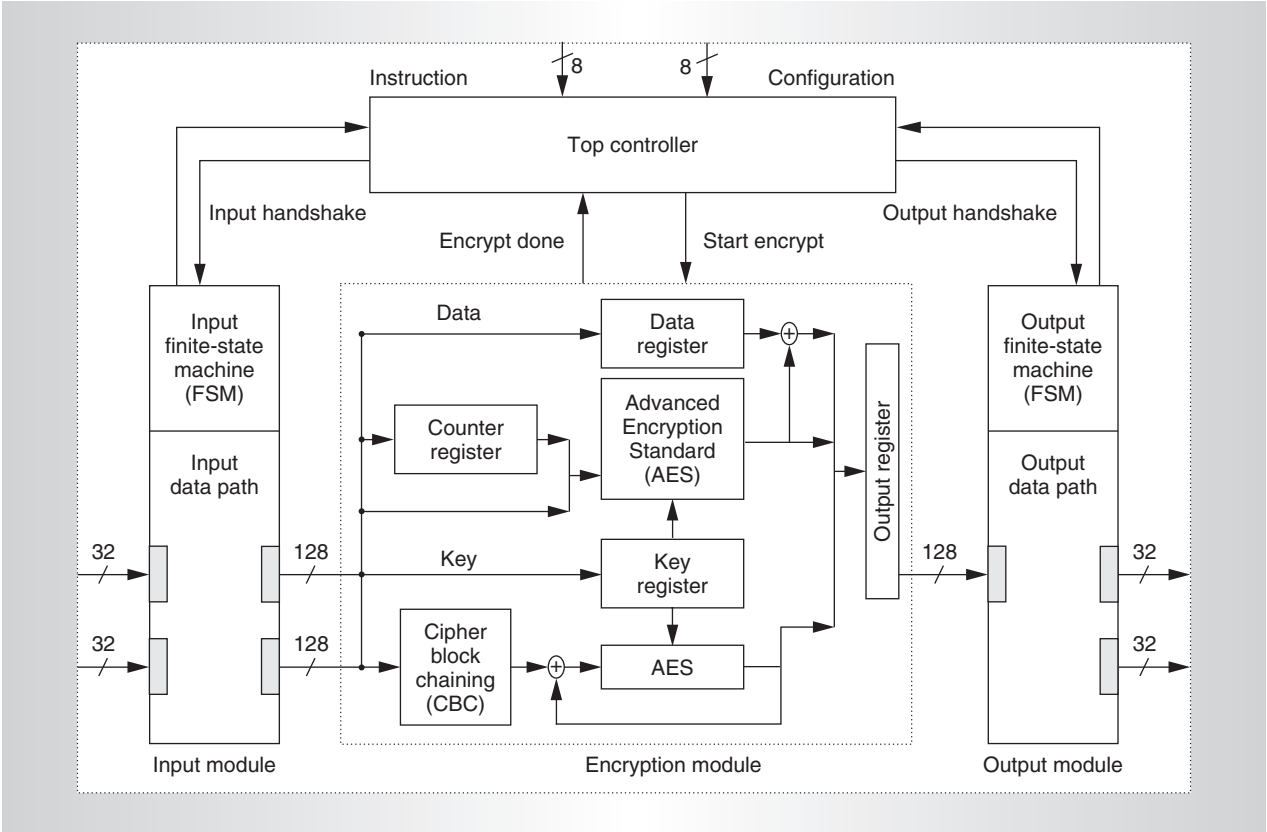


Figure 2. Cryptocoprocessor block architecture.

Separate data and control streams

Separating data and control streams enables high-throughput data encryption and a high level of programmability. In Figure 2, data flows through the coprocessor from the input module to the encryption module and then to the output module while the top controller handles instructions. The input and output finite-state machines (FSMs) perform handshaking to read input and write encrypted data without interference from the top controller. Following this methodology, we can program the coprocessor to encrypt the input data stream and produce output continuously while the top controller interface processes new instructions.

Control hierarchy

Designing with multiple controllers requires partitioning the control over different modules, particularly when multiple modules communicate asynchronously. Hierarchical control design simplifies the controllers' communications and lets us combine high performance and programmability. Harel proposed a control hierarchy for specification in Statecharts.[12] We propose this design technique for implementation. We implement the system's top-level control in the main processor core. Instructions from the main embedded CPU bring commands to the coprocessor's top controller. The top controller also controls the lower-level modules.

Figure 4 shows the control hierarchy for the cryptocoprocessor in Figure 2. The top controller unit manages the input FSM, CBC FSM, counter FSM, and output FSM. As mentioned, the input and output FSMs perform the handshaking sequence for reading and writing of the 128-bit data blocks. The CBC FSM controls the encryption sequence to generate a CBC-MAC, and the counter FSM controls the encryption sequence for the counter operation mode.

Depending on which instruction it reads, the main controller asserts the start signal for a subcontroller. The submodule starts its operation, asserting the done signal when fin-

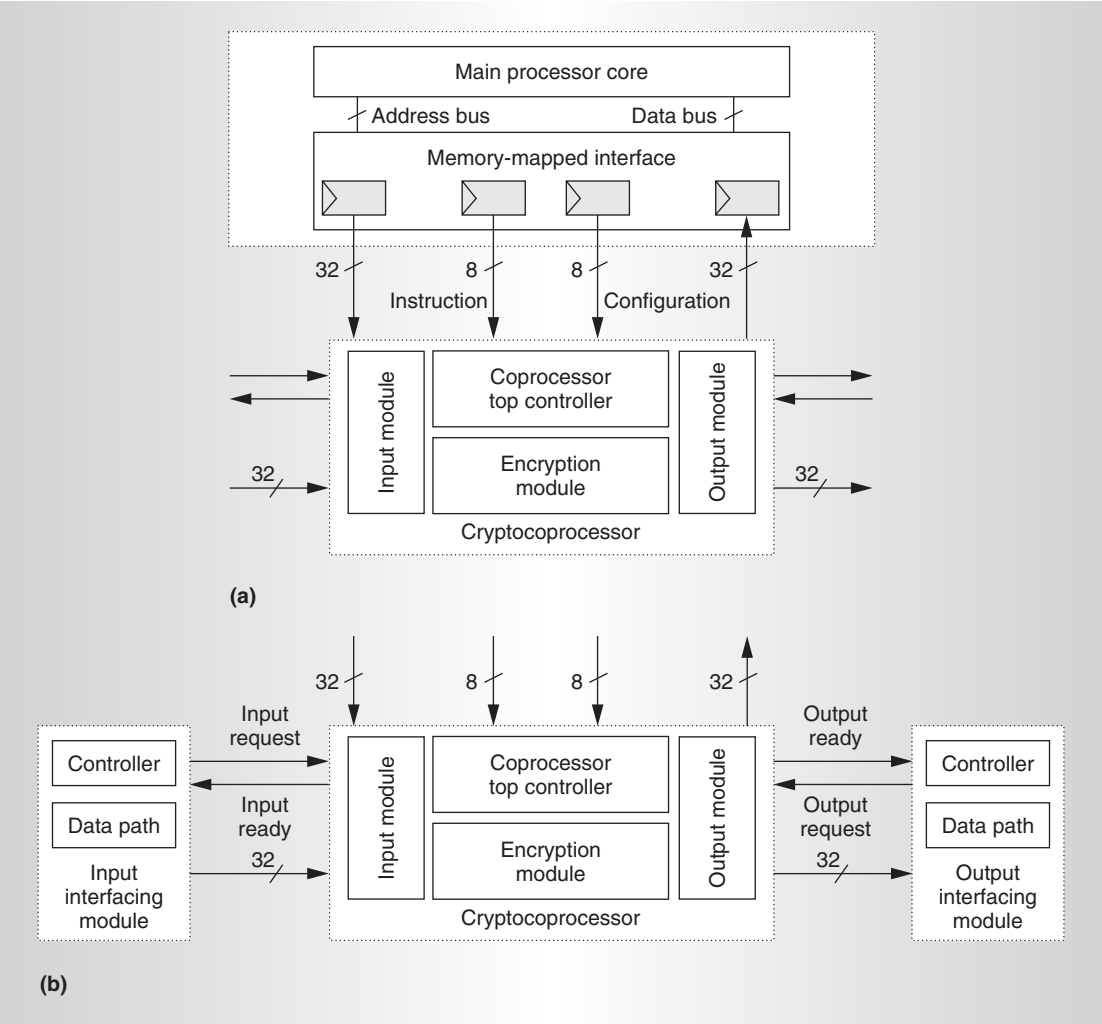


Figure 3. Cryptocoprocessor interfaces: memory-mapped interface with main CPU core (a) and asynchronous I/O interface with input and output modules (b).

ished. The done signal lets the main controller reassert the start signal for subsequent instructions.

Block pipelining

Because the CBC-MAC and CCM modes involve feedback, we cannot pipeline the AES core. We use a block pipelining technique to achieve multi-Gbps throughput for high-throughput applications. The hierarchical control design and the handshaking interface between modules enable this technique. Figure 5 shows how we designed this pipeline for AES in the ECB mode. In steady state, the coprocessor encrypts data while it reads new data from the input and writes the previous encryption result to the output. In this methodology, the slowest block (the encryption module) decides the cycle time. Encrypting one block of data in any of the supported modes takes 11 cycles. Therefore, one block of output will be ready every 11 cycles.

Single and continuous instructions

We use two categories of instructions. *Single instructions* perform one task and execute in a finite number of cycles. One-cycle single instructions move data from a specific module or register to another module or register and are required for key setup and initialization. Examples of multiple-cycle single instructions include single-block encryption, reading one block of data or key, and writ-

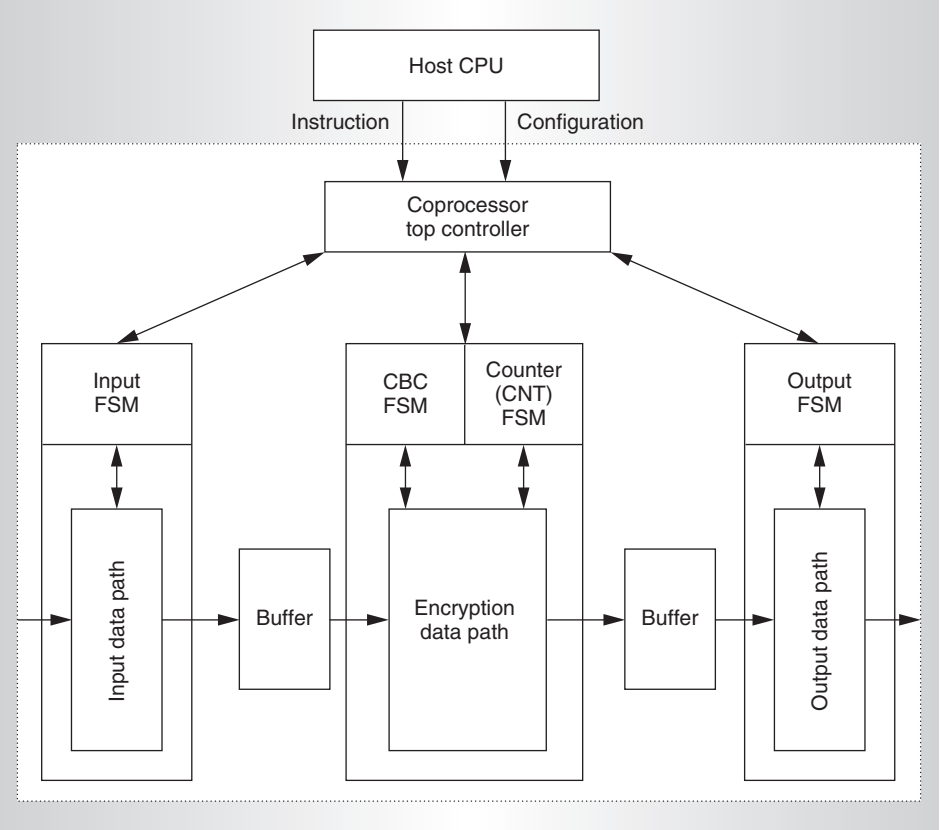


Figure 4. Hierarchical control design.

ing one block of output. *Continuous instructions* provide high-throughput encryption streams using block pipelining. The most important continuous instruction is Encryption_pipeline_begin, which performs block pipeline encryption in any operation mode. Continuous instructions let the coprocessor encrypt the data stream and write the result to the output continuously, until the main CPU issues a done instruction.

For example, consider the CCM operation mode: As Figure 6a shows, each input block goes to both the AES CBC-MAC and the AES counter modules. Using AES in feedback mode, the CBC-MAC module generates the MAC value, which is used for authentication. In parallel, the counter module encrypts the input payload for confidentiality. Figure 6b shows the CCM pipeline model. The Encryption_pipeline_begin instruction, along with CCM mode configuration, starts the pipeline. In steady state, the coprocessor reads a new block of data (IN) and encrypts a new counter value (CNT). It XORs the previous encrypted counter value with the previous input and writes it to the output (OUT). The previous input block is XORed with the last CBC-MAC value and is encrypted (CBC). This continues until the main CPU inserts Encryption_pipeline_done. In the last pipe stage, the coprocessor calculates the encrypted MAC value and writes it to the output.

Modularity

As mentioned earlier, the cryptocoprocessor architecture has a modular design. For example, we could replace the AES core with an implementation of any other symmetric-key encryption algorithm without changing the control hierarchy and programming interfaces.

Programming interface

Table 1 lists the instructions supported by our cryptocoprocessor. Different types of read input and write output instructions load the data or key from the input and return the result to the output. Moreover, various types of encryption instructions can refresh the

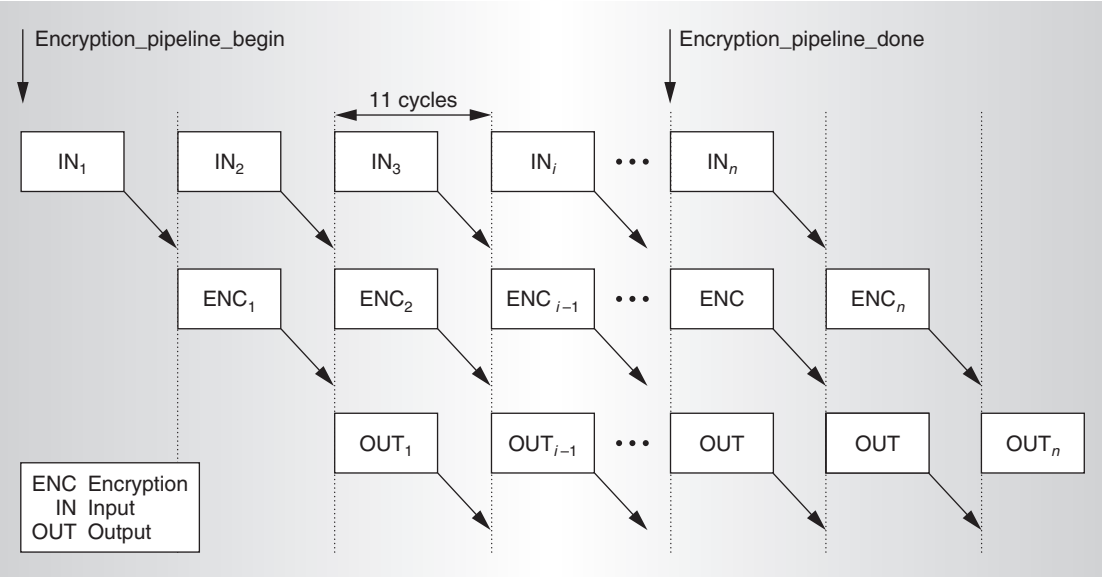


Figure 5. Block pipelining for electronic code book (ECB) operation mode.

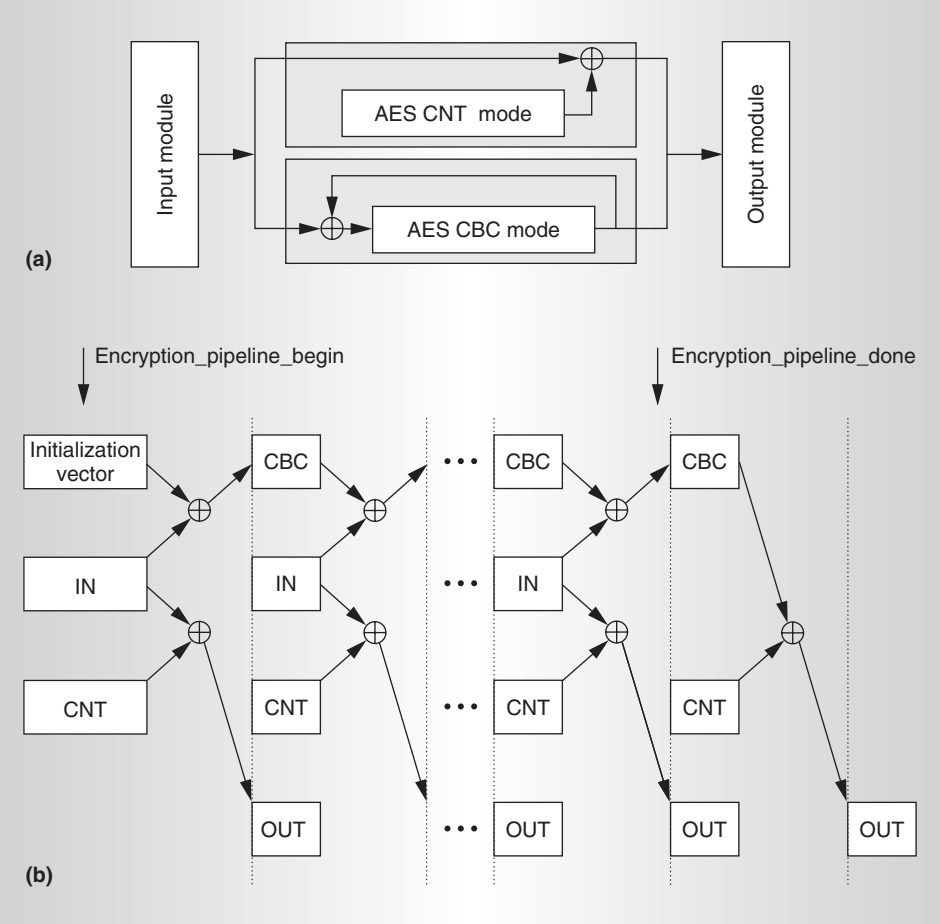


Figure 6. Encryption using AES in CCM (counter and CBC-MAC) mode: input block diagram (a) and block pipeline model (b).

Table 1. Cryptocoprocessor instruction set.

Mnemonic	Meaning	No. of clock cycles	Single or continuous
Software_reset	Reset the cryptocoprocessor	1	Single
Load_word_key	Read 32 bits of the key from the 32-bit input port connected to the main CPU core	1	Single
Load_word_data	Read 32 bits of data from the 32-bit input port connected to the main CPU core	1	Single
Write_word_out	Write 32 bits of output into the port connected to the main CPU	1	Single
Load_key_block	Read a 128-bit block of the key from the 32-bit input port asynchronously	10	Single
Load_data_block	Read a 128-bit block of data from the 32-bit input port asynchronously	10	Single
Write_block_out	Read a 128-bit block of the output result to the second 32-bit output port asynchronously	10	Single
Single_AES_encryption	Perform one encryption on the current data and key; takes 11 cycles	11	Single
Move_input_2reg	Move input to one of the data path registers	1	Single
Move_encrypt_2reg	Move encryption output to one of the data path registers	1	Single
Move_outreg_2out	Move output register to the output module	1	Single
Increase_counter_value	Increase the counter value for the CCM and counter modes based on the protocol specification	1	Single
Encryption_pipeline_begin	Start the continuous encryption pipeline	11/block	Continuous
Encryption_pipeline_done	Finish the continuous encryption pipeline	11	Single

Table 2. Configuration definitions.

Instruction	Configuration definition
Load_word_key	Which word of the key block (first, second, third, or fourth 32 bits of the key) to load
Load_word_data	Which word of the data block (first, second, third, or fourth 32 bits of the data) to load
Write_word_out	Which word of the output block (first, second, third, or fourth 32 bits of the output) to write
Single_AES_encryption	Whether the encryption is performed on input data, counter register, or CBC initial vector
Move_input_2reg	Where the input is moved: to the key register, counter register, CBC initial register, or data register
Move_encrypt_2reg	Where the encryption result is moved: to the output register, key register, or CBC initial register
Encryption_pipeline_begin	Whether the continuous encryption is in ECB, counter, CBC-MAC, or CCM mode

content of the internal registers of the encryption module's data path.

Most of the instructions use special settings through the configuration register. Table 2 lists some meaningful configurations for each instruction. For example, when Encryption_pipeline_begin executes, the configuration register specifies the operation mode to be performed—ECB, CBC-MAC, counter, or CCM.

Figure 7 shows an example program that performs the AES encryption in CCM mode on the continuous stream of input data.

We use the CCM mode from Housley's specifications.[8] First, we load the required key for the CBC-MAC and counter modes and move them to the key registers. We then load the CBC-MAC initial value and move it to the initial vector register, where it is encrypted and stored again in the IV register. To do this, we use the Single_AES_encryption and Move_encrypt_2reg instructions. We load

```
Software_reset
Load_key_block
Move_input_2reg (Move the input key to the key register)
Load_data_block
Move_input_2reg (Move the input data to the initial vector register)
Single_AES_encryption (Single AES encryption on CBC initial value)
Move_encrypt_2reg (Move encryption result into the initial vector register)
Load_data_block
Move_input_2reg (Move the input data to the counter register)
Load_data_block
Increase_counter_value
Encryption_pipeline_begin (Continuous encryption in CCM mode of operation)
Encryption_pipeline_done
```

Figure 7. Example program performing AES encryption in CCM mode.

Table 3. Synthesis results for the AES-based cryptocoprocessor.

Critical path	Clock frequency	Total area	Equivalent gate count	Throughput	Power estimate
3.38 nsec	295 MHz	0.732 mm^2	73,200	3.43 Gbps	86 mW

the initial counter value and place it in the counter register, where it is incremented. When initialization is complete, we start the block pipelining in CCM mode. On starting the pipeline, the cryptocoprocessor loads the data stream, encrypts it, and writes it to the output. For every 11 cycles, one block of 128-bit output is generated. The pipeline continues until Encryption_pipeline_done is set.

Performance results

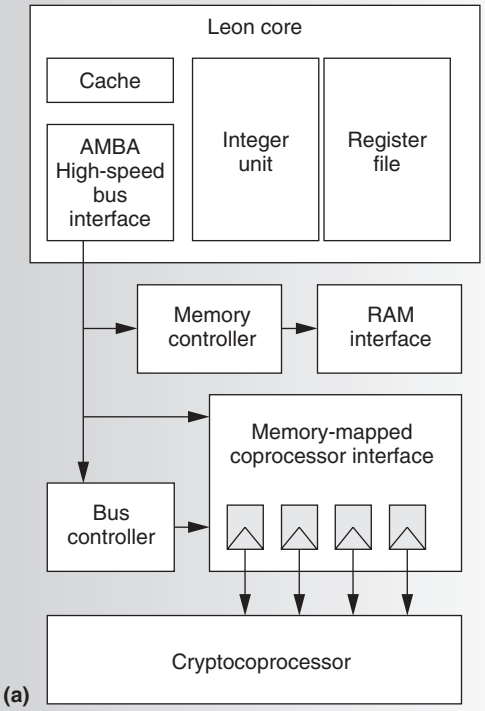
Table 3 presents synthesis results for our AES-based cryptocoprocessor. We performed synthesis using a typical United Micro-Electronic Corp. (UMC) 0.18-micron standard cell library with the Synopsys synthesis tools and the conservative wire load model. Because the core produces a 128-bit output every 11 cycles, we calculated throughput by multiplying the frequency by 128 and dividing the result by 11, giving us the number of bits produced per second. We achieved the maximum throughput of 3.43 Gbps at 295 MHz clock frequency with a power consumption of 86 mW at 1.8 V and 295 MHz.

System prototype

The complete system includes the Leon CPU core[13] and our cryptocoprocessor. We implemented a prototype system and tested the software routines on the Virtex-II FPGA board. As Figure 8a shows, the Leon CPU core controls the cryptocoprocessor through a memory-mapped interface. Figure 8b is a sample software routine for programming the cryptocoprocessor. The program performs one AES encryption in ECB mode. To fairly compare ours with earlier results,[14] we implemented the same setup, meaning the required data for the coprocessor is transferred through the Leon core. Thus, all data flows through the Leon core and does not use the separate streaming-data I/O modules.

The AES algorithm takes 24,419 cycles per 128-bit block (1,526.2 cycles per byte) using an efficient, high-speed software code on Xtensa; and it takes 1,400 cycles per 128-bit block (87.5 cycles per byte) to run AES using custom instructions on the customized Xtensa core.[14] Running the AES algorithm on our memory-mapped cryptocoprocessor using the program in Figure 8b takes 1,228 cycles. The coprocessor uses most of the 1,228 cycles for transferring the data and key from the Leon core and for the context switching necessary to call the program of Figure 8b from the main C program. The performance bottleneck is in the data transfers through the main CPU and the associated control overhead.

We also tested a program that uses block-pipelining continuous instructions for different operation modes. The difference between this program and the program in Figure 8b is




```
• Software_reset
• Move the key from memory to the registers
• Load_word_key (First 32-bit)
• Load_word_key (Second 32-bit)
• Load_word_key (Third 32-bit)
• Load_word_key (Fourth 32-bit)
• Move the data from memory to the registers
• Load_word_data (First 32-bit)
• Load_word_data (Second 32-bit)
• Load_word_data (Third 32-bit)
• Load_word_data (Fourth 32-bit)
• Single_AES_encryption (ECB mode)
• Write_word_out (First 32-bit)
• Write_word_out (Second 32-bit)
• Write_word_out (Third 32-bit)
• Write_word_out (Fourth 32-bit)
• Move the data from registers to the memory
```

Each of these instructions is equivalent to a group of assembly instructions that can set a value for a register, store a register to a memory address, or load a memory value to a register. For example:

```
"set 0x80000058, %o2"
"set 0x00000001, %o3"
"st %o3,[%o2]"
"ld [%l0],%o3"
```

(b)

Figure 8. Cryptocoprocessor connected to Leon CPU core: memory-mapped interface (a); sample software routine for programming the cryptocoprocessor (b).

that we replace Single_AES_encryption with Encryption_pipeline_begin and Encryption_pipeline_done. We also used the input and output interfacing modules to transfer the actual data to the cryptocoprocessor, as Figure 3b shows. The coprocessor continuously encrypts the input data in all the supported modes in only 11 clock cycles per block of input data.

Most future embedded systems will require high throughput and programmable security engines similar to the cryptocoprocessor presented in this article. Therefore, a system-level design methodology providing a hardware and software codesign environment for constructing similar embedded security engines is of interest for future work. Achieving such a methodology requires research on highly efficient interfaces for programming encryption accelerators through an embedded CPU core as well as high-throughput data transmission schemes between the hardware accelerators of a typical embedded system on a chip. MICRO

Acknowledgments

This article is based on work supported by the Space and Naval Warfare Systems Center, San Diego, under contract N66001-02-1-8938.

References

1. *Advanced Encryption Standard*, Federal Information Processing Standards (FIPS), publication 197, Computer Security Resource Center, Nat'l Inst. for Standards and Technology (NIST), Nov. 2001; http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf.
2. S. Frankel, R. Glenn, and S. Kelly, "The AES-CBC Cipher Algorithm and Its Use with IPsec," RFC 3602, Internet Soc., Sept. 2003; http://rfc.sunsite.dk/rfc/rfc3602.html.
3. A. Menezes, P. Oorschot, and S. Vanstine, *Handbook of Applied Cryptography*, CRC Press, 1996.
4. M. Dworkin, "Recommendation for Block Cipher Modes of Operation," NIST special publication 800-38a, Dec. 2001.
5. M. Dworkin, "Recommendation for Block Cipher Modes of Operation: The CCM Mode for Authentication and Confidentiality," NIST

special publication 800-38c, Sept. 2003.
6. R. Housley, "Using Advanced Encryption Standard (AES) Counter Mode with IPsec Encapsulating Security Payload (ESP)," RFC 3686, Internet Soc., Jan. 2004; http://www.faqs.org/rfcs/rfc3686.html.
7. S. Frankel and H. Herbert, "The AES-XCBC-MAC-96 Algorithm and Its Use with IPsec," RFC 3566, Internet Soc., Sept. 2003; http://www.faqs.org/rfcs/rfc3566.html.
8. R. Housley, "Using AES CCM Mode with IPsec ESP," Internet draft, work in progress, Internet Eng. Task Force, Nov. 2003; http://www.ietf.org/internet-drafts/draft-ietf-ipsec-ciph-aes-ccm-05.txt.
9. A. Satoh et al., "A Compact Rijndael Hardware Architecture with S-Box Optimization," *Proc. AsiaCrypt 2001*, LNCS 2248, Springer-Varlag, 2001, pp. 239-254.
10. J. Wolkerstorfer, E. Oswald, and M. Lamberger, "An ASIC Implementation of the AES S-boxes," *Proc. Cryptographer's Track at the RSA Conf. 2002*, LNCS 2271, Springer-Verlag, 2002.
11. I. Verbauwhede, P. Schaumont, and H. Kuo, "Design and Performance Testing of a 2.29-Gbps Rijndael Processor," *IEEE J. Solid-State Circuits*, vol. 38, no. 3, Mar. 2003, pp. 569-572.
12. D. Harel, "Statecharts: A Visual Formalism for Complex Systems," *Science of Computer Programming,* vol. 8, no. 3, June 1987, pp. 231-274.
13. J. Gaisler, "The Leon2 IEEE-1754 (SPARC V8) Processor," http://www.gaisler.com.
14. S. Ravi et al., "System Design Methodologies for a Wireless Security Processing Platform," *Proc. 39th Design Automation Conf.* (DAC 02), ACM Press, 2002, pp. 777-782.

Alireza Hodjat is pursuing a PhD in the Electrical Engineering Department at the University of California, Los Angeles. His research interests include hardware architectures and VLSI implementation for embedded and domain-specific processors and coprocessors. Hodjat has an MS in embedded computing systems from UCLA. He is a student member of the IEEE.

Ingrid Verbauwhede is an associate professor in the Electrical Engineering Department at UCLA. Her research interests include architecture design, VLSI implementation, and design methods for low-power embedded systems. Verbauwhede has a PhD from KU Leuven, Belgium. She is a senior member of the IEEE.

Direct questions and comments about this article to Alireza Hodjat, UCLA, Electrical Eng. Dept., Room 53-109, E-IV Building, 420 Westwood Plaza, Los Angeles, CA 90095-1594; ahodjat@ee.ucla.edu.

Coming Next Issue

July–August 2004
Embedded Systems

Guest Editors
Alessio Bechini, University of Pisa, Italy
Antonio Prete, University of Pisa, Italy
Tom Conte, North Carolina State University

xDSPcore— A Compiler Based Configurable Digital Signal Processor
Andreas Krall et al.

Integrating Cache Coherence Protocols for Heterogeneous Multiprocessor Systems
Taeweon Suh, Hsien-Hsin S. Lee, and Douglas M. Blough—Georgia Institute of Technology

Programming Models for Hybrid FPGA/CPU Computational Components: A Missing Link
David Andrews et al.—University of Kansas

Design Space Exploration for Real-Time Embedded Stream Processors
Sridhar Rajagopal, Joseph R. Cavallary, and Scott Rixner—Rice University

Efficient Real-Time, Fine-Grain Concurrency on Low-Cost Microcontrollers
Alexander G. Dean—North Carolina State University

QoS for High-Performance SMT Processors for Embedded Systems
Francisco J. Cazorla et al.