ASEP: Application-Specific Encryption Processor

Benjamin A. Collins, Yixuan Li, Rabi N. Mahapatra

Abstract— **Embedded systems are increasingly dependent on secure information processing, particularly where mobile and fixed wireless networks are concerned. Modern embedded systems generally do not have the resources necessary to carry out the amount of processing required for adequate security, and it is a gap that will widen if current trends are left unchecked. We will focus on one core aspect of such processing — encryption. In this paper, we propose a new encryption processor architecture that provides a high level of flexibility as well as additional computational abilities to an embedded system.**

Index Terms— **Embedded, Co-processor, Concurrent Processing, Encryption Algorithm**

I. Introduction

SECURITY in embedded computing environments is growing in importance as many new types of devices and applications emerge, particularly where wireless communication is concerned. However, there are several significant challenges to meeting the security needs of the present and — more so — the future. There is already an identifiable gap between the trends of the requirements for adequate security and the capabilities of embedded systems. Current computational requirements for implementing security protocols (and the cryptographic algorithms that they employ) already outstrip the capabilities of even today's high-end embedded computers, and the gap will widen at an exponential rate for the foreseeable future [1]. Our focus is on cryptography in embedded systems.

A. Performance Gap

Current protocols use a number of cryptographic algorithms, including symmetric ciphers, one way hash functions, and asymmetric ciphers. An asymmetric cipher — public key encryption — depends on some very difficult computational problem (like finding the prime factors of extremely large numbers as in the RSA algorithm) to ensure that only the intended recipients can decode encrypted data, thus decoding is expensive even when the appropriate information is known. Symmetric ciphers are based on the involved parties having private information that allows data to be decoded. The difficulty with symmetric ciphers is getting the private information to the parties without said privacy being compromised — the key distribution problem. However, once the private key is distributed, the necessary computation is manageable. In many systems, asymmetric ciphers (such as RSA, DSA, Diffie-Hellman, and schemes based on elliptic curves) are used to encrypt keys for use in symmetric ciphers such as AES, DES, IDEA, CAST, etc. Hash functions can be used to verify data integrity. However, even when using symmetric cipher algorithms to encrypt data, it is still quite difficult (or expensive) to implement an embedded system based on a general-purpose processing unit that can process encrypted data at commonly used rates. For example, a copper-based LAN can operate at rates from 10 Mbps to 10 Gbps (and will soon be able to operate at 40 Gbps!) and wireless LANs operate in the 2–54 Mbps range. The latest embedded processors would only barely be able to keep pace at about 4 Mbps with full utilization [1]. This is not useful if the system processes encrypted video [2], high-bandwidth multimedia, or an otherwise rate-demanding application.

There is also a significant difference in the energy capacity of modern portable devices and the power requirements of security processing. In addition, the growth of battery capabilities is low (5-8%) while the growth of security processing requirements steadily increases [1](However some significant improvements are being made in this area, particularly with Lithium Polymer energy sources [3], [4]). You can see in figure 2 that there is a "battery gap" that is the difference between power requirements for security processing and the energy available from battery sources. In one example from [1], the energy cost of transmitting an encrypted message was about 200% more than the cost of transmitting an unprotected signal in a wireless sensor network.

One solution to these problems is to use more and more powerful CPU and let software to handle the encryption. However, this solution is very costly. Another solution that current industry is leaning toward is to have a special co-processor and let it handle the encryption work. This is a potentially cost effective solution and there have been several approaches. We will discuss these approaches in the next section.

B. Contributions

The main contribution of this work is to describe a detailed architecture for an encryption co-processor that would enable an embedded system to perform tasks that would normally be unmanageable for an unaided embedded processing system. This architecture also provides a high level of flexibility to the embedded system, allowing service availability and reliability requirements to be better met in a cryptographic environment.

II. Previous Work

There are several existing architectures for cryptographic co-processors targeted toward embedded systems. Some general

architectures are presented in [5], [6], [7], [8]. Chodowiec [5] and his group show that advanced architectural techniques can be used to improve performance for block-ciphers; they implement pipelines and loop-unrolling in a FPGA-based architecture. Similar approaches are taken in [6], [7] with regards to the International Data Encryption Algorithm (IDEA). They compare the differences in performance of serial and parallel implementations of IDEA using a Xilinx Virtex platform; power consumption is not considered. In [8], the authors explore a methodology for hardware-software partitioning between ASICs, DSPs, and FPGAs for optimizing power and performance for customized encryption units. Author of [8] also points out that since there are limited resources available to mobile communication devices, proper balance of performance, flexibility, and power usage is important. He presents design choices associated with these factors in his FPGA-based implementation of IDEA.

In [9], [10], [11], [12], reconfigurable architectures are considered. Goodman describes a CMOS based domain-specific reconfigurable cryptographic processor (DSRCP). A flexible FPGA-based architecture is described in [11]. Their work exploits reconfigurability in order to attain an high-performance implementation of cryptographic hardware that supports various algorithms. In a related project [12], an architecture called *Cryptographic (Optimized for Block Ciphers) Reconfigurable Architecture* (COBRA) is introduced. In contrast to [11], COBRA is more generally designed to be efficient for a number of block ciphers, with only minimal reconfiguration needed for significant flexibility. COBRA is focused on performance and throughput. Piperench [10], [13] is a project at Carnegie Mellon University based on the idea of virtualizing the hardware so that any particular implementation is not limited to the number of gates on the reconfigurable fabric.

There have also been a number of implementations of the recently standardized AES algorithm, Rijndael [14], [15] as well as the former NIST[1] standard DES[16] and others.

All designs mentioned above all have one common problem: They are only implemented to have one algorithm or one session available at any given time. If an application requires multiple algorithms or needs to have multiple sessions open at the same time, a second device is needed. The goal of our project is to design an architecture that is able to handle multiple sessions and algorithms concurrently while not sacrificing much performance. This approach requires the embedded system to have more resources — including the cryptographic co-processor described here, potentially more energy consumption, and a larger physical size.

A. Cryptographic Algorithms Basics

1) Data Encryption Standards (DES): The Data Encryption Standard (DES) [17] algorithm is a very popular secret key encryption algorithm that is widely used; it was standardized by the National Bureau of Standards in the Federal Information Processing Standards #46 and reaffirmed in #46-1 and #46-2 [17].

DES is a block cipher which processes 64-bit plaintext blocks and produces 64-bit ciphertext blocks. The encryption process is the following. Let K be a 48-bit block from the key, chosen by a key scheduler. The 64-bit input is split into to 32-bit inputs, L and R. During one iteration with input LR, the output is $L'R'$ where $L' = R$ and $R' = L(+)f(R,K)$ where (+) denotes bit-by-bit addition modulo 2 [17]. Function $f(R,K)$ takes the 32-bit R bitwise adds it to K. It then passes the 48-bit result through 8 fixed 6-to-4 bit mappings, also known as S-boxes, that produce a 32-bit vector that is finally permuted before output. This single iteration is called a 'round', and each round takes the output of the previous round as input. DES specifies 16 rounds of this type, followed by an end permutation that is the inverse of the initial permutation.

Because each stage only requires the output of the previous stage as its input, pipelining can be used to parallelize the implementation of the algorithm and thus achieve a performance gain. The two most heavily used operations in DES are the binary *XOR* operation and table lookups. The performance realized by any design depends heavily on the implementation of these two operations.

2) Triple-DES Algorithm (3DES): With rapidly increasing processing amount of processing power available, DES is no longer able to meet today's security requirements. Plaintext encrypted using a 56-bit key can be found within 24 hours using commercially available computing resources [18]. One possible solution is simply to increase the size of the key used in order to make searching the entire keyspace an intractable problem given available computing resources. In order to preserve compatibility with the previous standard, triple-DES was introduced [16]. The algorithm chains 3 DES cores together and uses three keys to produce an effective key length that is three times the length of a single DES encryption. Since each of the three cores are identical to single DES, 3DES can either be pipelined using three cascaded cores or use a single core 3 times. These choices present a trade-off involving power and performance.

3) International Data Encryption Algorithm (IDEA): The International Data Encryption Algorithm is another popular secret-key encryption algorithm. IDEA is a block cipher similar to DES in structure. It takes a 64-bit plaintext input and produces a 64-bit ciphertext using a 128-bit key. IDEA uses operation from different algebraic groups including *XOR*, addition modulo 2^{16}, and multiplication modulo the Fermat prime $2^{16}+1$. IDEA can operate under different modes: Electronic Code Book (ECB) mode and Cipher-Block Chaining (CBC) mode [19]. In ECB mode, the algorithm doesn't have feedback paths, and could be more vulnerable to differential cryptanalysis. In CBC mode, every plaintext block is *XOR*-ed with the previous ciphertext block before being encrypted.

[1]National Institute of Standards and Technology.

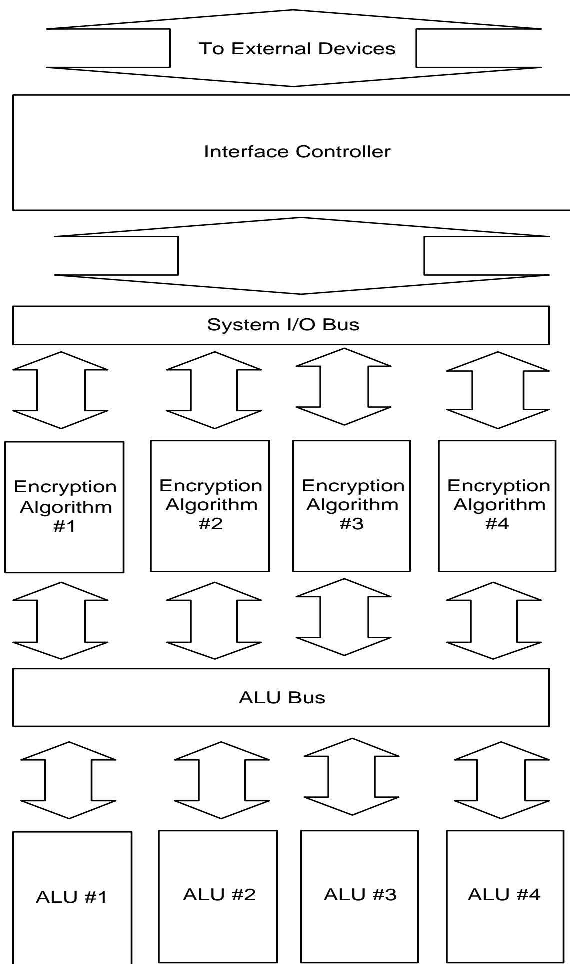

Fig. 1. ASEP System Overview.

Unlike DES, IDEA has more complicated calculations that can sometimes lead to bottlenecks; modulo multiplication is often the culprit [7]. In each round, there are four modular multiplication operations, so the performance of the algorithm depends heavily on how fast the implementation can produce the modulo multiplications.

III. ASEP System Definition

Balancing performance and flexibility is important for practical security mechanisms in resource-limited embedded systems [8]. Our work is an approach that targets on producing a architecture that produces relative good performance, yet have the flexibility to be able to handle different algorithms and sessions.

A. Basic Architecture Model

The ASEP architecture is broken down into three major parts according to functionality: an interface unit, an algorithmic unit, and an arithmetic unit. A schematic for this decomposition can be found in 1The interface unit is in charge of the I/O functions servicing external devices, such as a processor. It includes a microcontroller that handle session request and internal data routing. Currently, the interface is does not conform to any known interface standard. Future work to be done in this area might include implementing an open standard I/O interface for this type of device.

The algorithm unit contains a group of algorithm modules that direct control and data-flow within the device to execute various encryption algorithms. The arithmetic unit is a group of specialized arithmetic logic units (ALUs) that perform special functions that are frequently needed such as exclusive or (XOR), modular multiply, modular exponentiate, and greatest common divisor (GCD).

One of the major design features of ASEP is to support algorithm concurrency — that is, the ability to service multiple requests. Unlike previous designs, ASEP is designed to be able to handle multiple algorithms at the same time without user intervention. In other words, different applications can share ASEP without requiring any external management. This is done by requiring any application that intends to use ASEP to establish a "encryption session". Each encryption session is uniquely identified by a tag.

Once an encryption session is established, interface unit will receive key and data stream through external bus and route them to proper algorithm module. To save hardware space, algorithm unit will only contain algorithm specific operation such as table look up and permutation. Operations that is not algorithm specific, such as modular multiply and exponential, will be delegated to the arithmetic units. This way, we can reduce amount of hardware that is consumed by this operations. When an algorithm needs to have an arithmetic operation done, it will send a request along with data through the ALU bus to the specific ALU to have that operation completed. The detailed structure design and data flow will be discussed in section V.

B. Instruction Format

ASEP uses a set of instruction to communicate with external devices. An ASEP instruction is 44 bits in length. It consists of an 8-bit op-code field, 4-bit tag field, and 32-bit data field. Figure 2 illustrates this format.

The 8-bit op-code consists of a 4-bit algorithm identifier, a 3-bit action code, and a 1-bit flag to indicate a request for an encoding or an decoding session; the format is shown in Figure 3. Currently, the action code includes CLEAR, FLUSH, and READ. The functionality details of these action code will be discussed in Section V. The 4-bit tag field allow 16 concurrent sessions to be established at any given time.

While designing ASEP architecture, we decided to establish some user conventions in order to ease the design problem. We assume that the system processor shall only request another

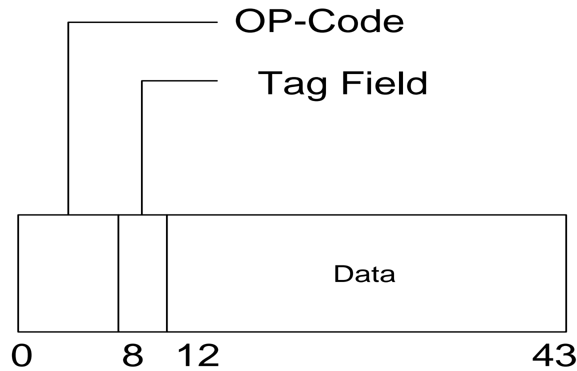

Fig. 2. Instruction Format.

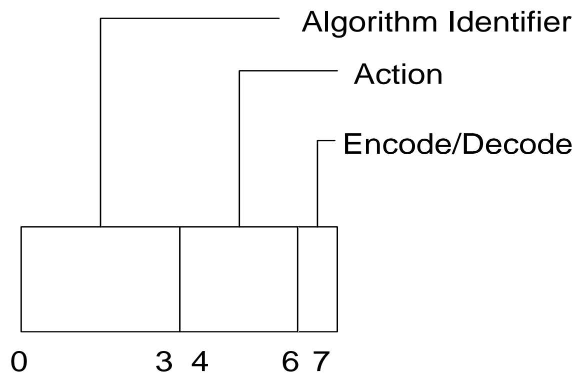

Fig. 3. Op-Code Format.

block to be processed when it has received output from the previous requested block for any particular session.

C. Algorithm Concurrency

Algorithm concurrency is the primary goal of the ASEP design. Concurrency is achieved by using sessions and parallelizing arithmetic operations.

1) Sessions: In order to allow different applications to use ASEP concurrently, each session needs to be uniquely identified because each session may use a different algorithm and have different key. We introduced session tags to solve this problem. All data that flows in and out and even within ASEP has a session tag associated with it. A session is established by an external device that sends a session request, and closed by a session destroy request.

By using sessions, data blocks can be interleaved by external device if necessary. This can be viewed as sending data to ASEP in a special form of "packet". Since each block of data contains a session tag, no exclusive lock of the ASEP for any device is necessary, because the system will be able to track sessions within each algorithm module.

2) Parallel Processing: Each encryption module will be able to handle as many session as the tag allows. A typical block cipher algorithm consists of sequences of table look-up and ALU operations. Almost all ALU operations will be sent off and processed by special ALUs. While the encryption module is waiting for the ALU results, it can start to process other sessions that is handled by this algorithm module. This would utilize ASEP modules more efficiently and give the effect of pipelining.

D. Hardware / Software Partitioning

Partitioning for the ASEP system is relatively easy due to the distinct characteristics of each component of the system. Our initial partitioning scheme can be seen at an abstract level in Figure 1.

The I/O interface is not very well defined and could vary from system to system; flexibility is a very significant factor. Also, this part of the ASEP system is not critical to the performance of ASEP. We decide to partition this part of the system to software.

Algorithm modules and specialized ALUs are performance critical units. Also flexibility is not an issue for these parts of the ASEP system since all algorithm and ALU operations are very well defined. We would partition this part of ASEP to hardware.

E. Implementation

Work on the actual implementation of ASEP has begun, though there are still features and modules left to design. The project is implemented using the SystemC™language extensions to the STD C++ language. This facilitates a shorter development time by allowing for design at a high level — as opposed to HDL descriptions — while avoiding traditional pitfalls of manual translation into a synthesize-able form during later development stages.

We are able to construct a transaction level model of ASEP system and run simulations through it. This transaction level model is accurate to clock level. Sub-clock events are not consider in this simulation. This model serves as a proof-of-concept of a complete working ASEP system.

IV. Functional Unit Design

The design was driven in part as key parts were implemented. The algorithm cores were implemented and the details of the architecture were determined first. Other parts of the design followed from these.

A. Cryptographic Cores

These modules are the core part of this design. Most multi-session processing is assigned to these modules; they also

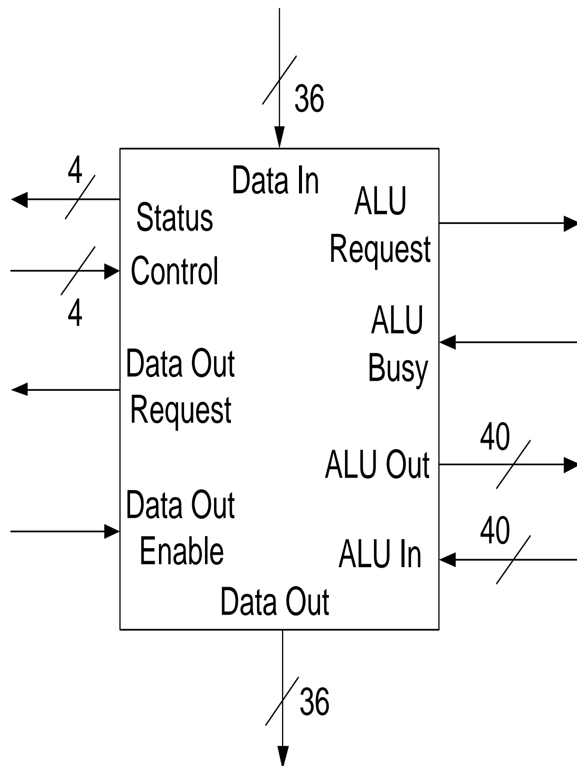

Fig. 4. An ASEP Algorithm Module.

handle all algorithmic operations. Two modules have been implemented; DES and IDEA. See cryptographic section II-A for details on the algorithms themselves. The DES module is relatively straight forward.

Multi-session processing is one of the priorities of this design. In the two cores that we have implemented, each tag uses its own set of buffers and status codes so that the algorithm core can keep the data for each session separated. The microcontroller — which serves the purpose of the interface controller in Figure 1 — sends one of the following signals as data is acquired: CLEAR, READ, or FLUSH. If "flush" is sent, the core will erase all data in all buffers for all tags. If the signal is "clear", the core will erase only data for the specified tag. If "read" is sent, then the core will accept data for either encryption or decryption. Depending on the algorithm, the first block or first several blocks of data acquired are stored as the key for the session and key schedules — if necessary — are calculated from it. For DES, the necessary blocks are composed of 56 bits of key data along with 7 parity bits. Since data is acquired by the cores from a 32 bit bus, two blocks are necessary to obtain the key. Once the key is stored in the buffer associated with the current session, all other blocks received are processed as data to be either encrypted or decrypted. As incoming data arrives it is stored in a buffer specific to the session it is associated with. In SystemC, this buffer can be represented in a simple way by a two-dimensional array; for example:

```
int buffer[16];
```

would allow each of 16 sessions to store `sizeof(int)` bytes, 32 bits on most modern architectures. Once the data for a single block is received, an action is performed on it; either encrypt or decrypt. The key schedule is calculated when the first block of data arrives and is subsequently processed.

During this stage, many arithmetic operations are needed. The XOR operation is used heavily, as are modular operations such as modular multiply, and modular add. Table lookups are very common and are implemented on the core itself. When the algorithmic module needs to make use of one of the ALUs, it writes an ALU request on the internal bus and sets a flag so that when the core is activated again, it will be able to continue operations at the appropriate stage of processing. There is an ALU controller that listens to these requests, and is described in detail in Subsection IV-C. The controller queues the requests until the ALU being requested is free. At that time, the controller will assign the operation to the ALU and listen for a result. When the ALU finishes and writes a result to the controller, the controller will then forward the result to the requesting algorithm core. The core will be notified by the ALU controller when a result is ready. Once the result is obtained, the algorithm core will use the previously set flags to step back into the encrypt/decrypt process where it interrupted to make an ALU request.

When the final result of the requested operation is complete, the core will write it to the I/O bus.

All cryptographic cores will follow this pattern, and interfaces will be common. SystemC^{TM}-based design makes this operation relatively simple. All interaction between modules is handled on "ports" and the internals of any particular core are of little concern.

Cryptographic cores for DES, and IDEA have been implemented.

B. Arithmetic Logic Units

Several ALUs have been implemented as part of the design. The most common operations in the cryptographic cores determine what ALUs are needed. Ideally, various different cores could be configured based on which cryptographic cores were in use. The most common operations for the particular cores that we designed are XOR and modular arithmetic operations. Each ALU is implemented similarly to the cryptographic cores: one SystemC^{TM}"port" for the internal bus. The inner workings of any particular ALU is of little concern. Each ALU will interpret blocks written to the internal bus differently.

C. ALU Controller

In order for cryptographic cores to use the dedicated ALUs, some sort of arbiter is necessary. A simple bus master is not sufficient since the status of each ALU needs to be tracked,

in addition to the status of the bus. The ALU controller will serve this purpose. Instead of having a bus in the traditional sense, each cryptographic core will have an internal bus to the ALU controller — which is much larger than the external bus by design — as well as a single wire for relaying wait status (further explained below). Each ALU also has its own bus to the controller as well as a wait status wire. The controller will track one request per cryptographic core. If a request is received by the controller and the appropriate ALUs are available, the request will be fulfilled immediately and the response written back to the requesting core when it is available. If the appropriate ALUs are not available, the controller will queue the request and set the wait status wire back to the core to indicate that there is a pending request. The algorithm modules, if another operation is needed for the busy ALU, will have to delay until the next cycle if the wait status wire is unset; this will be checked by the cores at the positive edge of every clock cycle. Queue depth is not an issue, because of the convention of not submitting further blocks of data to be processed until results are obtained from the previous block; thus, the queue in the ALU controller is one entry deep for each session as are many similar buffers throughout the design.

D. Microcontroller

The microcontroller handles all external communication from the system processor. Its primary function is to decode instructions and set the appropriate control flags for the cryptographic cores. A simple instruction set was defined to be interpreted by this unit. As an element of future work, the relevant standards could be applied to the instruction set as well as to the behavior of the microcontroller.

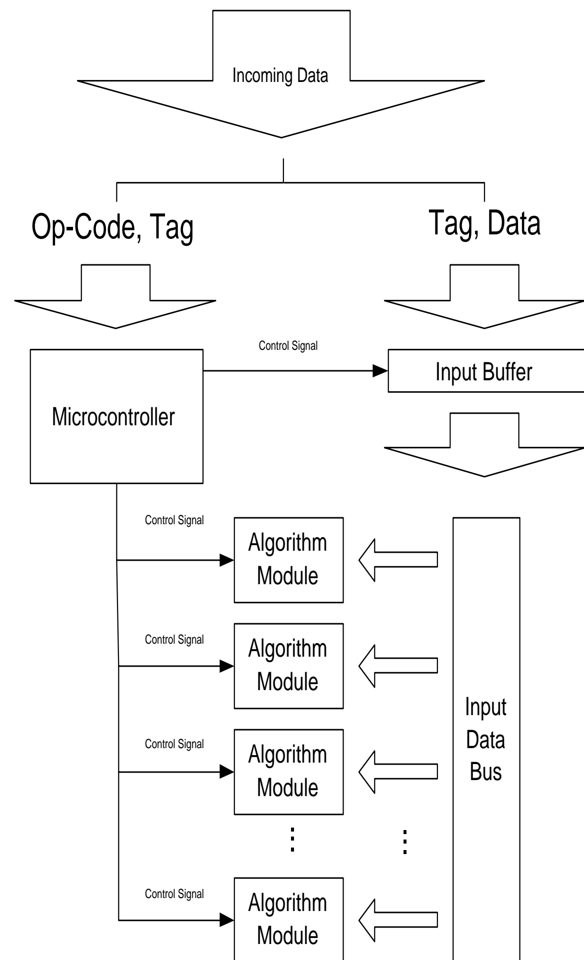

Fig. 5. ASEP Input Interface.

V. Architecture

A. Component Sharing

Popular encryption algorithms tend to use many similar arithmetic operations such as XOR, modular multiply, and GCD. Making requests to specialize ALUs to do the required operations rather than encapsulating the ALUs into each algorithm component can help achieve higher space and energy efficiency. Although it could create request conflicts during a very heavy system load, if a proper ALU bus controller is implemented, this conflict should not create significant system delay.

B. I/O System

The external input interface is 44 bits wide. As described earlier, the instruction include a 8 bit op-code, 4 bit tag, and 32 bit of data. Since no op-code is necessary for output, output interface contains only data and tag, which is 36 bits wide.

1) Input interface: Immediately after instruction comes into ASEP, the op-code and tag part of the instruction are routed to the microcontroller; the tag and data part of the instruction is routed to an input buffer which is controlled by the microcontroller.

The microcontroller then asserts a signal (CLEAR, FLUSH, READ) to the appropriate algorithm module to receive data and take proper action according to the op-code, and signals the buffer to release the data received from external interface into system I/O bus.

2) Output Interface: With parallel processing, it is possible to have more than one session to complete its operation at any given clock cycle. Therefore, the output needs to be internally buffered. Since by convention no more than one pending block from each session at any given time is allowed, only 16 buffer blocks, one for each session, are required to buffer all pending outputs. The output buffer is run by a round robin scheduler to select the session to output in the next clock cycle.

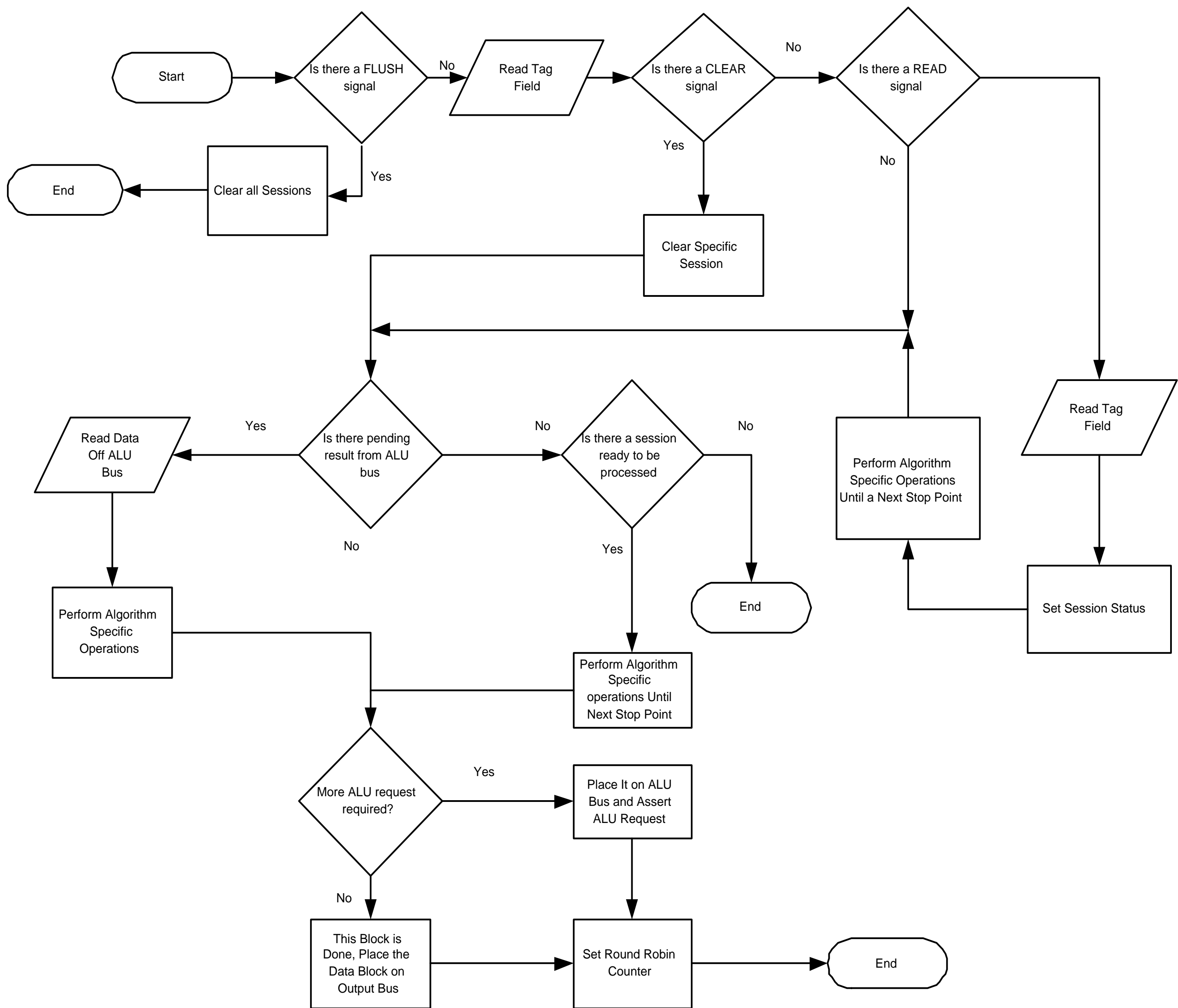

Fig. 7. ASEP Algorithm Module Flow Diagram.

C. Algorithm Module

We defined a standard interface to all algorithm modules. Each algorithm modules must have a 36-bit wide bus interface that connects to the main system I/O bus, and a 40-bit wide bus interface that connects to the ALU bus. Each module also have a 4-bit status lines and 4-bit control signal lines that connects to the central control module, which is attached to microcontroller. There will be a single encryption-block-sized buffer for each session. Each session has a status byte attached indicating what state is this session in. Also, three round robin counters (ALU request, Processing, and Output) are implemented to prevent starving between sessions. The ALU r-r counter indicates who gets the next ALU request if there is a conflict between sessions. The processing r-r counter indicates who can be processed for next algorithm specific operation in next op cycle if more than one session is ready to be processed. The output r-r counter will indicate who can be placed on output bus if there are more than one session has finished processing its current block.

D. Data Flow

When a instruction is received by ASEP, the microcontroller will first look up whether it is a special command such as CLEAR or FLUSH. If it is among one of the special commands, it will insert the control signal to the appropriate algorithm module for one clock cycle to let the algorithm module take the requested action. If the received command is a READ command, it will then place the received data and tag onto internal system I/O bus and signal the module for reading. Once the data has arrived at algorithm module, it will first check state of the session the data belongs to. If it is in a keying state, it simply stores the key and move on. If it is in an encryption/decryption state, the modules will then

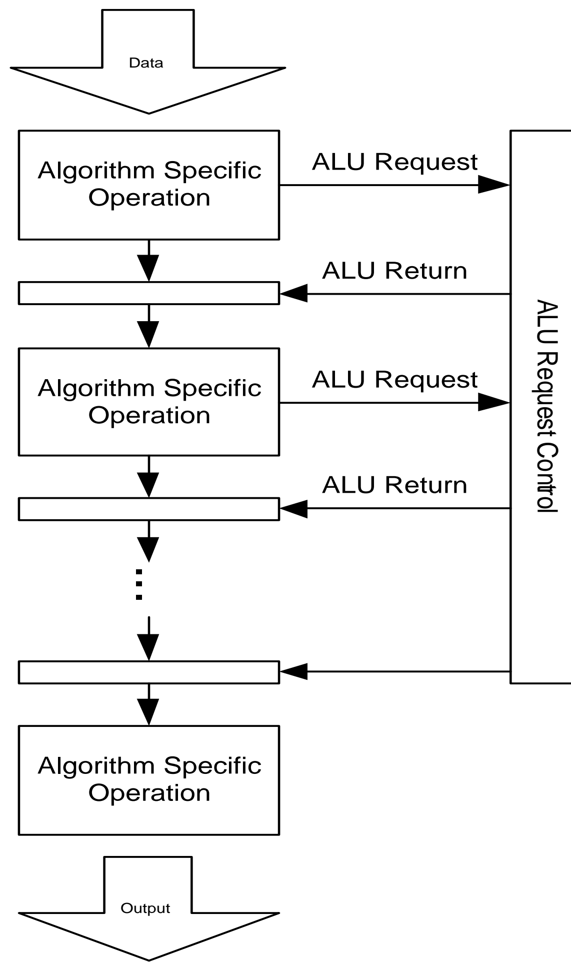

Fig. 6. ASEP Algorithm Module Pipeline.

perform algorithm specific operation within module itself until a "checkpoint" where it requires an operation from one of the specialized ALUs. At this point, algorithm module will check whether there is a result pending from the previous cycle. If the result signal of the ALU controller is set, it will then read ALU results from the ALU bus and place them in the session buffer. If there are multiple sessions, the algorithm module will refer to the round robin tables to see which session should be processed or placed on ALU bus in the current operation. At last, it will check the status of sessions and output round robin table (if necessary) to see which session is ready to be placed on output bus. This will end current operation.

E. Pipeline

Since encryption algorithms usually consist of interleaved algorithm specific operation and common arithmetic operations, when there are multiple session established at the same time, pipelining can be achieve by interleaving these operations between different sessions.

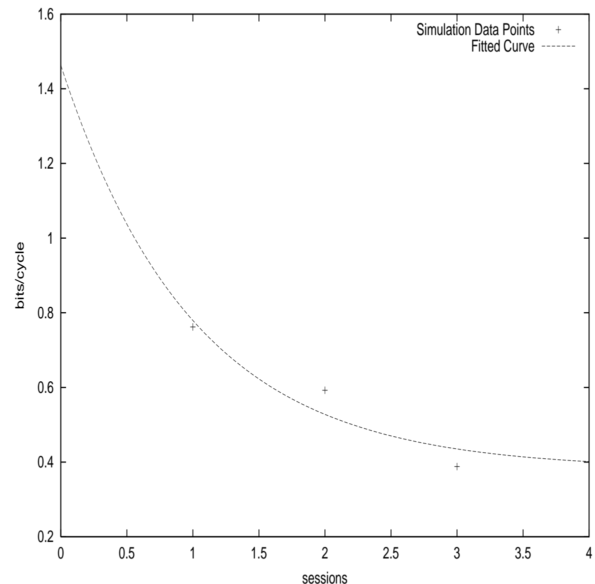

Fig. 8. ASEP Performance Degradation Estimate

VI. Results & Conclusions

In our simulations and analysis, we found that the ASEP has a bits processed to clock cycles ratio of 0.7619 (for single-session DES). By experimenting with various clock speeds, the achieved rate can be calculated. For example, if the clock is oscillating at 150 MHz, a throughput of 114 Mbits/second can be realized. Depending on the hardware used, this is competitive with other designs based on reconfigurable devices (e.g., FPGA devices), particularly when considering that the ASEP design does not yet incorporate optimizing features such as loop unrolling and pipelining. This throughput also shows that this design is more efficient than software alone. In multi-session mode, performance is somewhat degraded. In multi-mode, the ratios obtained using 2 sessions and 3 sessions were 0.5926 and 0.3879, respectively. See figure 8 for an estimate on performance degradation. However, the benefit of multi-session mode is the ability to accept requests from the system processor while an operation is currently being serviced, as well as a potential speedup over all operations even if there is a slowdown over any particular operation.

VII. Future Works

As a future work, we plan to develop optimizations that would be necessary to compete with commercially available products. Such optimizations include loop unrolling, pipelining, instruction reordering, and other features that are common to most modern processors. Work can be done to improve the I/O controller, the ALU request controller and algorithm cores to decrease the number of stalled cycles. More algorithm cores need to be implemented. In addition, low-power design issues will be considered, including dynamic voltage scaling. Appropriate coding techniques may also provide further improvements.

References

[1] S. Ravi, A. Raghunathan, and N. Potlapally, “Securing wireless data: system architecture challenges,” in *Proceedings of the 15th international symposium on System Synthesis*. ACM Press, 2002, pp. 195–200.

[2] J. Goodman and A. P. Chandrakasan, “Low power scalable encryption for wireless systems,” *Wireless Networks*, vol. 4, pp. 55–70, 1998.

[3] K. Lahiri, A. Raghunathan, S. Dey, and D. Panigrahi, “Battery driven system design: a new frontier in low power design,” in *Proceedings of ASP-DAC International Conference on VLSI Design*, January 2002, pp. 261–267.

[4] A. R. Nachiketh R. Potlapally, Srivaths Ravi, “Optimizing public-key encryption for wireless clients,” in *IEEE International Conference on Communications*, vol. 2, 2002, pp. 1050–1056.

[5] P. Chodowiec, P. Khuon, and K. Gaj, “Fast implementations of secret-key block ciphers using mixed inner- and outer-round pipelining,” in *FPGA*, 2001, pp. 94–102.

[6] M. Leong, O. Cheung, K. Tsoi, and P. Leong, “A bit-serial implementation of the international data encryption algorithm (idea),” in *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2000, pp. 122–131.

[7] O. Y. H. Cheung, K. H. Tsoi, P. H. W. Leong, and M. P. Leong, “Tradeoffs in parallel and serial implementations of the international data encryption algorithm IDEA,” *Lecture Notes in Computer Science*, vol. 2162, pp. 333–??, 2001.

[8] O. Mencer, M. Morf, and M. Flynn, “Hardware software tridesign of encryption for mobile communication units,” in *Proceedings of ASSP ’98*.

[9] J. Goodman and A. P. Chandrakasan, “An energy-efficient reconfigurable public-key cryptography processor,” *IEEE Journal of Solid-State Circuits*, vol. 36, no. 11, pp. 1808–1820, November 2001.

[10] R. R. Taylor and S. C. Goldstein, “A high-performance flexible architecture for cryptography,” in *Springer Verlog Lecture Notes in Computer Science #1717*. CHES, 1999.

[11] A. Dandalis, V. K. Prasanna, and J. D. P. Rolim, “An adaptive cryptographic engine for ipsec architectures,” in *IEEE Symposium on Field-Programming Custom Computing Machines Proceedings*. IEEE, April 2000.

[12] A. Elbirt and C. Paar, “Instruction-level distributed processing for symmetric-key cryptography,” in *International Parallel and Distributed Processing Symposium Proceedings*. IPDPS, April 2003.

[13] H. Schmit, D. Whelihan, A. Tsai, M. Moe, B. Levine, and R. R. Taylor, “Piperench: A virtualized programmable datapath in 0.18 micron technology,” 2002.

[14] V. R. Joan Daemen, “Aes proposal: Rijndael.” [Online]. Available: http://csrc.nist.gov/CryptoToolkit/aes/round2/r2algs.htm

[15] P. R. Schaumont, H. Kuo, and I. M. Verbauwhede, “Unlocking the design secrets of a 2.29 gb/s rijndael processor,” in *Proceedings of the 39th conference on Design automation*. ACM Press, 2002, pp. 634–639.

[16] “Data encryption standards,” vol. 46, no. 3, 1999.

[17] “Data encryption standard,” vol. 46, no. 2, 1993.

[18] “Des challenge 3,” 1999.

[19] B. Schneier, *Applied Cryptography: protocols, algorithms, and source code in C*. Wiley, 1996.