IBM POWER7+ processor on-chip accelerators for cryptography and active memory expansion

B. Blaner
B. Abali
B. M. Bass
S. Chari
R. Kalla
S. Kunkel
K. Lauricella
R. Leavens
J. J. Reilly
P. A. Sandon

With the heightened focus on computer security, IBM POWER® server workloads are spending an increasing number of cycles performing cryptographic functions. Active memory expansion (AME), a technology to dynamically increase the effective memory capacity of a system by compressing and decompressing memory pages, is also enjoying increasing deployment in POWER server systems. Together, cryptography and AME consume enough central processing unit (CPU) cycles in a typical installation to warrant adding dedicated hardware accelerators on the processor chip to offload the compute-intensive parts of these functions from the processor cores. IBM POWER7+™ is the first POWER server to include on-chip hardware accelerators for symmetric (shared key) and asymmetric (public key) cryptography and memory compression/decompression for AME. A true random number generator (RNG) is also integrated on-chip. This paper describes the hardware accelerator framework, including location relative to the cores and memory, accelerator invocation, data movement, and error handling. A description of each type of accelerator follows, including details of supported algorithms and the corresponding hardware data flows. Algorithms supported include the Advanced Encryption Standard, Secure Hash Algorithm, and Message Digest 5 algorithm as bulk cryptographic functions; asymmetric cryptographic functions in support of RSA and elliptic curve cryptography; and a novel dictionary-based compression algorithm with high throughput supporting AME. A presentation of accelerator performance is included.

Introduction

Secure computing is of growing concern to computer users. IBM POWER server systems offer state-of-the art secure computing technologies, allowing authentication, data privacy and data integrity solutions to be deployed in POWER data centers. The IBM AIX operating system (OS) supports a number of security technologies. One such technology is the encrypted file system (EFS), which uses symmetric key cryptography to keep files private. Others include network security protocols such as Internet Protocol Security (IPSec) and secure sockets layer (SSL). These use asymmetric cryptography algorithms such as RSA and elliptic curve cryptography (ECC) for secret key exchange, the secure hash algorithm (SHA) for message integrity checking, and the Advanced Encryption Standard (AES) symmetric key algorithm for bulk data encryption.

Since AIX version 6.1, active memory expansion (AME) [1] has been offered as a technology for expanding the effective memory capacity of an IBM POWER system and has enjoyed growing deployment. AME transparently compresses in-memory data, allowing more data to be placed into memory, thus expanding the system memory capacity. When an application needs to access data that is compressed, the OS automatically decompresses the data and makes it available to the application. CPU cycles are

Digital Object Identifier: 10.1147/JRD.2013.2280090

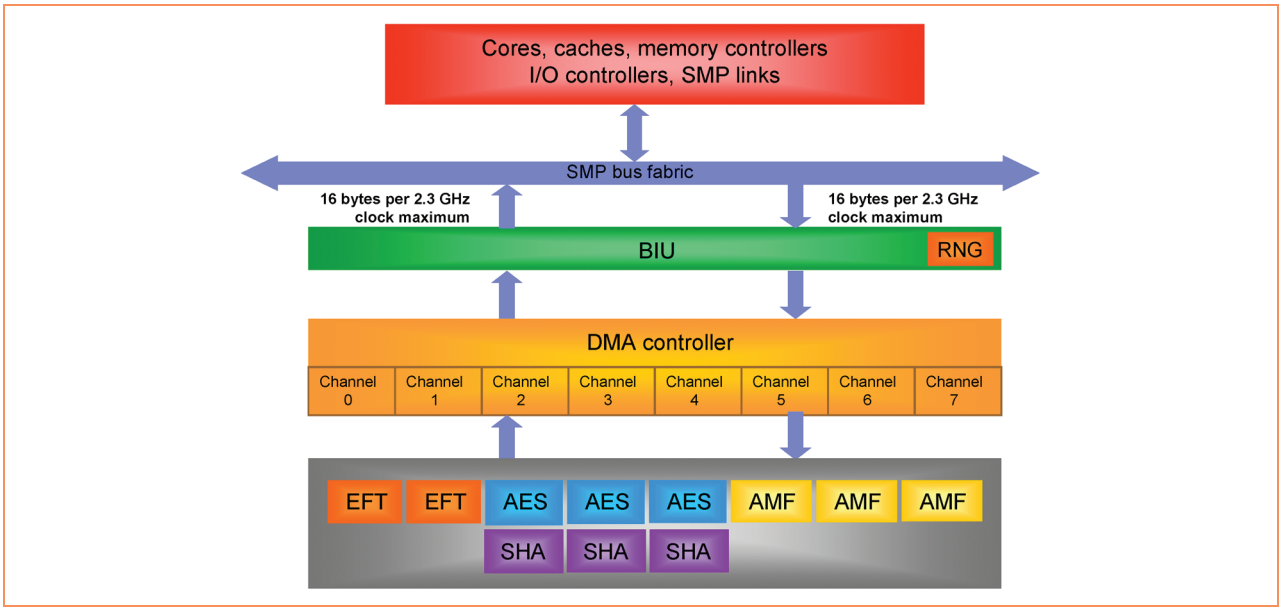

Figure 1

Accelerator complex in POWER7+.

expended for AME compression and decompression, in an amount that varies with workload and the level of memory expansion desired.

Contemporary POWER systems are shipped with PowerVM Hypervisor [2], a firmware layer running directly on the hardware, which virtualizes the hardware, allowing hundreds of virtual machines to run concurrently on a system, each isolated from the other in its own logical partition (LPAR). In a modern, large POWER system, hundreds of LPARs may be deploying OS security technologies and AME simultaneously. Early in the POWER7+ design process, it became apparent that system performance could benefit by off-loading to hardware accelerators some of the CPU cycles being spent on cryptography and AME, and the decision was made to include a complement of accelerators as a central resource on the processor chip. While various tradeoffs apply to the use of processor instructions to accelerate these functions [3], tightly coupled coprocessors [4, 5], off-chip acceleration [5], or on-chip loosely coupled accelerators, the latter approach was selected for POWER7+ to achieve a high level of performance and CPU off-load with low impact on chip area and design.

The accelerators are shared among the LPARs under the control of the hypervisor. The hypervisor manages quality of service, i.e., maintains fairness among the LPARs using the accelerators and translates LPAR logical real addresses to system real addresses for use by the accelerators. The complement of accelerators comprises three types most essential for cryptography and AME: AES and SHA bulk cryptography engines; asymmetric mathematical functions (AMF) engines, performing functions in support of RSA and ECC; and AME memory compression/decompression engines with high throughput and good compression efficiency. The engines are provided in sufficient quantity to service typical workloads of the eight-core chip. A true RNG is also provided in support of cryptography, as described by Liberty et al. [6].

This paper is organized as follows: First, we present the accelerator complex, including the logical location of the complex on the chip, accelerator invocation, data movement, and error handling. Then a section is dedicated to each of the symmetric cryptography accelerators, AMF accelerator, and memory compression/decompression for the AME accelerator. Measured performance results are presented, and then the paper concludes.

Accelerator complex

Figure 1 illustrates the accelerator complex. The POWER7+ chip integrates four different accelerators in these quantities: three each of AES, SHA, and AMF engines, and two eight-four-two (EFT) memory compression/decompression engines. The accelerators are attached to the symmetric multiprocessor (SMP) interconnect fabric, allowing access to all of the system memory through a bus interface unit (BIU) and DMA controller. The BIU performs transactions on the fabric on behalf of the accelerators, and the DMA controller performs command and data movement on their

behalf. The fabric is a split-transaction bus with separate command/address/response, data-in, and data-out buses. The primary unit of transfer on the bus is the 128-byte cache line, although partial-line transfers are permitted. Cache lines may be received out of order with respect to requested order. Data buses in and out of the accelerator complex are 16-bytes wide, delivering a peak transfer rate of 37 GBps per direction at the 2.3 GHz fabric clock frequency. Attached on-chip to the fabric are the eight local processor cores and caches, memory controllers, I/O controller, and memory-coherent off-chip links to other processor chips comprising the system.

An accelerator is invoked by the hypervisor issuing the `invoke coprocessor` instruction. (The terms "coprocessor" and "engine" will be used synonymously with "accelerator" in this paper.) The instruction has a 128-byte memory operand called a coprocessor request block (CRB). The CRB contains all the address and control information necessary for an accelerator to execute a job, where a "job" is a fundamental unit of work of an accelerator. For example, the AES engine encrypting scattered 512-byte blocks of memory is a *job*, the AMF engine performing modular exponentiation on one set of operands is another, and the EFT engine decompressing a 4 KB page is another. The CRB contains, among other things, the following information:

- Routing information to select a particular accelerator complex and accelerator type in the system, type being one of symmetric cryptography, asymmetric math functions (AMFs), or compression/decompression.
- Function control field, which identifies further the operation to be performed, for example, AES, SHA, AMF modular exponentiation, or EFT decompress.
- An address pointing to a coprocessor status block (CSB) and contiguous coprocessor parameter block (CPB). The optional CPB may contain further inputs for a job, for example, the particular cipher mode for an AES job, the key and initialization vector (IV) for that job, or all the operands for an AMF job. The CPB may also be updated with certain outputs from a job, for example, the continuation vector of an AES job, allowing a large job to be divided into multiple smaller jobs. The CSB will be updated with the completion status of a job, i.e., completion with or without error, and in the case of completion with error, a code identifying the particular type of error. It may contain other information such as the number of target bytes written by a compression operation.
- Source (input) and target (output) data descriptors comprising an address pointing to either source data or a gather list (a list of pointers to source data), an address pointing to either target data or a scatter list (a list of pointers to target data), and lengths of the source and target data.
- Job completion controls, for example, whether a job is to generate an interrupt upon completion.

When the `invoke coprocessor` instruction is executed, a coprocessor request command is issued on the bus fabric together with the CRB. Assuming the routing information is correct and the BIU has a place for the CRB in its command queues, the on-chip accelerator complex will accept the request and queue the CRB in one of three queues, one for each of the three accelerator types. A coprocessor request waits in a queue until the DMA controller has sufficient resources to process the command.

The DMA controller comprises eight channels: one per EFT engine, one per AMF engine, and one per pair of AES and SHA engines. Each channel services the attached engine(s) by signaling that a new coprocessor request is available, by prefetching and buffering CPB and source data, and by supplying such data to the engine upon request. The channel also buffers any output data from the engine. The output data can comprise target data, optional CPB updates, and completion status data, which may include an error indication and a request to generate an interrupt.

The DMA controller data bandwidth to the engines is up to 16 bytes per bus fabric clock per direction. In order to sustain this bandwidth and hide the latency between accelerator invocations, the DMA controller has 12 KB of SRAM buffer storage for cache lines heading to and from the accelerators. Concurrently, each DMA channel can have one CRB in progress with an accelerator for which it has fetched source data and is buffering target data, plus one queued CRB for which it is also prefetching CPB and source data. To service the various DMA data access requests at maximum throughput, the BIU has 16 independent read machines and 16 independent write machines, each capable of reading or writing a cache line from or to the bus fabric (and ultimately to system memory or a processor cache), allowing 32 fabric operations to be in flight concurrently. Between these parallel machines and the DMA controller's prefetching capabilities, most memory access and fabric latencies may be hidden, allowing 16-byte data transfers to be continuously streamed to and from the accelerator complex. The BIU read machines may return data out of order with respect to the DMA-requested order. The DMA controller restores the order in its cache line buffers so that data are always presented to the engines in order.

As stated earlier, a job may terminate with an error. Opportunities for both programming and hardware errors occur in several places in the control and dataflow of the accelerator complex. A great deal of attention has been paid to detecting these errors and handling them robustly in the accelerator complex design. Some errors may be detected by the DMA controller, such as target data buffer overrun. Other errors are detected by the accelerators themselves, such as invalid operands for an AMF modular exponentiation

Table 1 Supported bulk cryptographic algorithms.

Algorithm (mode)	*Key size (bits)*	*Digest/MAC size (bits)*	*Algorithm type*
AES (ECB, CBC, CTR)	128, 192, 256		Encryption
AES (CCA, GCA, GMAC)	128, 192, 256		Authentication
AES (CCM, GCM)	128, 192, 256		Authenticated encryption
AES (XCBC-MAC-96)	128		Authentication
MD5		128	Hash
SHA-1		160	Hash
SHA-2		256, 512	Hash
SHA-1 HMAC		160	MAC
SHA-2 HMAC		256, 512	MAC

operation. Still other errors, such as uncorrectable SRAM errors, are detected by the hardware wherever SRAMs are used. (Single bit error correct and double bit error detect error correction codes are used on all SRAMs in the accelerator complex.) In most cases, such errors lead to the termination of the associated job and to the DMA controller writing an error completion status to the CSB. Job processing then continues as normal. Hypervisor intervention is required in only a few rare cases, such as an uncorrectable error on the CSB address itself, which leaves the DMA controller with no place to write an error code.

A job may optionally generate an interrupt upon completion, obviating the need to poll for completion status. The CRB contains a control bit to indicate whether an interrupt is to be generated. If it is, the DMA controller signals the interrupt request to a local interrupt controller in the BIU, which in turn forwards the request on the bus fabric to the chip interrupt controller. Up to 16 interrupts may be in flight, allowing for overlap of software interrupt handling with running new jobs that also generate interrupts.

The true RNG is attached to the BIU and is accessed by an MMIO (memory-mapped input/output) load instruction with an address mapped to the accelerator complex address space. The `invoke coprocessor` instruction and DMA controller are not used to access the RNG.

Symmetric cryptography engines

The symmetric cryptography engines support a range of cryptographic algorithms that can be applied to bulk data. The AES engine implements several modes for several key sizes of the AES algorithm for use in encryption and authentication. Because it processes bulk data and is tightly coupled to the AES engine to support the combination mode operations (discussed later), the SHA engine is described here as part of the symmetric cryptography engine complex. The SHA engine implements the MD5 algorithm (a precursor to SHA), as well as the SHA-1 and SHA-2 algorithms for hashing and the hashed message authentication code (HMAC) based on the SHA-1 and SHA-2 hash algorithms for authentication. **Table 1** summarizes the supported bulk cryptography algorithms.

Symmetric operations

Symmetric key cipher algorithms, also known as shared key algorithms, use the same key for encryption and decryption of a message. The encryption of a single block of data is described by $C = E(\mathrm{K}, P)$, where C (ciphertext) is the result of encrypting the block P (plaintext) under the key, K. The plaintext can be restored by a decryption operation (D) described by $P = D(\mathrm{K}, C)$. The AES specification [7] defines the algorithm for three key sizes of 128 bits, 192 bits, and 256 bits, all using a single block size of 128 bits.

The AES engine implements the AES algorithm for all three key sizes and for six modes of operation, all based on the single block operation just described. The ECB (electronic code book) mode applies the single block computation independently to each block of plaintext to produce the corresponding ciphertext. Using subscripts to denote successive blocks of plaintext and ciphertext, ECB mode can be described as $C_i = E(\mathrm{K}, P_i)$. The CBC (cipher block chaining) mode takes the ciphertext result of one block and exclusive-ORs it with the next plaintext block before

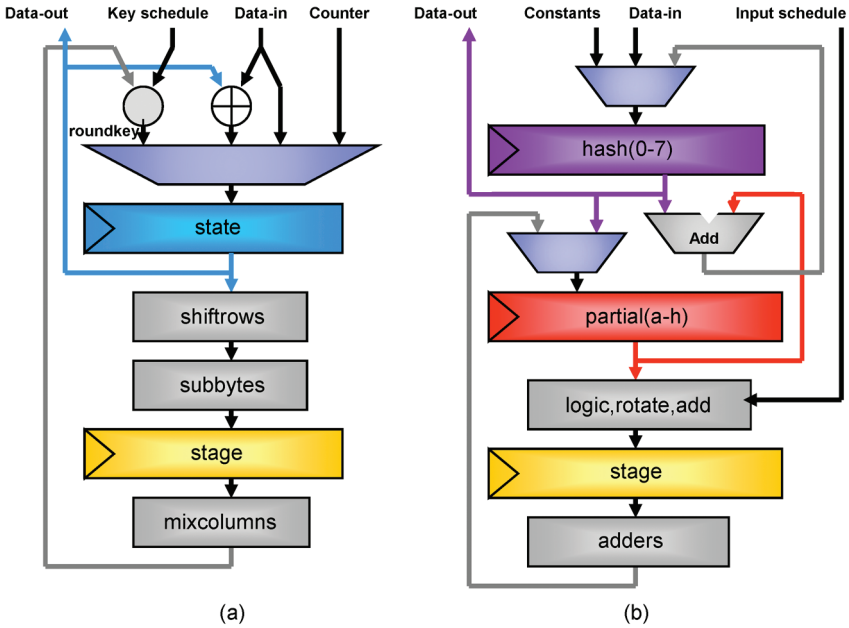

Figure 2

Bulk cryptography dataflows.

applying the encryption operation. This can be described as $C_i = E(\mathrm{K}, P_i \text{ XOR } C_{i-1})$. The CTR (counter) mode encrypts the value of a counter that increments for each block, then exclusive-ORs that result with the plaintext, described as $C_i = E(\mathrm{K}, CTR_i) \text{ XOR } P_i$.

The CCM (counter with CBC-MAC) mode combines CTR mode for encryption with CBC mode for authentication of a single data stream. Such combined modes sometimes require that some data be included in the authentication operation, but not in the encryption operation. Such additional authentication data (AAD), if more than one block, constitutes a second data stream that must be processed using an auxiliary operation supported by the AES engine, called CCA. The partial authentication tag (PAT) value computed over that AAD can then be used as an input to the CCM operation when encrypting and authenticating the plaintext itself.

The fifth AES mode is GCM (Galois counter mode), which also provides encryption plus authentication. In this case, the authentication operation uses Galois field (finite field) multiplication, while encryption is done using CTR mode. In a manner similar to CCM mode, AAD is processed using an auxiliary operation called GCA, if necessary, and the result of that operation is used by the GCM operation to complete the authentication part of the operation. In addition, when only authentication is required, a GMAC operation is available to compute a complete Galois field MAC over the data stream.

Finally, the AES XCBC-MAC-96 (XMAC) mode, used for authentication, computes a message authentication code (MAC) over a data stream. The MAC is essentially the final block of ciphertext produced by an AES-CBC operation applied to the message, except that several keys derived from the input key are used in the computation, and special processing is done on the final block of data that depends on whether it is a partial or complete block.

The AES algorithm applied to one block of data applies a sequence of substitutions and permutations to the input data, while combining that data with key schedule data derived from the original key. For a given key size, the computation proceeds in a number of "rounds", where each round of computation involves a sequence of four steps named shiftrows, subbytes, mixcolumn and roundkey. Briefly, these permute bytes within the block, substitute each byte with another from a table, apply a linear transformation, and exclusive-OR the key schedule value, respectively. The AES engine executes the shiftrows and subbytes steps in the first cycle, and the mixcolumn and roundkey steps in the second cycle for each round.

Figure 2(a) schematically shows the 128-bit wide dataflow for the AES block computation. For an ECB mode encryption, the first block of plaintext comes into the *state* register from the *Data-in* input bus. The *state* register value then undergoes the *shiftrows* and *subbytes* transformations in the first cycle of round 1, and that result is placed in the *stage* register. The *stage* register value then

undergoes the *mixcolumns* and *roundkey* transformations in the second cycle of round 1, and that result is placed back in the *state* register. The *roundkey* step is implemented by the exclusive-OR function shown at the top-left of the figure, using a particular value from the key schedule that is generated in separate logic from the encryption key. This two-cycle sequence is repeated for each round of the computation—10 rounds for AES 128, 12 rounds for AES 192, and 14 rounds for AES 256. At the end of the required number of rounds, the *state* register contains the ciphertext block corresponding to the plaintext block, and so it is put out on the *Data-out* bus. AES decryption requires a slightly different dataflow but uses the same overall structure.

In the case of ECB mode, the *state* register gets each successive block of plaintext from the *Data-in* bus, and that block is processed as just described to produce each successive block of ciphertext. In the case of CBC mode, the *state* register is first initialized with an "initialization vector," which is then exclusive-ORed with the first block of plaintext before it is processed. The ciphertext computed from that first block is put out on the *Data-out* bus and is exclusive-ORed with the next block of plaintext, as shown at the top-center of the figure, and placed in the *state* register. That block is then processed as just described. For CTR mode, the *state* register gets its value from the *Counter* register for each block of data processed. The counter value is encrypted as described above, and then that value is exclusive-ORed with the next plaintext block to produce the next ciphertext block.

The other AES modes require additional dataflow elements and introduce corresponding control complexity in the engine design to support the operation sequences defined for those modes. Similarly, the decryption operation for some modes requires a slightly different dataflow than the encryption operation. However, the basic dataflow shown in Figure 2(a) is used in all cases for the encryption of each individual block of data.

Hashing operations

A cryptographic hash operation can be used to check the integrity of a data stream, and as part of an authentication operation. A hashing operation uses a one-way function to map an arbitrary sized message to a fixed size digest. The SHA engine implements the Secure Hash Algorithm (SHA) version 1 [8], which generates a 160-bit digest, and version 2 [9], for both the 256-bit and 512-bit digests. These are also known as the SHA-1, SHA-256 and SHA-512 operations, respectively. It also implements the message digest 5 (MD5) algorithm [10], which generates a 128-bit digest. In addition, the SHA engine implements the hash-based message authentication code (HMAC) operation [11] associated with each of the three supported SHA operations, referred to as the SHA-1 HMAC, SHA-256 HMAC, and SHA-512 HMAC operations.

Figure 2(b) shows the dataflow for the hashing function. The four hash functions have a similar structure for this computation but use different word sizes, numbers of words, and mixing functions. The word size is 64 bits for SHA-512 and 32 bits for all others. The number of words in the dataflow corresponds to the size of the digest being computed, and so it is four words for MD5 (4×32 bits $= 128$ bit digest), five words for SHA-1, and eight words for the SHA-2 algorithms.

At the beginning of a hashing operation, the *hash* word registers, 0 through 7, are initialized with constant values specific to the particular algorithm. That value is also copied into the *partial* result word registers, *a* through *h*. Each round of computation takes two cycles. In the first cycle, the mixing functions corresponding to the particular algorithm, including logical operations and rotates, are applied to the words in the *partial* registers. These mixing functions and the separate function used to generate the message schedule from the current message block are described in the specification for each algorithm. The output words of the mixing functions are combined with a word from the *message schedule* using word addition with no carry-out, and those results are latched in the *stage* register. On the second cycle of each round, additional word-wide adders are used to compute the values that update the *partial* registers. This two-cycle round of computation is then repeated for some number of rounds, in which the current *partial* value and the next word from the *message schedule* provide the inputs. At the end of 64 rounds for MD5 and SHA-256, or 80 rounds for SHA-1 and SHA-512, the *partial* words are added to the corresponding *hash* register value and are stored in the *hash* registers. This process is then repeated for each block of the message, including a final, appropriately padded input message block. The final value in the hash registers after all blocks of the message have been processed constitutes the message digest, which is put out on the *Data-out* bus.

The HMAC operations use the same hashing computations just described, but incorporate a key and some special processing at the beginning and end of the operation. The key is combined with a constant to generate a first block of data that is prepended to the message. The result is hashed to produce an initial digest. The key is then combined with a second constant to generate a block of data that is prepended to the initial digest. The result is hashed to produce the final digest, which constitutes the MAC.

Authentication-encryption operations

In addition to operations that implicitly provide both authentication and encryption, such as the CCM and GCM modes of AES, some protocols use a combination of a standalone authentication operation with a standalone

encryption operation, such as SHA HMAC with AES-CBC. Serial execution of these two operations on a given message can achieve the desired combination of authentication and encryption. However, provision of a single command to achieve that same result reduces the memory bandwidth requirements, reduces the latency of the compound operation, and increases the available encryption and authentication throughput of the bulk cryptography engines.

The combinations of authentication and encryption modes supported in the accelerator complex are any SHA-1 or SHA-2 HMAC with any key length AES CBC or AES CTR mode. For any such combination, all three commonly used methods to compose the authentication and encryption operations are supported. These are the encrypt-then-authenticate (E-then-A) method, used by IPSec; the authenticate-then-encrypt (A-then-E) method, used by SSL; and the encrypt-and-authenticate (E-and-A) method, used by SSH. The corresponding authentication and decryption modes are also available. Support for additional authentication data is included in these combined modes as well.

The authentication-encryption combination operations are implemented using the AES and SHA engine pair attached to a particular DMA channel. The data buffering in those engines that is normally used for input and output data between the engine and the DMA channel is shared with that function to also provide data buffering between the engines. For example, the E-then-A method is implemented by streaming the plaintext data into the AES engine, then sending the resulting ciphertext both back to the DMA channel and to the SHA engine. The SHA engine computes the HMAC over the ciphertext stream and supplies the result to the DMA channel after all the ciphertext has been transferred.

Asymmetric math function accelerator

The AMF accelerator has several functions to support two common cryptographic algorithms used for public key cryptography. The first is RSA, an algorithm that is built on modular arithmetic operations over large numbers and whose security is based on the difficulty of factoring these large numbers. The AMF accelerator provides several functions for modular arithmetic including modular exponentiation, $R = A^B \bmod N$, to support the RSA algorithm for operand sizes of 512, 1024, 2048, and 4096 bits. Another popular public key algorithm that is supported is ECC where the key operations are based on arithmetic over elliptic curves and whose security is based on the difficulty of solving the discrete logarithm problem in this context. The AMF accelerator provides functions to do point multiplication on an elliptic curve over a prime field $[(\mathrm{X_r}, \mathrm{Y_r}) = (\mathrm{X_p}, \mathrm{Y_p})^*\mathrm{K}]$ using 192, 224, 256, 384, and 521 bit numbers or over a binary field $([\mathrm{X_r}(x), \mathrm{Y_r}(x)] = [\mathrm{X_p}(x), \mathrm{Y_p}(x)]^*\mathrm{K})$ using 163, 233, 283, 409, and 571 bit numbers.

RSA encryption and decryption operations

The RSA algorithm uses a pair of keys, one for encryption, the other for decryption. Let n be a large number chosen as the product of two prime numbers p and q. The value n is the modulus and all arithmetic is done modulo n. The encryption (public) key, denoted by the ordered pair (e, n), is known by all, and is used to encrypt the message m (plaintext) using

$$c = m^{\mathrm{e}} \bmod n.$$

Corresponding to this public key, a private key (which is always kept secret) is chosen using the equation

$$(\mathrm{e}^* \mathrm{d}) = 1 \bmod [(p-1)(q-1)].$$

With this relation, the decryption (private) key, (d, n), can decrypt a message using

$$m = c^{\mathrm{d}} \bmod n.$$

The security of the RSA algorithm relies on the difficulty of factoring n into the component primes p and q. These two factors must also be kept secret, since if either factor is known then the private exponent d can be determined from the public key and thus the cryptosystem will be compromised.

Since modern applications require the use of key lengths of 1024 bits and larger for secure operation, these calculations can take a considerable amount of processing time. The AMF accelerator provides a modular exponentiation function to offload the processor. For the function

$$R = A^B \bmod N,$$

the processor provides the operand A, the exponent B, and the modulus N. The accelerator will return the result, R. This function can be used with either the public or private key to perform the encryption or decryption operation.

A second RSA function, modular exponentiation using the Chinese remainder theorem (mexpCRT), is also supported. It can be used by the holder of the private key to decrypt the encoded message. The basic premise of the mexpCRT is that the private key operation can be split into two approximately half-size modular exponentiations. The two results can be calculated and combined more quickly than the original larger modular exponentiation.

In addition, the AMF accelerator provides a modular multiplicative inverse function (Minv):

$$R = A^{-1} \bmod N.$$

The accelerator computes the result, R, using the extended Euclidean algorithm.

ECC encryption and decryption operations

ECC uses operations on points on elliptic curves defined over prime or binary fields to define encryption and decryption operations, as well as related functions, such as key exchange and digital signing. Elliptic curves over a prime field are defined as points on the graph of the equation

$$y^2 = x^3 + ax + b,$$

where a and b are from the underlying field. Given two points P and Q on the elliptic curve, the point P + Q (point addition) results in another point on the elliptic curve. Similarly, given a point P, the point multiplication operation, kP, is defined as P added to itself k times using point addition, where k is an integer. For binary fields, the definition is slightly more general on the underlying curves. The AMF accelerator provides support for the five prime field pseudo-random curves and the five binary field pseudo-random fields recommended by NIST [12].

ECC is becoming increasingly popular in certain applications, in particular those that are deployed on low-powered, end-user devices. The strength of these cryptosystems relies on the difficulty of the discrete logarithm problem, i.e., given a point B, known to be kA for some point A, find k, the discrete logarithm of B. The discrete logarithm problem over elliptic curves is conjectured to be harder than the corresponding factoring problem over regular prime fields and thus the key sizes for ECC are smaller than those required for the RSA algorithm for the same targeted level of security.

The AMF accelerator provides several functions to support each of prime field and binary field ECC. These include point multiplication, which operates on ordered pairs representing points on the elliptic curve, as well modular multiplication, modular reduction and modular inverse, which operate on elements of the prime or binary field. The point coordinates are integers belonging to the prime field, or are polynomials with binary coefficients to represent elements of the binary field. The corresponding modulus in the first case is the prime number defining the field, while in the second case is an irreducible polynomial of order equal to the key size.

AMF microarchitecture

A block diagram of the AMF accelerator is shown in **Figure 3**. The *channel* interfaces with the DMA controller to move the initial parameters into the accelerator's internal registers and to return the results once the calculation is complete. The interface to the DMA controller is 130 bits wide and the internal dataflow within the AMF is 65 bits wide. The choice of the non-power-of-two 65-bit width was to reduce the processing time. Modular math requires two extra bits to handle overflow and a sign. Since the RSA keys have sizes of order 2^n, using a standard order of 2^n dataflow would have resulted in an additional multiplication per loop and an extra loop per multiplication. In addition to moving data, the channel decodes each command to determine the parameter set and to route the parameters to the correct registers. This includes the preload of any registers with constants needed for the given operation. While transporting the parameters, the channel will perform basic checks on a parameter to determine if it is odd, zero, or negative and of the correct size. A parameter that fails a check will cause an abort of the operation with the appropriate return code sent back to the DMA controller.

The register bank (*regs* in the figure) consists of eleven 64 × 73 bit arrays. These arrays hold the initial parameters, the intermediate data and the results of the AMF operations. To accommodate the 4K-bit parameters of RSA operations, the arrays are defined as eleven 4160-bit registers. For the smaller-sized ECC operations, which require more registers, the arrays are split into a high and low half for a total of 22 (2080-bit) registers. There are two 65-bit write buses into the register bank and each can write all the registers at the same time. There are four 65-bit read buses from the registers with each bus having access to only a subset of the registers. Register entries are protected with single bit error correction or double bit error detection codes.

The *red* unit in Figure 3 contains the computational dataflow to perform addition, subtraction, and shifting. There are two, parallel, 3-cycle, 65-bit paths through this unit. The two paths allow for the modulus to be shifted at the same time that an add or subtract is occurring on the two operands. The main data path has a pre-adder shift, an adder, and a post-adder shift. The second path of the dataflow has an adder and a post-adder shifter. These shifters in the path of the adders enable compound functions like doubling of the modulus before using it as an operand for parameter checking, or for halving after a subtraction as needed by multiplicative inverse calculations. The dataflow of this unit uses all four read ports and both write ports to the register bank.

The *mg* unit contains the computational dataflow to perform the multiplication type operations of the AMF accelerator. This unit consists of two, parallel, 4-cycle, 65-bit pipelined multipliers. One multiplier is for the two operands, the second is for the multiplication of the modulus. Following the multipliers are three stages of multiplexers (muxes) and adders such that all of the intermediate values and carries in a single pass of multiplication stay within this dataflow, and only the final output of the pass is stored back to the register bank. This dataflow uses all of the read buses from the register bank, but only uses one of the write buses.

The *cntl* unit consists of 11 state machines that orchestrate the access of the register bank and the control of the dataflows. Some state machines call other state machines that are building blocks of higher-level functions. This can result

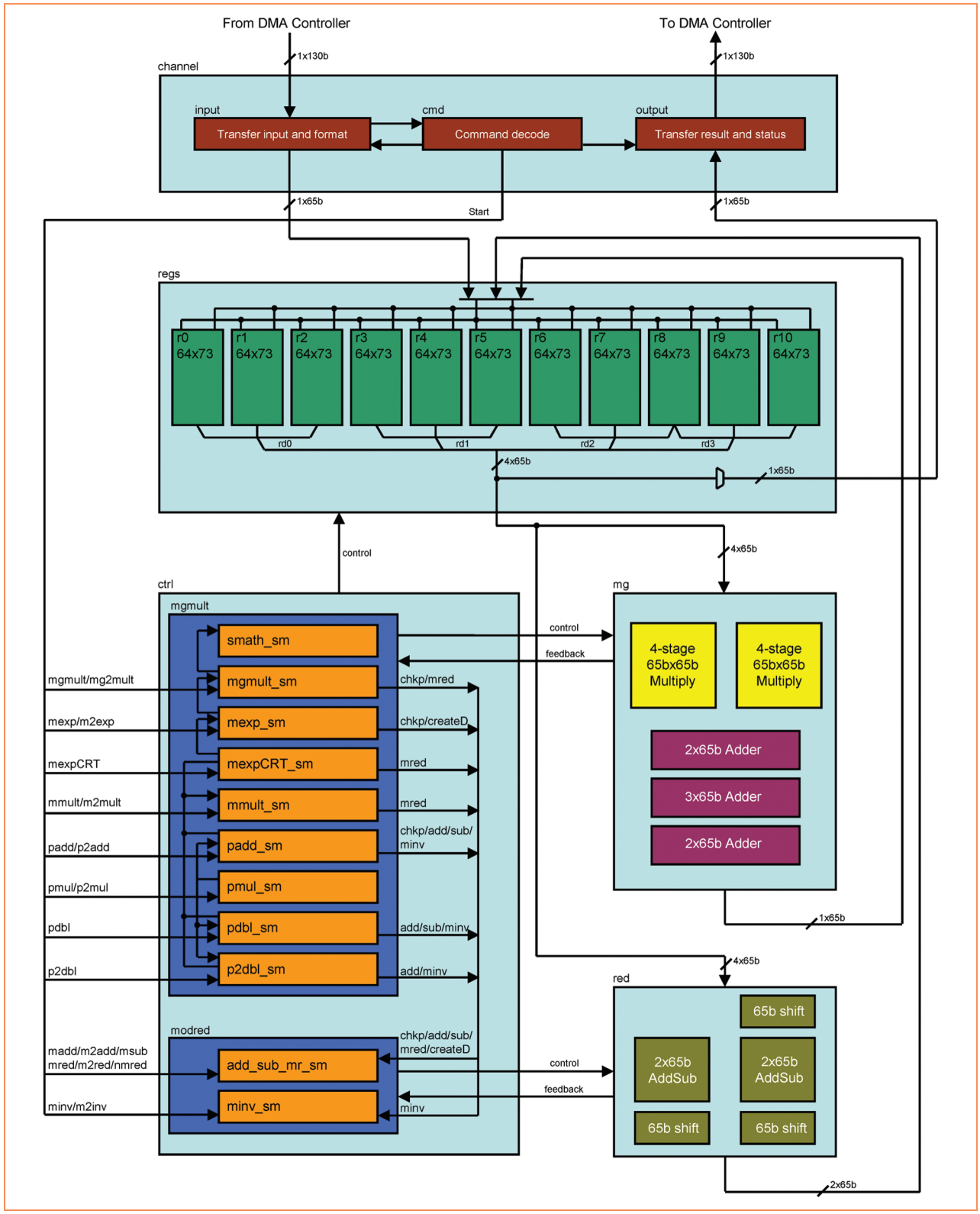

Figure 3

AMF accelerator dataflow and control elements.

in several state machines being active at a time. For example, the point multiply state machine calls the point double, which calls the multiplication, which calls the modular reduction. The AMF accelerator supports commands for higher level functions, like modular exponentiation and point multiply, as well as for lower level functions, like Montgomery multiply, modular addition, and modular reduction, that can be used as building blocks for more complex computations.

As an example of the data and control flows in the AMF accelerator, consider the modular exponentiation function. A naive implementation of computing

$$R = A^B \bmod N$$

would multiply A with itself, reduce modulo N, then multiply that result by A and reduce modulo N repeatedly. There are a number of optimizations that can be applied to this computation. One is the use of the Montgomery multiplication (mgmult) operation [13] that allows most of the time consuming modular reduction operations to be omitted. Another is that instead of multiplying by A on each step, the partial result can be squared and then multiplied by A according to the value of the exponent. This reduces the number of multiplications needed from B to roughly 1.5 log B.

The modular exponentiation operation proceeds as follows. The registers are initialized with the values of the three operands, A (in r2), B (in r6), and N (in r10). A and N are first routed to the *red* unit and compared using the subtraction operation to check that $A < N$, as required by the implementation. Next, the Montgomery representation of A is computed as

$$A' = A \times 2^{\mathrm{L}} \bmod N,$$

where L is the bit length of the operands and k (see below) is the word size supported by the implementation, in this case, 65 bits. This is accomplished by a sequence of steps in which the Montgomery representation of 2^{L} is computed. This involves a call to the modular reduction operation implemented in the *red* unit to reduce the constant $2^{(\mathrm{L}+\mathrm{k})} \bmod N$ (from r10), and put the result in r4. Then a series of squaring of this value using mgmult implemented in the *mg* unit is applied to r4 with the result returned to r4 with a final mgmult of this result with 2^{k}. That result is then (mg) multiplied with A (still in r2) to get A', which is stored in both r2 and r4. Now the squaring and multiplications needed to raise A' to the power B commence, using another sequence of mgmult operations applied to the partial result in r4, and using the original value of A' in r2. The result of the exponentiation is converted back from Montgomery representation by a final mgmult with the value 1. This final result is written to r2, from where it is transferred to the DMA controller, which writes it to the bus fabric as output data.

Memory compression/decompression accelerator for AME

The AME technology uses the EFT engines to compress pages and to decompress them on-demand. EFT consumes or produces uncompressed data at a fixed rate of 8 bytes per cycle. At the 2.3 GHz fabric clock frequency, this results in a peak throughput of 37 GBps using two engines. EFT engines occupy a fraction of a percent of the POWER7+ chip area.

Lossless data compression research and utilities such as gzip and bzip2 focus mostly on efficient compression of large files to minimize storage capacity or channel utilization. Compressed memory systems, on the other hand, must provide short latency, use small blocks, use minimal silicon area, and perform an equal number of compress and decompress operations. Processors access memory much more frequently than storage and in small cache line size units. Compressed block sizes are usually chosen to be small to reduce average memory latency but at the expense of reduced compression efficiency. EFT is primarily used for compressing 4 KB pages, the OS's unit of memory management. To increase the system's effective memory capacity, a page of data should reside either in the compressed or the uncompressed regions of memory but not both. This results in an equal number of compress and decompress operations, unlike storage systems where decompression (read) operations are more frequent. The EFT design addresses the issues described above.

EFT is based on the 842 compression algorithm [14] related to the Lempel-Ziv (LZ) algorithms [15]. Compression is achieved by replacing 8-, 4-, or 2-byte phrases with pointers to the phrase copies recalled from a sliding window of input history as shown in **Figure 4**. Input is processed 8 bytes per cycle. Each 8-byte chunk is partitioned into seven sub-phrases, one 8 bytes, two 4 bytes, and four 2 bytes wide. Each size phrase uses a separate dynamic dictionary built from SRAM-based hash tables and FIFO buffers. Seven sub-phrases are searched in parallel in their respective dictionaries for pointers to their earlier copies in the sliding window. Each 8-byte chunk is then encoded with a 5-bit prefix called a "template" that describes the composition of the pointers and literals following it. Literals (LL) are 2-byte raw phrases from the current input chunk. Pointers point to 8-, 4-, and 2-byte phrases in the sliding window. For example, the four-bit template called P8 is followed by an 8-bit pointer to an 8-byte phrase in the dictionary, resulting in a total of 13 bits for encoding 8 bytes of input. In another example, the five-bit template P4_P2_LL is followed by a pointer to a 4-byte phrase, a pointer to a 2-byte phrase, and a 2-byte raw literal from the input. Considering all the permutations of pointers and literals and including the special templates, there are a total of 29 templates encoding every possible 8-byte input chunk, therefore requiring a 5-bit template. Z8 is a special template that

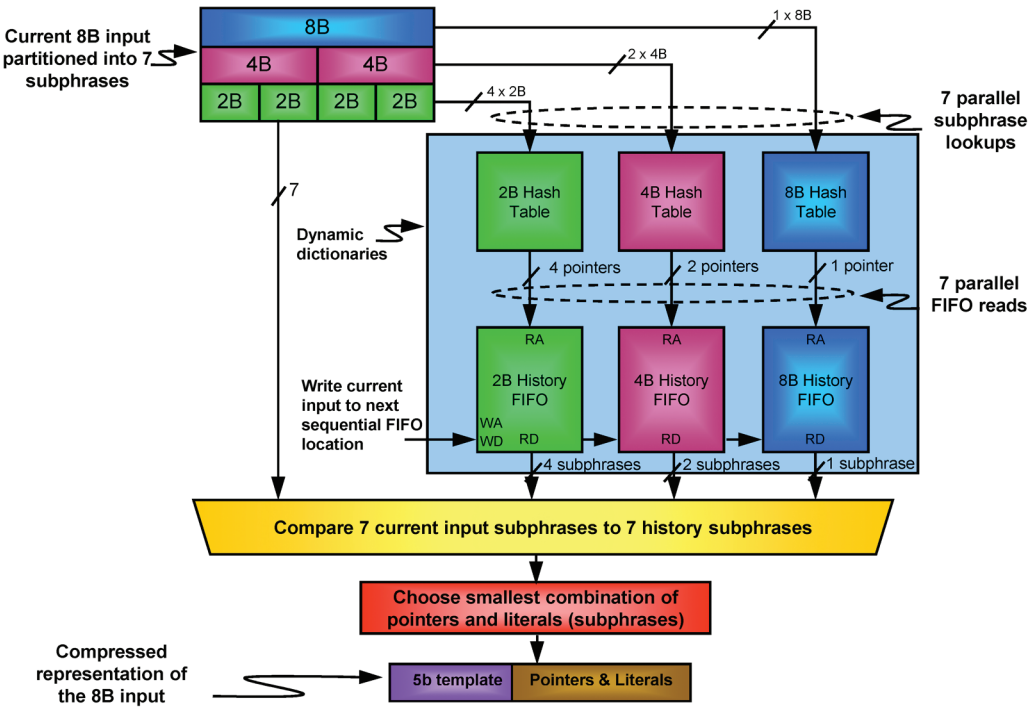

Figure 4

The 842 compression algorithm concept.

encodes 8 bytes of zeros, commonly found in memory. RPT(N) is a special template with a 6-bit argument N used for run length encoding, which encodes the previous 8 bytes N times. For example, a 512-byte run of zeros is encoded by the bit sequence "Z8, RPT(63)," resulting in $5 + 5 + 6 = 16$ bits.

Decompression involves reading the packed 64 bits of data from memory in each clock cycle, and decoding the 5-bit template field followed by its literals and pointers. Pointers de-reference the dictionaries to obtain 8-, 4-, and 2-byte phrases. Those phrases and are then combined according to the template to reconstitute the 8-byte output chunk. The seven sub-phrases of the chunk are added to their respective dictionaries as well.

The dictionaries are constructed with hash arrays and FIFO arrays, using high-speed SRAM macros. Many existing hardware LZ implementations use a content-addressable memory (CAM) as a dictionary, but CAMs typically have slower access time, low density, and high power consumption compared with SRAMs. In EFT, each size phrase has its own hash array and FIFO array as shown in Figure 4. The hash array contains pointers to the FIFO. The FIFO stores the input data to the next sequential location as it is received. Each phrase runs through a hash function to generate an address for the hash array. Each hash function is a carefully selected set of XOR trees that sample various bits of the phrase. The address is used to both read and write the hash array. The read occurs first and returns a pointer to the FIFO. The FIFO address where the input phrase will be stored is written into the hash array. If the same phrase shows up later it can access the same hash array location which returns a pointer to an earlier phrase copy in the FIFO. The input phrase and the phrase read from the FIFO are then compared for a match. If it is a match, the input phrase may be encoded in the output stream with a pointer.

The hash array is analogous to a direct-mapped cache, however, the hash array entries are replaced regardless of the match or mismatch status. The hash array can be expected to find fewer phrase matches than a CAM because of hash collisions and the fact that only a single phrase match is possible due to the direct-mapped-like organization. Sparse hash arrays are employed with about four times as many available entries as the number of used hash entries to reduce the probability of multiple phrases hashing to the same location. Set-associative hash arrays could provide higher hash hit rate and multiple phrase matches. However, these were judged to not improve compression ratio sufficiently to warrant the cost.

Processing of the seven sub-phrases requires 7-ported hash and FIFO arrays, which are implemented by employing

banked hash arrays and replicated FIFO arrays containing identical data. This approach gave an opportunity to optimize the compression encoder separately for each size phrase. Three different sliding windows for 8-, 4-, and 2-byte phrases are used, unlike in conventional LZ algorithms.

The 8-byte phrase uses a 1K $\times$ 8-bit hash array with one read port and one write port (1R1W). The 8-bit pointers allow a data FIFO size of 256 by 8 bytes. Data read from the FIFO is compared with the 8-byte phrase from the input. The input phrase is written into the FIFO location referenced by the next address counter. Once the FIFO is full, the counter rolls over and an earlier data is overwritten. This creates a 2K byte sliding window for the 8-byte phrases.

The 4-byte phrases share a 2K $\times$ 8-bit hash array with 2R2W ports. The 8-bit pointer values read from the hash array allows a FIFO size of 256 by 8 bytes, which stores 512 phrases of 4 bytes each, with two phrases per FIFO entry. A 9-bit pointer is needed to dereference one of 512 phrases. The low-order bit determines which half of the FIFO entry is used during encoding. If the input matches the high (low) 4 bytes of the FIFO, the low order bit is set to 0 (1). The FIFO operates as a 2K byte sliding window for the 4-byte phrases.

The 2-byte phrases share a 1K $\times$ 6 bit hash array. Ideally, this would use four read and write ports each, but SRAMs with four ports were not available. Therefore, the hash array is constructed using eight banks of 128 $\times$ 6-bit SRAMs with 2R2W ports. The high order three bits of the hash address select one of eight banks. If more than two hashes happen to address the same bank, the collision detection logic prevents all but two write accesses to the bank. The collision priority is fixed with the high order 2-byte phrase having the highest priority. During a collision, read data will be returned to the priority losers but from another phrase's access. In sum, write collisions are dropped and read collisions are returned with data from a wrong address. This does not result in a correctness problem as the hash array stored pointers are merely hints to where the phrases may be found in the FIFO. Only after the input phrase compares identical to the FIFO phrase, it is encoded with a pointer.

The 6-bit pointer stored in the hash array allows a 2-byte phrase FIFO size of 64 by 8 bytes with one write port and four read ports. The four read ports are implemented by duplicating arrays with two read ports each. The 64 by 8-byte SRAM allows for 256 phrases of 2 byte each. Therefore, 8-bit pointers are used for encoding 2 byte phrases. The two low order pointer bits are determined after the input data is compared to each 2-byte portion of the 8-byte FIFO data. If the 2-byte input matches the high 2 bytes of the FIFO, the low order pointer bits are "00." A match with the low 2 bytes of the FIFO gives low order pointer bits of "11." The FIFO operates as a 512-byte sliding window for the two byte phrases.

Decompression does not require the use of the hash arrays since pointers are already encoded in the compressed input data. The FIFO arrays are used as dictionaries for pointer references. The input stream is unpacked and decoded based on the 5-bit template field. Literals and FIFO data read using the pointers in the input stream are assembled back into an 8 byte output chunk. This chunk is also written back to each FIFO for future references. The pipelined nature of the design may result in FIFO read-after-write hazards, which are avoided by many levels of write data bypassing to forward data to the read stage.

Figure 5 shows the pipeline stages of the EFT engine. The compression and decompression pipelines are 64 bits wide, 18 and 16 stages deep, respectively. The pipelined data flows at the maximum rate without stalls, provided the DMA input/output buffers are not blocked. Pipeline hazards are avoided with bypass circuits forwarding data between stages, and by a compression-specific arbitration scheme to avoid port conflicts in arrays.

DMA Channel Input contains the bus protocol state machine and input buffers for both compression and decompression. The compression flow starts with *Compression Hash Function* for the XOR of phrase bits as described earlier. The function outputs are addresses of the 8-, 4-, and 2-byte *Hash Arrays*. *Hash Table Control* controls accesses to the *Hash Arrays*. Address decoding for bank selection and collision detection is performed here. Data read from the arrays are collected and then output as pointers. Counters generate pointers that are written into the hash arrays.

FIFO Input Control controls accesses to the *FIFO Array*. Pointers from hashing are used to read the FIFOs. Counters generate FIFO write addresses for input data. Error correction code generation on write data is done in this stage. Read and write addresses are decoded for bank selection. Staged input data and pointers are outputs from this stage. *FIFO Output Control* does error correction code checking and correction on read data. The read data is output from this stage.

Compression Data Compare compares phrases from input data with FIFO read data. *Compression Output Encode* uses comparison results to determine the template and result format. Pointers and literals are combined to create compressed data and a length field. Repeating input data is detected for special templates. The output is the template and result.

The *Output Data Packing* block takes the variable length template and result, up to 69 bits in length, and packs it into a 196 bit buffer. When 16 bytes of data are available, it is sent to the *DMA Channel Output*. This contains output buffers and the bus protocol state machines. This block also has the capability to stall the pipeline when the buffers are almost full.

The decompression flow starts with *Compressed Data Unpack* accepting 64-bit wide data from *DMA Channel Input*. The template field is used to determine the length of

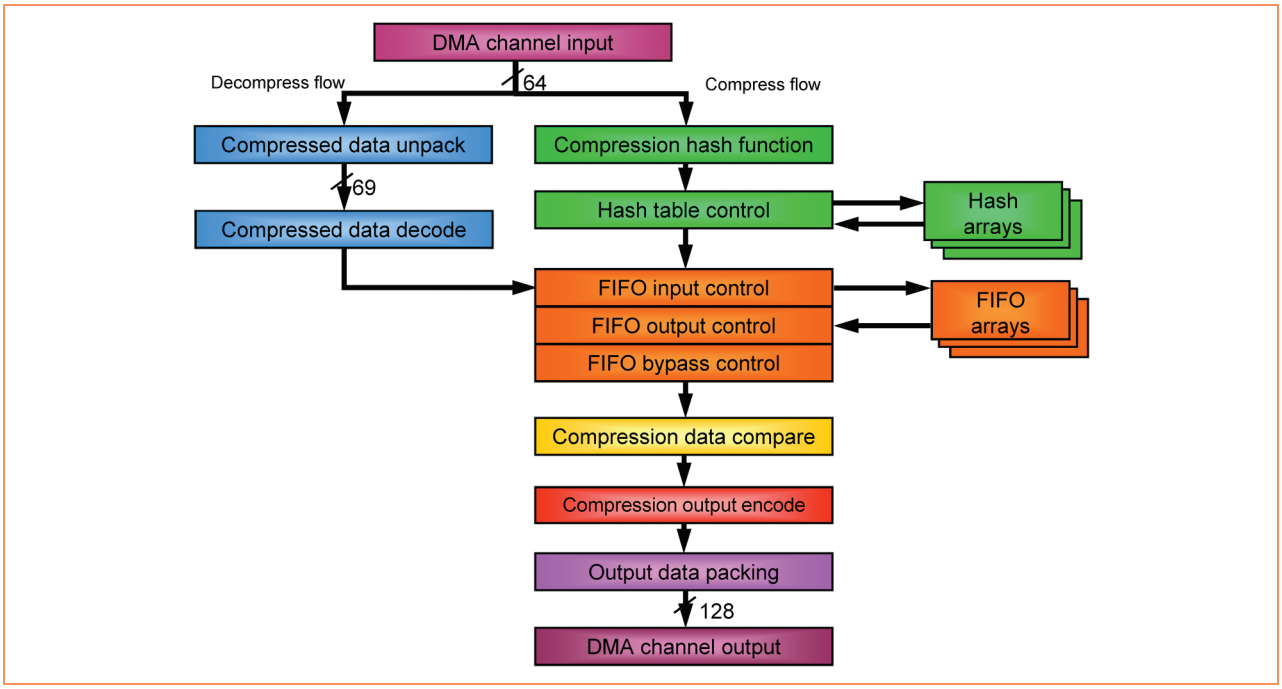

Figure 5

EFT pipeline dataflow.

the packed data. Using this length the buffers are shifted to pull out the variable length data. The output is a block of compressed data, including the template, which is a maximum of 69 bits long. The repeat function is handled here by sending the RPT(N) result N times to the next stage. *Compressed Data Decode* decodes the template and extracts the pointers and literals, which will go to the FIFO arrays.

FIFO Input Control uses pointers from data decode to read the *FIFO Array*. *FIFO Output Control* does error correction code checking and correction on read data. The read data goes to *FIFO Bypass Control*. Since FIFO writes happen many cycles later than reads, write bypass is required for decompression. The bypassed data and literals are assembled into decompressed data. This data is an output that also feeds back to *FIFO Input Control* for writing into the FIFO. A counter indicates the location to be written. ECC is generated on the write data.

The *Output Data Packing* block takes the decompressed data and packs it into a 196-bit buffer. When 16 bytes of data are available, it is sent to the *DMA Channel Output*. This contains output buffers and the bus protocol state machines. This block also has the capability to stall the pipeline when the buffers are almost full.

Performance

Accelerator performance was measured on test hardware using synthesized benchmarks with the objectives of finding maximum throughput of the accelerator complex and verifying there are no performance-limiting hardware issues. Several system factors affect accelerator complex performance and must be considered in the performance measurement process. As mentioned earlier, the hypervisor initiates an accelerator operation by executing an `invoke coprocessor` instruction on a core, and the hypervisor may then either poll the completion bit or receive an interrupt indicating completion. Because most operation types complete relatively quickly, and because of the overhead of an interrupt and task switch, polling is used for all operations except AMF operations. Also mentioned above is the fact that the accelerator complex has multiple engines and has the ability to queue operations. To measure the maximum throughput of the accelerator, therefore requires multiple outstanding operations. Performance measurements indicate that the complement of engine instances and queuing capabilities of the accelerator complex requires up to 12 operations to be in flight to saturate the implementation. This provides sufficient throughput with margin to meet the acceleration demand from the up to 32 threads on a POWER7+ chip.

The measured performance of the cryptographic and hash elements of the accelerator complex is shown in **Table 2**. Overall, the cryptographic accelerators met the performance objectives set at the outset of the project. Table 2(a) shows the throughput of AES operations for two common block

Table 2 Measured performance of cryptographic and hash functions.

(a)

GBps	*128-bit key*	*256-bit key*
1,504-byte block	3.25	2.57
4,096-byte block	3.25	2.57

(b)

GBps	*256-bit digest*	*512-bit digest*
256-byte block	1.06	1.38
4,096-byte block	1.41	2.17

(c)

	Operations/second
1,024-bit key	32,738
2,048-bit key	6,305

sizes and two different key sizes at a 2.3 GHz clock frequency. The maximum packet size of most Ethernet networks is 1504 bytes, and 4096 bytes is a commonly used page size. As the results show, the throughput is the same for both block sizes and approaches the peak performance expected for the operation. While 128-bit keys are common, there is movement toward larger keys. The results show only a modest decrease in throughput with the 256-bit key compared with the 128-bit key, which is as expected given the additional rounds per block necessitated by the larger key. The throughput of decryption is the same as that of encryption.

Table 2(b) shows the throughput of SHA hash operations. Again, results are shown for two block sizes and two digest sizes. Large-block throughput approaches peak engine throughput. For small block sizes, throughput is less than peak. These operations complete so quickly that it is not possible to queue enough operations to overlap all parts of the operations. With the larger block, the operation takes longer to complete, enabling more overlap. Although throughput is lower with the small block than with the large block, it is higher for both with the 512-bit digest than with the 256-bit digest. Although the 512-bit digest requires more rounds per block, the word width is double that of the 256-bit digest case, enabling higher throughput for the former.

Table 2(c) shows the rate at which AMF modular exponentiation operations can be executed for two different key sizes. Again encrypt and decrypt performance is the same. These operations do not encrypt and decrypt large amounts of data and are very long running operations, so throughput in terms of gigabytes per second is not appropriate. Because the operations run for such a long time, overheads essentially vanish, and measured operations per second closely approach peak performance of the engine. Although a 1024-bit key is common today, there is movement toward 2048-bit keys, with broader deployment in the future. The larger key requires more operations per bit, so throughput is reduced as expected.

The throughput of the EFT engines was also measured using 4096-byte blocks that were filled with data that had a compression ratio of 2.2, which is considered a typical compression ratio. For compression, a total bandwidth of 21.6 GBps was achieved. For decompression, a total bandwidth of 25.8 GBps was achieved. This is lower than the peak bandwidth of two engines of 37 GBps but matched expectations, given overheads and memory bandwidth limitations. A larger bandwidth was measured with larger blocks, but most usage of these engines is with 4096-byte pages. Measurements were also made with pages filled with zeros, which are common in some workloads. As expected, a lower bandwidth was measured because the amount of compressed data is very small. However, a higher operations per second rate was achieved.

Conclusion

This paper described the new hardware acceleration technologies included in the POWER7+ processor. The on-chip accelerator complex consists of acceleration engines as well as data movement and control hardware. The accelerator complex is attached to the processor SMP interconnect fabric, enabling access to data in the cache and memory subsystem at high bandwidth. The accelerators consist of engines that perform symmetric and asymmetric cryptographic functions, hashing, true random number generation, and memory compression and decompression. These are important operations in contemporary POWER system workloads. Movement of data between the bus fabric and the engines is accomplished by a BIU and DMA controller. These are endowed with read and write machine and buffer resources, as well as prefetch and overlap capabilities that allow memory access and job initiation latencies to be often hidden, thereby maximizing throughput. The algorithms and key sizes selected for acceleration in the cryptographic and hashing engines as well as the performance provided by all the engines, DMA controller, and BIU ensure their utility in the POWER7+ processor lifetime and in the years to come. Indeed, future POWER systems will see a broadening scope for acceleration technologies of various forms and functions, providing further CPU off-load and performance gains on algorithms suited to hardware acceleration.

References

1. D. Hepkin, *Active memory expansion—Overview and usage guide*, Feb. 2010. [Online]. Available: http://www-03.ibm.com/systems/power/hardware/whitepapers/am_exp.html

2. W. J. Armstrong, R. L. Arndt, D. C. Boutcher, R. G. Kovacs, D. Larson, K. A. Lucke, N. Nayar, and R. C. Swanberg, "Advanced virtualization capabilities of POWER5 systems," *IBM J. Res. & Dev.*, vol. 49, no. 4/5, pp. 523–532, Jul. 2005.
3. J. Rott, *Intel advanced encryption standard instructions (AES-NI)*, Feb. 2012. [Online]. Available: http://software.intel.com/en-us/articles/intel-advanced-encryption-standard-instructions-aes-ni
4. R. Golla and P. Jordan, "T4: A highly threaded server-on-a-chip with native support for heterogeneous computing," in *Proc. Hot Chips 23*, 2011, pp. 1–21. [Online]. Available: http://www.hotchips.org/wp-content/uploads/hc_archives/hc23/HC23.19.7-Server/HC23.19.731-T4-Golla-Oracle-hotchips_corrected.pdf
5. I. Dobos, W. Fries, P. Hamid, O. Lascu, G. Laumay, S. Ng, F. Nogal, F. Packheiser, V. Ranieri, Jr., K. Singh, A. Spahni, E. Ufacik, H. Wijngaard, and Z. Zhang, *IBM zEnterprise EC12 technical guide*, Dec. 2012. [Online]. Available: http://www.redbooks.ibm.com/abstracts/sg248049.html?Open
6. J. S. Liberty, A. Barrera, D. W. Boerstler, T. B. Chadwick, S. R. Cottier, H. P. Hofstee, J. A. Rosser, and M. L. Tsai, "A true hardware random number generation implemented in the 32-nm SOI POWER7+ processor," *IBM J. Res. & Dev.*, vol. 57, no. 6, Paper 4, pp. 4:1–4:7, 2013.
7. *Specification for the Advanced Encryption Standard (AES)*, FIPS Pub. 197, Nov. 2001. [Online]. Available: http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf
8. *Secure Hash Standard*, FIPS Pub. 180-1, Apr. 1995. [Online]. Available: http://www.itl.nist.gov/fipspubs/fip180-1.htm
9. *Secure Hash Standard (SHS)*, FIPS Pub 180-4, Mar. 2012. [Online]. Available: http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf
10. R. Rivest, "The MD5 message-digest algorithm," IETF, Fremont, CA, USA, RFC 1321. [Online]. Available: http://tools.ietf.org/html/rfc1321
11. H. Krawczyk, M. Bellare, and R. Canetti, "HMAC: Keyed-hashing for message authentication," IETF, Fremont, CA, USA, RFC 2104, Feb. 1997. [Online]. Available: http://www.ietf.org/rfc/rfc2104.txt
12. *Recommended Elliptic Curves for Federal Government Use,* NIST, Washington, DC, USA, July 1999. [Online]. Available: http://csrc.nist.gov/groups/ST/toolkit/documents/dss/NISTReCur.pdf
13. P. Montgomery, "Modular multiplication without trial division," *Math. Comput.*, vol. 44, no. 170, pp. 519–521, Apr. 1985.
14. P. A. Franaszek, L. A. Lastras-Montaño, S. Peng, and J. T. Robinson, "Data compression with restricted parsings," in *Proc. DCC*, 2006, pp. 203–212.
15. J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Trans. Inf. Theory*, vol. IT-23, no. 3, pp. 337–343, May 1977.

Received January 10, 2013; accepted for publication March 20, 2013

Bart Blaner *IBM Systems and Technology Group, Essex Junction, VT 05452 USA (blaner@us.ibm.com).* Mr. Blaner earned a B.S.E.E. degree from Clarkson University. He is a Senior Technical Staff Member in the POWER development team of the Systems and Technology Group. He joined IBM in 1984 and has held a variety of design and leadership positions in processor and ASIC development. In the recent past, he led the POWER7 fixed-point unit implementation and the POWER7+ accelerator complex design. He is presently focused on the architecture and implementation of hardware acceleration technologies spanning a variety of applications for future POWER processors. He is a Senior Member of the IEEE and holds more than 30 patents.

Bulent Abali *IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, NY 10598 USA (abali@us.ibm.com).* Dr. Abali is a Research Staff Member in the Systems Department. He works in the area of servers, storage, and systems software. His most recent work includes advanced memory architectures, data compression, processor and I/O virtualization, and fault tolerant systems. He has contributed to the hardware and software development of numerous IBM products for high-performance and commercial computing based on POWER and x86 microprocessors. He has authored more than 20 patents and 40 technical papers. He received a B.S. degree from Middle East Technical University, and M.S. and Ph.D. degrees from The Ohio State University in electrical engineering.

Brian M. Bass *IBM Systems and Technology Group, Research Triangle Park, NC 27709 USA (bass@us.ibm.com).* Mr. Bass earned his B.S. degree in engineering from the University of Alabama at Birmingham and his M.S. degree in electrical engineering from Clemson University. He joined IBM in 1984 and has worked in a variety of areas including token-ring LAN development, LAN switching, PowerNP network processor development, PowerEN accelerator architecture, and is now part of the POWER processor development organization. He is a Senior Technical Staff Member working in the area of POWER processor accelerator architecture, an IBM Master Inventor, and has filed more than 60 patents.

Suresh Chari *IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, NY 10598 USA (schari@us.ibm.com).* Dr. Chari is a Research Staff Member in the Security Department where he manages the Information Security Group. His research has addressed problems in applied cryptography, cryptography performance in software and hardware, systems security, web and application security, and cybersecurity. He has authored more than 30 technical papers in technical conferences and journals. He received a Ph.D. degree from Cornell University.

Ron Kalla *IBM Systems and Technology Group, Austin, TX 78758 USA (rkalla@us.ibm.com).* Mr. Kalla is the Chief Engineer for IBM POWER7 and future POWER processors. He has 29 years of processor design experience. He has worked on processors for IBM S/370, M68000, iSeries, and pSeries machines. He holds numerous patents on processor architecture. He also has an extensive background in post-silicon hardware bring-up and verification. He has 30 issued U.S. patents and is an IBM Master Inventor.

Steve Kunkel *IBM Systems and Technology Group, Rochester, MN 55901 USA (srkunkel@us.ibm.com).* Dr. Kunkel received his Ph.D. degree from the University of Wisconsin–Madison in 1987. He then joined IBM in Endicott, New York, doing performance analysis of a vector facility for a mid-range S/390 product. In 1989, he transferred to the IBM Rochester, Minnesota, site where he currently works. Spending most of his years in Rochester, he has done architecture and performance analysis for AS/400 and RS/6000 (now called POWER systems) products. This included such topics as NUMA, VLIW, caches, MP cache coherency, multithreading, MP interconnects, chip multiprocessors, and converting AS/400 to PowerPC architecture processors. Currently, he is a Senior Technical Staff Member and performance lead for POWER7+ and continues to do architecture and performance analysis for POWER systems servers.

Ken Lauricella *IBM Systems and Technology Group, Essex Junction, VT 05452 USA (cella@us.ibm.com).* Mr. Lauricella joined IBM in Kingston, New York, after earning a B.S.E.E. degree from Cornell University in 1979. He is a Senior Engineer in the POWER development team of the Systems and Technology Group. He has held a variety of design positions in processor and ASIC development. In the past few years, he has focused on the design and implementation of hardware accelerators.

Ross Leavens *IBM Systems and Technology Group, Research Triangle Park, NC 27709 USA (rossl@us.ibm.com).* Mr. Leavens has a B.S. degree in electrical and computer engineering from The Ohio State University (1987) and an M.S. degree in electrical engineering from

North Carolina State University (1989). He is a Senior Engineer in the POWER development team of the System Technology group. In 1989, he joined the IBM Networking Hardware Division in Research Triangle Park, North Carolina. He was a logic designer on several token-ring MAC and network processor chips. In 2003, he joined the Systems and Technology Group and has worked on several POWER processors in areas of I/O subsystem and cryptography acceleration. He has ten issued patents in network processor and microprocessor design.

John J. Reilly *IBM Systems and Technology Group, Essex Junction, VT 05452 USA (johnre@us.ibm.com).* Mr. Reilly earned a B.S. degree in electrical engineering from Pennsylvania State University. He is an Advisory Engineer in the IBM Systems and Technology Group. He has worked on design and verification of processors, including PowerPC 970, POWER7, POWER7+, and future POWER processors. Currently his focus is on hardware accelerators for compression and cryptography.

Peter A. Sandon *IBM Systems and Technology Group, Essex Junction, VT 05452 USA (sandon@us.ibm.com).* Dr. Sandon is a Senior Technical Staff Member in the Power Technology Development organization. He received a B.S. degree in electrical engineering from Cornell University, an M.S. degree in electrical engineering from the University of California at Berkeley, and a Ph.D. degree in computer science from the University of Wisconsin. He joined IBM in the Data Systems Division, Poughkeepsie, New York, to work on logic design and verification of the IBM 3081 mainframe. More recently, he has contributed to the design of a number of PowerPC and POWER processors in areas including architecture, performance, and formal verification. His current interests are in computer architecture and computer security.