Interfacing a High Speed Crypto Accelerator to an Embedded CPU

Alireza Hodjat
ahodjat @ee.ucla.edu
Electrical Engineering Department
University of California, Los Angeles

Ingrid Verbauwhede
ingrid @ee.ucla.edu
Electrical Engineering Department
UCLA and K.U.Leuven

Abstract- **Crypto co-processors are needed for acceleration of encryption functions. But critical to the performance gain is the selection of an adequate interface. This paper presents the AES acceleration for two interface options to the LEON CPU core: the CPI interface and the memory-mapped interface. The complete system including the LEON core and the loosely coupled AES accelerators are implemented on an FPGA and the software programs that control the AES accelerators are tested. The cycle count, the throughput, the LUT usage, and the energy cost of running a complete AES program using the above accelerators are compared with a pure software implementation, and with a tightly coupled instruction set extension option.**

I. INTRODUCTION

Hardware coprocessors for cryptographic algorithms are required for different applications. Choosing the right interface for the hardware accelerator is always a challenge to satisfy the required throughput of the security applications. This paper presents a complete system that includes the LEON CPU core [1] and a hardware AES [2] accelerator. The AES accelerator is attached to the LEON core using two different interfaces. One of them is memory-mapped interface and the other one is the CPI interface (Co-Processor Interface), which is a LEON specific interface. The cycle count, throughput, the area cost, and the energy consumption of running the AES algorithm using these interfaces are computed and compared with a software implementation of AES on the LEON core. The FPGA prototype is implemented and the software routines are tested on the actual board. The complete design is mapped onto the Virtex-II (XC2V1000) FPGA [3] and the Virtex-II V2MB1000 board from Insight [4] is used for this experiment.

Our approach is based on a loosely coupled, independently working coprocessor, which is a programmable coprocessor that is designed for a specific domain and is attached to the main processor on a dedicated interface [5]. An alternative approach, based on a tightly coupled, instruction set extension approach is presented by Ravi in [6]. It presents a system level design methodology for programmable security processor platforms. It uses the Xtensa processor from Tensilica [7] and includes customized instructions. This way, the performance is improved from less than one Mbits/s to several 10's of Mbits/s. The main CPU (Xtensa) is customized for a specific domain by adding a new functional unit to its pipeline. Then custom instructions flow through the pipeline and are decoded and executed on the new functional unit.

A typical embedded system contains multiple domains. Examples are the networking domain for network protocol processing, DSP domain for image or speech processing, and the security domain to provide authentication and privacy to the application. Figure 1-a shows a typical example where the stream of data samples flow from the image-processing unit (DSP) to the security unit and continue to the networking unit. Figure 1-b shows the solution for mapping this dataflow on a processor with a tightly coupled instruction set extension. The processor is customized for each domain by adding new functional units (Dsp, Sec, Net) to its pipeline. This way, the domain instructions are decoded and executed in the corresponding functional units. On the other hand, figure 1-c shows the mapping of this dataflow on our architecture with loosely coupled coprocessors. The programmable co-processors that meet the throughput requirements are designed for each domain. Then it is the job of the main embedded processor to program each domain specific coprocessor. Here, control is done through the embedded processor while the data is transferred between the coprocessors themselves. In comparison, the tightly coupled approach is a top-down methodology because each domain is first specified by a C code and then it is decomposed to custom instructions and the software that calls the domain instructions. On the other hand, what we propose is a mixed bottom-up, top-down methodology. Each domain is first mapped on a programmable, high throughput coprocessor (bottom-up) and then the application is defined as the software on the embedded processor core that controls each domain specific coprocessor (top-down). In the top-down methodology, the control and data streams remain coupled. In the mixed methodology, a better performance is achieved since the data and control are not tightly coupled and each coprocessor is designed for the required performance.

0-7803-8622-1/04/$20.00 ©2004 IEEE

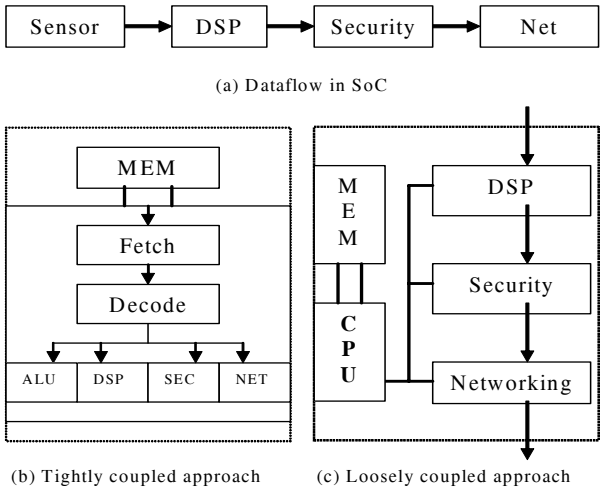

Figure 1. Acceleration options in a typical embedded SoC

This paper compares the performance trade-offs of using two interface options of the LEON CPU core, either the memory-mapped interface or the CPI interface. Moreover, the results will be compared to the tightly coupled processing approach as is reported in [6]. The AES algorithm is used as the driver application and the accelerators are designed for the LEON embedded CPU core.

II. ARCHITECTURE OF THE AES ACCELERATOR

Figure 2 shows the internal architecture of the AES core that is designed for hardware acceleration of the LEON CPU core. Different steps of the algorithm are byte substitution, shift row, mix column, and key addition [2]. The key scheduling datapath generates the round keys for each iteration of the algorithm. This AES core performs encryption on data and key size of 128-bits length. It uses a single round implementation of the AES algorithm and it takes total of 11 cycles to perform one encryption on the input data.

Figure 3 shows the architecture of the AES accelerator and its IO interfaces. The AES core of Figure 2 is used along with input and output interfacing modules [8]. There are four different IO ports: an 8-bit instruction port, an 8-bit configuration port, an input port, and an output port. The input and output ports are 32-bits and 64-bits long for the memory-mapped interface and the CPI interface, respectively. In order to program the accelerators, the main CPU (LEON) has to write data into or read from these ports. The interface to these ports is built by placing registers between the ports of the co-processor and the main CPU core.

III. COPROCESSOR INTERFACE OPTIONS

The two different interface options of the LEON CPU core are the memory-mapped interface and the CPI interface (Co Processor Interface) as explained next.

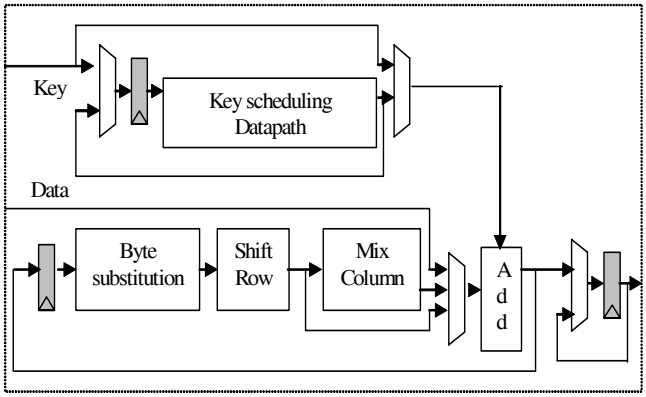

Figure 2. The Advanced Encryption Standard core

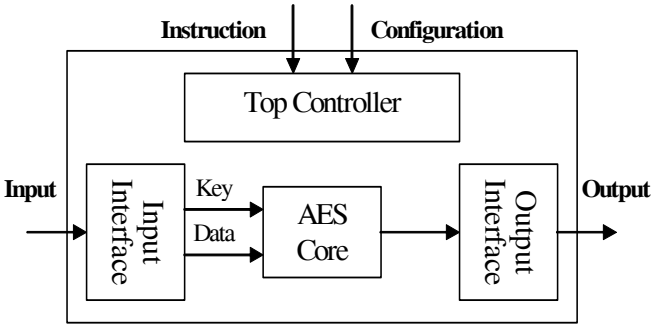

Figure 3. The architecture of the AES accelerator

A. *Memory-mapped Interface*

Figure 4 shows the architecture of the LEON CPU core with the AES coprocessor attached to its memory-mapped interface. The coprocessor can be programmed using a series of the LEON assembly instructions. Figure 5 shows a software routine that is used to program the AES coprocessor. This program performs one AES encryption in the ECB mode of operation. In the case of memory-mapped interface, each of the coprocessor instructions consists of a group of LEON data transfer assembly instructions. Using these instructions, the required instruction opcodes and configuration values are loaded into the AES coprocessor.

B. *CPI Interface*

Figure 6 shows the architecture of the CPI interface that is used to program the AES accelerator from the LEON core. The main difference between the CPI interface and the memory mapped interface is that the CPI interface is accessible directly from the integer unit of the LEON core while the memory mapped interface is accessible using the AMBA bus. Moreover, in the case of CPI interface the input and output data registers are 64 bits while they are 32 bits long in the case of memory-mapped interface.

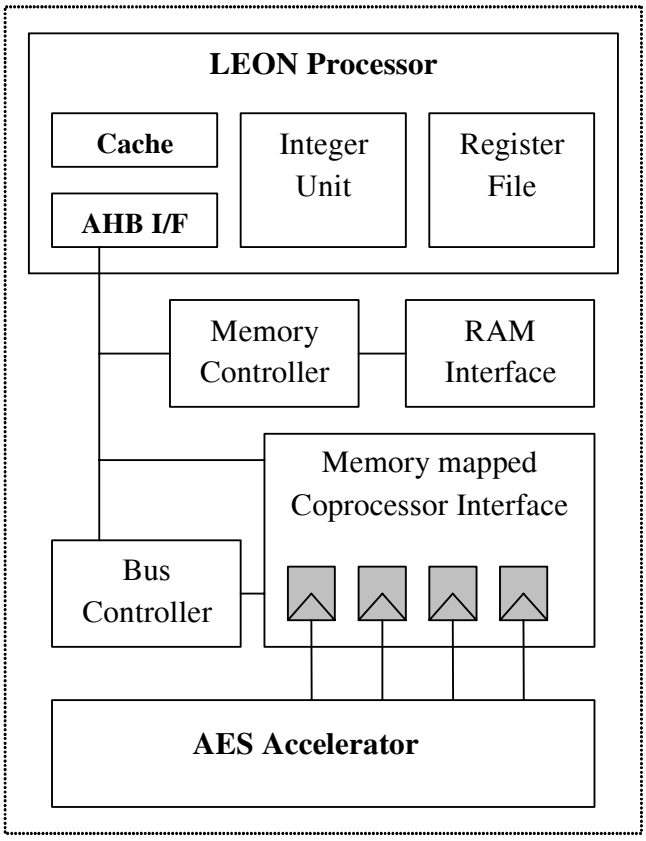

Figure 4. AES acceleration using the memory-mapped interface

- *Reset*
- *Move the key from memory to the registers*
- *Read_32bit_key (First 32-bit)*
- *Read_32bit_key (Second 32-bit)*
- *Read_32bit_key (Third 32-bit)*
- *Read_32bit_key (Fourth 32-bit)*
- *Move the data from memory to the registers*
- *Read_32bit_data (First 32-bit)*
- *Read_32bit_data (Second 32-bit)*
- *Read_32bit_data (Third 32-bit)*
- *Read_32bit_data (Fourth 32-bit)*
- *Encrypt_once (ECB mode)*
- *Write_32bit_out (First 32-bit)*
- *Write_32bit_out (Second 32-bit)*
- *Write_32bit_out (Third 32-bit)*
- *Write_32bit_out (Fourth 32-bit)*
- *Move the data from registers to the memory*

Hint: Each of the above instructions is equivalent to a group of assembly instructions, which can set a value for a register, store a register to a memory address, or load a memory value to a register.

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

Figure 5. Programming example for the memory-mapped interface

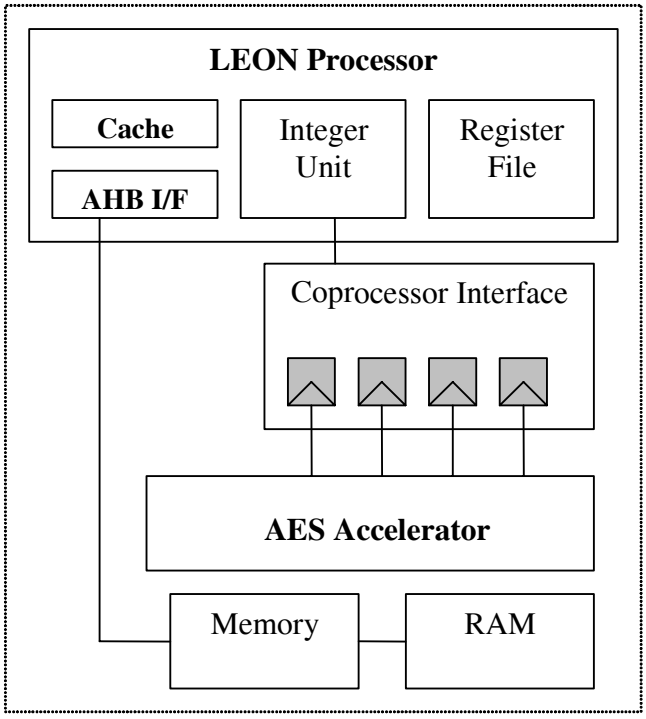

Figure 6. AES acceleration using the CPI interface

- *Reset*
- *Move the key from memory to the registers*
- *CPOP Instruction #1: Load the whole key to the coprocessor*
- *Move the data from memory to the registers*
- *CPOP Instruction #2: Load the whole data to the coprocessor*
- *CPOP Instruction #3: Encrypt_once (ECB mode)*
- *CPOP Instruction #4: Write first half of the output data to the registers*
- *CPOP Instruction #4: Write second half of the output data to the registers*
- *Move the data from registers to the memory*

Hint: The CPI interface allows moving two 64-bit data (128-bit) from the integer unit to the coprocessor with a single instruction. Also one 64-bit output result can be moved to the integer unit with a single instruction.

Figure 7. Programming example for the CPI interface

Figure 7 shows a software routine that is used to program the AES accelerator through the CPI interface. This program is similar to the program of figure 5 and performs one AES encryption in the ECB mode. The instructions that are shown in this figure are the AES accelerator instructions. The CPI instructions of the LEON core are transferred to the required instructions and configurations of the accelerator using the CPI interface.

Notice that the programs of figure 5 and 7 include the instructions to load the data and key from the LEON core to the accelerator and return the encrypted results back to the LEON. This shows that the proposed memory-mapped and CPI interfaced coprocessors can be considered as an acceleration option that lies between tightly and loosely coupled approaches. In a tightly coupled coprocessor (Figure 1-b), the interaction is on an instruction by instruction base, i.e. every clock cycle. However, the CPI and memory mapped interfaces are more coarse grain. The communication with the CPU core is only for the data load/store and crypto instructions. Here, there is still a large performance overhead due to the data communications that is caused by the read and write instructions as shown in Figures 5 and 7. In the loosely coupled approach (Figure 1-c) the data and control is really split. Therefore, better performance is achieved for high throughput encryption applications [8].

IV. PERFORMANCE RESULTS

Table 1 shows the result of the two different interfaces that are presented in the last section. An efficient C code of the AES algorithm is compared with the design of Figures 4 and 6. In this table the complexity of running a complete AES algorithm is measured in terms of VHDL line count for the accelerator and its interface, C program line count of the software routine, the program size in ROM, and the FPGA LUT usage. On the other hand, the performance is compared in terms of the cycle count, the execution time, and the estimated energy consumption. In summery, Table 1 shows that the AES accelerator using the CPI interface is 1.7 times faster and 35 % more energy efficient than the AES accelerator using the memory-mapped interface while it consumes 1.1 times more LUTs of the FPGA.

Figure 8 shows the cycle count comparison of performing one AES encryption in ECB mode. According to [6], the AES algorithm takes 24419 cycles per 128-bit block (1526.2 cycles per byte) using efficient, high-speed software codes on Xtensa and it takes 1400 cycles per 128-bit block (87.5 cycles per byte) to run AES using AES custom instructions on the customized Xtensa core. In our case, it takes 45254 cycles to run an efficient software implementation of the AES algorithm on the LEON core. It takes 1228 cycles to run the AES algorithm on the memory-mapped crypto coprocessor using the program shown in figure 5 and it takes 704 cycles to run it on the CPI-interfaced crypto coprocessor using the program

TABLE 1
Performance of running AES using the memory-mapped and CPI interfaced accelerators compared to the pure software implementation

AES in ECB mode	C code on LEON	Memory-mapped coprocessor	Coprocessor using CPI interface
VHDL code (lines)	-	2260	3030
C program code (lines)	251	156	107
Performance (# of cycles)	45254	1228	704
Execution time (1 AES at 50 MHz)	905 (µsec)	24.5 (µsec)	14.1 (µsec)
Program size in ROM	36.3 Kbytes	8.6 Kbytes	8.2 Kbytes
FPGA usage (LUTs)	47.4 % (4856)	81.3 % (8330)	95.2 % (9756)
Estimated Power for FPGA	558 mWatts	645 mWatts	734 mWatts
Estimated Energy (one AES in ECB)	505 µJoules	15.8 µJoules	10.3 µJoules

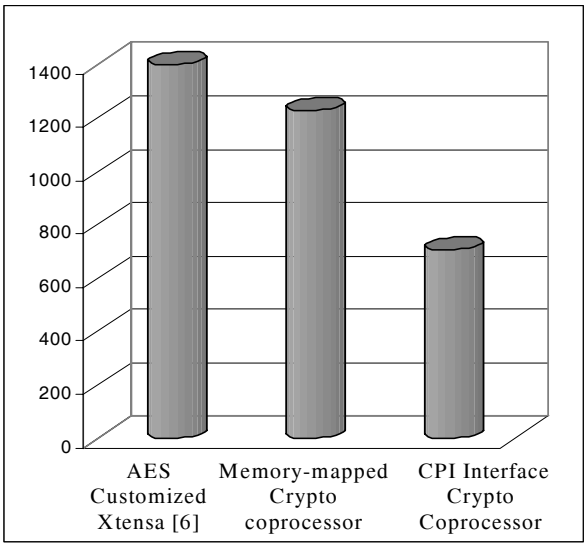

Figure 8. Cycle count comparison for the AES in ECB mode

shown in figure 7. This means that the proposed memory-mapped AES accelerator and CPI interfaced AES accelerator are 1.2 and 2 times faster than the AES customized Xtensa processor of [6], respectively.

Table 2 compares the throughput of our AES-based coprocessor example with Tensilica's platforms based on the experiment in [6]. The throughput is calculated for the case of FPGA implementation (50 MHz clock) and the ASIC implementation (188 MHz clock) based on the cycle count presented in figure 8. In the case of ASIC the clock frequency of 188 MHz is used because according to [6], the Xtensa

processor core runs at 188 MHz. On the other hand, in the FPGA case the 50 MHz clock frequency is the actual frequency of the board on the FPGA prototype implementation. Notice that the numbers that are reported in Table 2 include the time that is needed to load the input and store the output to memory, which takes several hundred cycles per encryption. As an example, Table 2 shows that the total cycle count for one AES encryption using the CPI interface is 704 cycles. The AES encryption itself takes only 11 cycles, but the complete program that includes loading the data and key, AES encryption, and returning the result back to the software routine takes total of 704 cycles. The reason is that all data traffic is still through the LEON core. However the main performance gain compared to the tightly coupled approach is because the actual encryption instructions are coarse grain.

V. CONCLUSION

The AES acceleration for the two interface options of the LEON CPU core, the CPI interface and the memory-mapped interface, are explored. The cycle count, throughput, the area cost, and the energy consumption of running the AES algorithm using these interfaces are computed and compared with a pure software implementation, and with a tightly coupled instruction set extension option. The result shows that the AES accelerator using the CPI interface is 1.7 times faster and 35 % more energy efficient than the AES accelerator using the memory-mapped interface. Moreover, the memory-mapped AES accelerator and CPI interfaced AES accelerator are 1.2 and 2 times faster than an AES customized processor, respectively.

TABLE 2
Cycle count and throughput comparison with approach [6]

Platform	Cycle Count	Throughput for FPGA (At 50 MHz)	Throughput for ASIC (At 188 MHz)
Embedded Xtensa	24419	0.26 Mbits/s	0.98 Mbits/s
Customized Xtensa for AES	1400	4.57 Mbits/s	17.2 Mbits/s
LEON core	45254	0.14 Mbits/s	0.53 Mbits/s
LOEN core with memory-mapped cryptocoprocessor	1228	5.21 Mbits/s	19.6 Mbits/s
LEON core with CPI interfaced cryptocoprocessor	704	9.1 Mbits/s	34.2 Mbits/s

ACKNOWLEDGMENT

This material is based upon work supported by the Space and Naval Warfare Systems Center - San Diego under contract No.N66001-02-1-8938. The authors would like to acknowledge the funding of this project.

REFERENCES

[1] LEON SPARC V8 CPU Core, Gaisler Research, http://www.gaisler.com/

[2] Federal Information Processing Standards (FIPS), Publication 197, "Advanced Encryption Standard" November 2001, available at: http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf .

[3] http://www.xilinx.com/xlnx/xil_prodcat_landingpage.jsp?title=Platform+FPGAs

[4] http://www.insight-electronics.com/Memec/iplanet/link1/VIRTEX11MBV1000.PDF

[5] F. Barat, R. Lauwereins, G. Deconinck, "Reconfigurable Instruction Set Processors from a HW/SW Prospective", IEEE Transactions on Software Engineering, Vol. 28, No. 9, September 2002.

[6] S. Ravi, A. Raghunathan, Potlapally, Sankaradass, "System design methodologies for a wireless security processing platform", DAC 2002, New Orleans, USA.

[7] *Xtensa application specific microprocessor solutions – Overview handbook*. Tensilica Inc., 2001.

[8] Alireza Hodjat, Ingrid Verbauwhede, "High-Throughput Programmable Cryptocoprocessor", IEEE Micro Magazine, Volume: 24 , Issue: 3, Pages: 34 - 45, May/June 2004.