VIA PadLock Programming Guide

3rd May 2005

This is Version 1.60 of the VIA PadLock Programming Guide

LIFE SUPPORT POLICY

VIA processor products are not authorized for use as components in life support or other medical devices or systems (hereinafter called life support devices) unless a specific written agreement pertaining to such intended use is executed between the manufacturer and an officer of VIA. 1. Life support devices are devices which (a) are intended for surgical implant into the body or (b) support or sustain life and whose failure to perform, when properly used in accordance with instructions for use provided in the labeling, can be reasonably expected to result in a significant injury to the user. 2. This policy covers any component of a life support device or system whose failure to perform can cause the failure of the life support device or system, or to affect its safety or effectiveness.

Contents

Chapter 1

Introduction to PadLock

The recent explosive growth of PC networking and Internet-based commerce has significantly increased the need for good computer and network security mechanisms. VIA addresses this need with Padlock Security Suite. This guide discusses the details of programming the PadLock features of your VIA processor. The whats, whys, and wherefors of PadLock and the underlying algrorithms are discussed in depth in the *VIA Security Application Note*, to which the reader is referred.

Features and Terminology

PadLock features have been expanded over the years, and not all VIA processors support the complete PadLock feature set. PadLock functions include:

- Random Number Generator (RNG)
- Advanced Cryptography Engine (ACE)
- Advanced Cryptography Engine (version 2) (ACE2), includes all original ACE capability
- Hash Engine (PHE)
- Montgomery Multiplier (PMM)

CPUID indentification for VIA processors that support PadLock

	Nehemiah (C5XL)	Nehemiah (C5P)	Ruth (C5Q)	Esther (C5J)
Supports	RNG	RNG, ACE	RNG, ACE2	RNG, ACE2, PHE, PMM
Vendor ID	Centaur Hauls	Centaur Hauls	Centaur Hauls	Centaur Hauls
Type	0	0	0	0
Family	6	6	6	6
Stepping	3	8-F	1-7	8-F

Enabling PadLock

To use PadLock, three conditions must be met:

1. PadLock must physically exist on the chip. Its existence can be discovered from the Centaur Extended CPUID Feature Flags. This condition is set as part of the manufacturing process and cannot be changed by software.
2. PadLock must be enabled and configured appropriately. The enabled state can be discovered from the Centaur Extended CPUID Feature Flags. PadLock can be enabled or disabled by writing to an MSR. The RESET default is enabled. But, see the next condition.

3. SSE instructions must be enabled via the standard x86 method of enabling the FXSAVE/FXRSTOR instructions using CR4[9] This enables the full set of SSE instructions. If CR4[9] is not set, PadLock behaves as if it were disabled via the MSR, regardless of the setting of the enable bits MSRs.

Note: This dependency on SSE is because PadLock datapaths just happens to be buried inside the SSE datapath. This is an implementation choice and future VIA processors may eliminate this dependency.

CPUID

If CPUID with EAX = 0xC0000000 returns EAX >= 0xC0000001 then Centaur Extended Feature Flags are supported. A CPUID with EAX = 0xC0000001 then returns the Centaur Extended Feature Flags in EDX. These bits in EDX refer to PadLock:

EDX[2] If this bit = 0, RNG does not exist on this chip. A RDMSR/WRMSR of MSR 0x110B will cause a General Protection Fault, and XSTORE instructions will always cause an Invalid Opcode Fault. If this bit = 1, RNG exists, and RDMSR/WRMSR of MSR 0x110B is allowed. The behavior of XSTORE is dependent upon whether RNG is enabled or not.

EDX[3] If this bit = 0, RNG is disabled. XSTORE instructions will cause an Invalid Opcode Fault. This bit reflects the result of setting bit 6 of MSR 0x110B. If this bit = 1, RNG is enabled (usually the RESET default), and XSTORE instructions can be used at any privilege level (provided that CR4[9] =1)

EDX[6] If this bit = 0, ACE does not exist on this chip. A RDMSR/WRMSR of MSR 0x1107 will cause a General Protection Fault, and REP XCRYPT instructions will always cause an Invalid Opcode Fault. If this bit = 1, ACE exists, and RDMSR/WRMSR of MSR 0x1107 is allowed. The behavior of REP XCRYPT instructions is dependent upon whether ACE is enabled or not.

EDX[7] If this bit = 0, ACE is disabled. REP XCRYPT instructions will cause an Invalid Opcode Fault. This bit reflects the result of setting bit 28 of MSR 0x1107 if ACE is present. If this bit = 1, ACE is enabled (usually the RESET default), and REP XCRYPT instructions can be used at any privilege level (provided that CR4[9] =1)

EDX[8] If this bit = 0, ACE2 does not exist on this chip. If this bit = 1, PadLock ACE2 exists, and the behavior of ACE2 REP XCRYPT instructions is dependent upon whether ACE2 is enabled or not. For Ruth and Esther processors, bit 8 is a copy of bit 6.

EDX[9] If this bit = 0, ACE2 is disabled. This bit reflects the result of setting bit 28 of MSR 0x1107 if ACE2 is present. If this bit = 1, PadLock ACE2 is enabled (usually the RESET default), and REP XCRYPT instructions can be used at any privilege level (provided that CR4[9] =1). For Ruth and Esther processors, bit 9 is a copy of bit 7.

EDX[10] If this bit = 0, PHE does not exist on this chip. REP XSHA instructions will always cause an Invalid Opcode Fault. If this bit = 1, PHE exists, and the behavior of REP XSHA instructions is dependent upon whether PHE is enabled or not.

EDX[11] If this bit = 0, PHE is disabled. REP XSHA instructions will cause an Invalid Opcode Fault. This bit reflects the result of setting bit 28 of MSR 0x1107 if PHE is present. If this bit = 1, PHE is enabled (usually the RESET default), and REP XSHA instructions can be used at any privilege level (provided that CR4[9] =1).

EDX[12] If this bit = 0, PMM does not exist on this chip. REP MONTMUL instructions will always cause an Invalid Opcode Fault. If this bit = 1, PMM exists, and the behavior of REP MONTMUL instructions is dependent upon whether PMM is enabled or not.

EDX[13] If this bit = 0, PMM is disabled. REP MONTMUL instructions will cause an Invalid Opcode Fault. This bit reflects the result of setting bit 28 of MSR 0x1107 if PMM is present. If this bit = 1, PMM is enabled (usually the RESET default), and REP MONTMUL instructions can be used at any privilege level (provided that CR4[9] =1).

MSR 0X110B

For all VIA processors, if RNG is present:, MSR 0x110B is initialized at RESET with the value: 0x00000040. This enables RNG (bit 6 = 1) and sets all control fields to 0. The meaning of bits in MSR 0x001B is discussed in detail in the RNG chapter of this guide.

MSR 0X1107

A Function Control Register (FCR), MSR 0x1107, which can be written by software to enable or disable various features, exists except for Nehemiah stepping 3. One bit controls PadLock:

FCR[28]=0 PadLock (except RNG) is disabled. If PadLock is present, a WRMSR can set this bit to 1 thus enabling ACE/ACE2/PMM/PHE.

FCR[28]=1 PadLock (except RNG) is enabled. A WRMSR can set this bit to 0 thus disabling ACE/ACE2/PMM/PHE.

Start-Up

The RESET signal is an immediate asynchronous process: RESET halts operation immediately regardless of the state of the processor. As part of the RESET process, the following PadLock actions are performed:

- Any in-progress PadLock instruction is immediately cancelled and undefined results may be stored.
- The FCR is reset to its default value. This may or may not directly change the enable status of PadLock features.
- The CR4 register is reset to zero. Since this resets the FXSAVE/FXRSTOR enable, (CR4[9]), PadLock features are indirectly disabled.
- EFLAGS:30 is cleared to zero. (See Chapter 3: Advanced Cryptography Engine)
- RNG is enabled, and the RNG hardware accumulation buffers are cleared and the accumulation count is set to zero. Random numbers start to accumulate overlapped with any other RESET or software function.

As opposed to RESET, the INIT signal causes no direct actions relative to RNG. That is, INIT does not internally cause a write to MSR 0x110B. The INIT signal causes CR4, however, to be reset. Since this resets the FXSAVE/FXRSTOR enable, CR4[9], RNG is indirectly disabled.

REP Prefixes and Exception Handling

During execution of a PadLock instruction, any number of exceptions or interrupts may occur.

For most exceptions, microcode instructions appear, logically, just like atomic x86 instructions: that is, the exception or interrupt occurs only before the instruction begins, or after it ends.

For normal x86 REP instructions, these exceptions occur naturally when the base instruction being repeated has completed, and after all architecture registers such as source and data pointers, and the REP counter have been updated. PadLock instructions behave the same way with respect to these controlled exceptions and interrupts.

However some exceptions occur in the middle of the microcode sequence. A good example is a page fault that occurs when loading a plaintext block from ES:[ESI] into the input_0 register of ACE. From the perspective of the REP XCRYPT instruction, the page fault occurs instantaneously. Execution branches to some fault handler, and eventually (well, almost always) the original REP XCRYPT instruction will be re-issued by the translator.

Thus the microcode must not change an architecturally visible register until it is guaranteed that no page fault or similar exception can occur, which would cause an incorrect state on instruction restart. This is handled by not changing the x86 registers until after a PadLock microcode store instruction has been issued and is waiting in the SSE queue. On VIA processors, that store instruction is guaranteed to complete, and so only then are the x86 visible registers updated to reflect the state of the operation at the completion of the store.

PMM requires special microcde to ensure that it is protected from page faults. This is handled by testing the memory buffers for all big integers read or written by the instruction before actually commencing calculations. This ensures that any page fault will occur before any architecture register is changed and before any data are written to memory.

Normal x86 REP instructions, when ECX == 0, are null operations. However, REP XCRYPTECB instructions with ECX == 0 can page fault. This is because the microcode does not check for a null REP count until after the first data blocks have been loaded, and those load operations could page fault. The advantage of starting the AES round engine immediately is that a single encryption operation runs some 3% to 4% faster than if ECX were tested first, because the AES round engine executes in parallel with the integer pipeline that executes the microcode.

Multi-Tasking

PadLock is specifically designed to address two major (and related) goals:

- To allow multiple applications (including the operating system) to use PadLock without any operating systems support (such as a device driver), and
- To allow multiple applications (including the operating system) to use PadLock without any awareness of, or visibility of, other tasks (and their data) that are also using PadLock.

RNG

The RNG is inherently multi-tasking safe, as there is no state with the exception of the global settings for the Random Number Generator, which may affect the rate at which random bits are delivered and the entropy of said bits. While Task A's bits are safe from inspection by Task B, the cryptographic value of these bits may be affected by changes to the RNG MSR. Readers are referred to the *VIA Security Application Note* for a discussion of security concerns and for appropriate levels of paranoia.

ACE

The ACE approach to satisfy these requirements is to make REP XCRYPT self-contained. That is, all of the information needed to perform an encryption is contained within REP XCRYPT instructions by using pointers in general purpose registers to all of the operands. These pointers are saved across task switches by existing operating systems.

A problem with this approach is that loading the extended key sequence into the hardware registers takes a longer time than performing the encryption operation itself. To improve performance, ACE has a mechanism to avoid reloading keys into the hardware unless they have changed.

For all REP XCRYPT instructions, bit 30 in EFLAGS specifies whether the processor needs to load the extended key sequence and the control word from memory into the hardware registers:

- If EFLAGS:30 is 0 when a REP XCRYPT instruction is executed, the control word and extended key sequence are loaded into the hardware registers. EFLAGS:30 is then set to 1.
- If EFLAGS:30 is 1 when a REP XCRYPT instruction is executed, the control word and extended key sequence in the hardware registers are presumed correct and are not reloaded. EFLAGS:30 is not changed.

In addition to this REP XCRYPT behavior, VIA processors implement changes to existing x86 instructions and operation that affect EFLAGS:

- EFLAGS:30 is set to 0 by any x86 instruction, interrupt, exception, task switch, etc. operation that causes EFLAGS to be stored (even if executed in 16-bit mode). Centaur Technology recommends using the instruction sequence PUSHFL; POPFL; prior to any REP XCRYPT instruction which uses a different key than the previous REP XCRYPT instruction.

- EFLAGS:30 cannot be set to 1 by any x86 instruction that causes EFLAGS to be loaded. Only REP XCRYPT instructions set this bit to 1.
- EFLAGS:30 is set to 0 by a SYSENTER instruction.

The effect of this mechanism relative to multitasking is (starting at the beginning of time):

1. Task 1 executes a first REP XCRYPT instruction. Since EFLAGS:30 is 0, the REP XCRYPT logic loads the control word and extended keys into the hardware, and sets EFLAGS:30 to 1.
2. Task 1 executes a second REP XCRYPT instruction. Since EFLAGS:30 is 1, the REP XCRYPT logic bypasses loading the control word and extended keys. This improves the performance of the second REP XCRYPT instruction.
3. A task switch to task 2 occurs. The process of loading the context of task 2 always involves saving task 1's EFLAGS and loading task 2's EFLAGS. The save of task 1's EFLAGS sets bit 30 to 0 in the saved value. The load of task 2's EFLAGS may be done by the operating system with an explicit POPF instruction, or it may be performed by the built-in x86 exception or task switch mechanism. Regardless, this load causes task 2's EFLAGS:30 to be set to 0.
4. Task 2 executes a REP XCRYPT instruction. Since its EFLAGS:30 is 0, the REP XCRYPT logic loads the control word and extended keys into the hardware, and sets EFLAGS:30 to 1.
5. A task switch back to task 1 occurs. The EFLAGS:30 for task 1 has bit 30 cleared to 0, so the next task 1 REP XCRYPT instruction will reload the task 1 control word and keys.

PHE

As there is no state information within PHE that is not preserved in x86 architecture registers or in the user program memory, REP XSHA instructions are safe for multi-tasking.

However, PHE instructions use portions of the ACE control word logic to distinguish between 160 bit (REP XSHA1) and 256 bit (REP XSHA256) hash functions. Because the ACE control word is written to by the microcode of PHE, REP XSHA instructions set the ACE EFLAGS:30 bit to zero to indicate that any subsequent REP XCRYPT instructions must reload the control word and encryption key.

Programmers who are both encrypting and hashing a message may find it more efficient to complete the entire hash function before encrypting any blocks of the message.

PMM

There is no internal state in PMM that needs to be preserved across task switches, or whenever an interrupt or exception is taken. All state necessary to re-start the instruction at the interrupted point is preserved in either the EAX register, or in the result buffer, where the partial product is stored.

Thus REP MONTMUL instructions neither read nor set the EFLAGS:30 bit used by ACE and PHE.. The instruction has no multi-tasking or security ramifications beyond those of any normal x86 instruction.

Chapter 2

Random Number Generator

2.1 XSTORE Instructions

XSTORE: Store Available Random Bytes

0x0F	0xA7	0xC0

	Input	Output
EAX		RNG Status Word
EDX	Quality Factor	
ES:[EDI]	Output Buffer	0-8 Random Bytes

REP XSTORE: Store ECX Random Bytes

0xF3	0x0F	0xA7	0xC0

	Input	Output
EAX		RNG Status Word
ECX	REP count	0
EDX	Quality Factor	
ES:[EDI]	Output Buffer	ECX Random Bytes

RNG Status Word: EAX

On completion of an XSTORE, MSR 0x110B[31:5] are copied to EAX[31:5], and the actual number of random bytes stored by the instruction (between zero and eight) is placed in EAX[4:0]. The number of bytes actually stored depends on the number of bytes available from the RNG hardware, and on the setting of the quality factor register EDX.

RNG Quality Factor: EDX

EDX specifies the rate at which the random bits are returned, and the corresponding level of randomness. Only the lower two bits of EDX are meaningful, and the upper 30 bits will be set to zero by the instruction.

The rules for the number of bytes stored by an XSTORE instruction:

- The instruction tests whether a hardware accumulation buffer is full. If fewer than 8 bytes are available, the instruction performs no store, and returns with a zero byte count in the status word.
- If at least 8 bytes are available in the hardware buffers, the appropriate bits are selected and either one or two four-byte stores are performed as required.

- Note there may be less than four valid random bytes stored depending on EDX. In this case, the remaining bytes of the store contain zeroes. The byte count in EAX and the update to EDI are based on the number of random bytes stored, not on the size of the stores themselves.

EDX[1:0]	Bytes available	Total Bytes Stored	Random Bytes Stored	EAX[4:0] & EDI inc
N/A	less than 8	0	0	0
00	8 or more	8	8	8
01	8 or more	4	4	4
10	8 or more	4	2	2
11	8 or more	4	1	1

REP Count: ECX

When an XSTORE instruction is prefixed with REP, the ECX value defines the total number of random bytes to be stored. Four-byte stores (zero, one, or two, as required) are performed until the requested total number of bytes have been stored, and ECX and EDI are updated automatically such that random bytes are concatenated into a contiguous string. REP XSTORE processing continues until the desired quantity of random bytes is stored.

The implication of storing fewer random bytes than the size of the store is that unaligned four-byte stores may be performed. These are slower than aligned stores and will cause an alignment check if it is enabled. However, the asynchronous nature of the random number generator means that any alignment checks taken during a REP XSTORE will overlap with the generation of new bits and there will be no real-world performance penalty.

Another implication is that on the last iteration, REP XSTORE may store up to seven bytes to memory beyond the last random byte requested. Adequate memory must be allocated based on the EDX value to contain the last one or two four-byte stores.

RNG Buffer Pointer: EDI

EDI (or DI) contains the effective address (offset) where the random bytes generated by the instruction are stored. This address is always relative to the segment specified in ES, and any segment prefix on the instruction will be ignored.

At the end of instruction execution, or if an interrupt (or page fault, etc.) has occurred, EDI (or DI) is incremented by the number of random bytes stored. The address is always incremented: EFLAGS.DF has no effect. Whether DI or EDI is used and updated is based on the effective address size for the executed instruction.

Either zero, one, two, four or eight bytes are stored to memory depending on the data rate specified by EDX and the status of the RNG hardware. Software must always provide an appropriately sized write-enabled buffer at the ES:EDI destination address.

All RNG stores are four-bytes in length. Thus, all memory-operand restrictions and exceptions that are applicable to a four-byte store are also applicable to the XSTORE instruction. If alignment check is enabled and is to be avoided, then the address in the starting ES:EDI should be 4-byte aligned. Similarly, segment limit checks, protection check, page faults, etc. behave as if the data size is four bytes.

REP Prefix

- The REPNE prefix is undefined.
- The specification of a 16-bit mode in CS is ignored; the full 32-bit EAX is always updated.
- The DF (direction flag) in EFLAGS has no effect: instruction execution always adds the number of random bytes stored to EDI.

The REP XSTORE execution is interruptible. When an exception or interrupt occurs and is taken, or the instruction completes:

- ES:EDI is updated to address the next location for storing random byes

- ECX is updated to contain the number of random bytes remaining to be stored, and
- EAX contains a copy of the current status word. However, the byte count field (4:0) in the status word is not defined for the REP case.

Code that runs during an interrupt could, in theory, modify the RNG configuration. In (unusual) environments where this may occur, application programmers should be aware that the RNG status reported in EAX at the end of the operation may not reflect the configuration at the beginning of the operation. If it is important to make sure that the RNG configuration does not change, XSTORE should be used, not REP XSTORE.

The same rules apply as for a non-REP XSTORE for the REPNE prefix, the AS prefix, the OS prefix, the size for alignment purposes, the size for other exceptions, etc.

Note: There is a theoretical problem with using the REP XSTORE instruction because (1) the instruction does not return control until all requested bytes are stored, and (2) the generation bit rate is variable. The REP version of XSTORE prevents easy detection of a situation where the RNG hardware is not returning any bytes. In practice, this not a real risk, since this situation cannot occur without a catastrophic hardware failure. Applications can address this possibility by performing a startup test.

2.2 Configuring the Random Number Generator

MSR 0x110B provides both status and control of RNG. In addition, a copy of the MSR contents as they are at the start of execution of an XSTORE instruction is returned in EAX.

The EDX portion of this MSR is undefined. The EAX portion of the MSR contains:

31:22	21:16	15	14	13	12:10	9:8	7	6	5	4:0
RESV	FCNT*	FAIL*	FLTR*	RBTS	BIAS	SRCE	RESV	ENBL	RESV	BCNT

* Reserved (RESV) on Esther processors.
Reserved bits (RESV) have undefined and unpredictable values.

In the following detailed description, (O) means a read-only field output by the processor, and (I) means the field can be written by software. The phrase "when the MSR is read" means when it is read by a RDMSR instruction or when it is copied into EAX as part of an XSTORE instruction execution.

When writing the RNG MSR, all reserved bits should be set to zero. It is not possible to improperly configure current versions of RNG, but subsequent versions may define these reserved bits and software that incorrectly detects the processor stepping may improperly configure these later steppings.

Any write to MSR 0x110B causes a load reset of the RNG hardware. On load reset, the internal hardware buffers are cleared to zero and the options defined by the MSR are set to the values written.

The detailed field information:

BCNT Bits[4:0] Current Random Byte Count (O): When this field appears in the copy of the MSR placed in EAX by the execution of XSTORE, the number is the exact number of bytes actually stored.

For Nehemiah and Ruth processors: When the MSR is read, this read-only field contains the exact number of random bytes that are currently available for storing. Due to the asynchronous generation of random bytes, and the fact that multiple programs may be storing RNG data, the number that appears may not be the same as the number stored on the next execution of XSTORE.

For Esther processors: When the MSR is read, this read-only field reports the number of eight-byte accumulation buffers currently full. Each bit in the field represents a single buffer.

0x00 0 buffers available

0x01 1 buffer available

0x03 2 buffers available

0x07 3 buffers available

0x0F 4 buffers available

ENBL Bit[6] RNG Enable (I): When set to 1, this bit enables RNG. When set to 0, the XSTORE instruction becomes invalid and the random number generator is internally disabled to reduce power consumption. The extended CPUID feature flag that indicates whether or not RNG is enabled (EDX[3]) is a copy of this bit.

This enable bit may be set internally by the RESET process. In this case, the effect on the processor is the same as if a write MSR set the enable bit.

SRCE Bits[9:8] Noise Source Select (I): These bits control the two noise sources on the processor that input bits to the accumulation buffers. On Nehemiah processors prior to stepping 8, these bits are reserved and undefined. The default RESET state is both bits = 0.

00 Noise source "A" active

01 Noise source "B" active

1x Both noise sources active

BIAS Bits[12:10] DC Bias (I): These bits control the DC bias supplied to the random number generator. These bits may affect the speed of the generator and the randomness of the generated bits. Users who experiment with RNG are advised not to expect that settings appropriate for any one VIA processor will provide comparable behavior on another VIA processor.

The DC bias adjustment is intended for Centaur Technology's internal analysis, test, and debug. However, our philosophy is to provide all existing functions to software in case there is some external value in them. Following that philosophy, we make the DC bias control available.

RNG performs well with the default DC bias setting (0). It is possible, however, that alternate DC bias settings may improve bit-generation speed or randomness characteristics. The sophisticated or curious user may want to experiment with changing the DC bias. This requires protected mode access for writing the RNG MSR.

For those users interested in optimizing the DC bias, Centaur Technology may be able to provide some advice or assistance based on our testing experience.

RBTS Bit[13] Raw Bits Enabled (I): If this bit is set to 1, the von Neumann compressor (or whitener) in the hardware generator is disabled and the raw bits produced by the random generator are delivered to the accumulation buffers. A 0 in this bit selects the whitener function.

FLTR Bit[14] String Filter Enable (I): If set to 1, this enables a test feature that filters out strings of contiguous ones or zeroes longer than the value set in bits 21:16. Note that enabling the string filter may introduce non-uniform statistical characteristics in the output. The Esther processor does not have the string filter, and this bit is instead reserved (RESV).

FAIL Bit[15] String Filter Fail (I/O): This bit indicates that while the string filter was enabled, the hardware detected a string of contiguous ones or zeroes longer than the value defined by the filter count in bits 21:16. Only the hardware can set this bit to a 1, but a MSR write can set it to 0. The Esther processor does not have the string filter, and this bit is instead reserved (RESV).

FCNT Bits[21:16] String Filter Count (I): The value in this field defines the bit-string length (8 - 63) to be used by the string filter. Bit string lengths less than 8 will produce unexpected results and should not be used. The Esther processor does not have the string filter, and these bits are instead reserved (RESV).

2.3 Performance and Timing

The effective rate of generating random bits is variable and depends upon the processor's chip voltage, its temperature, process variations, phase of the moon, and so forth.

Note: sample size for the examples below is 10MB

For a VIA Nehemiah processor (stepping 8), with both noise generators selected, our data show that bits are delivered to the accumulation buffers at about 80 Mb/s to 120 Mb/s.

With the whitener selected, the rate falls to between 12 Mb/s and 20 Mb/s.

With only one noise source selected, or for stepping 3 of the VIA Nehemiah processor, the raw bit generation will be between 40 Mbs and 60 Mbs, and the whitened rates between 6 Mbs and 10 Mbs.

For a VIA Esther processor, with both noise generators selected, we have observed bit rates as high as 320 Mb/s and as low as 65 Mb/s (on different parts), depending on the DC bias setting. At the default DC bias setting the bit rate should be about 150-200 Mb/s.

With the whitener selected, the rate falls to between about 15 Mb/s and 80 Mb/s.

When both noise sources are active, bits accumulate in the buffers from both sources. As the noise sources are independent, there is some reduction in the bit-to-bit correlation but the effect is small. Use of both noise sources is recommended for faster generation of random bits, but should not be used to improve the entropy of the raw bits or the statistical qualities of the resulting random number stream.

The XSTORE instruction is partly implemented in microcode. The number of execution clocks depends on whether random data are available, and on the value in EDX (the data rate register). The XSTORE instruction does not return any random data unless an eight-byte accumulation buffer is full. If fewer than 8 bytes are available, the instruction returns status for the RNG with a zero byte count. When an accumulation buffer is ready, XSTORE returns the bits to memory as pointed to by ES:EDI and in the quantity specified by EDX. The status information in EAX correctly indicates the actual number of bytes stored.

The following table gives estimates for the number of clocks required for the XSTORE instruction. Note that a 2 GHZ Esther processor, with both of the noise sources active and with the whitener OFF, requires an average of about 500 clocks to fill a single 8-byte accumulation buffer.

XSTORE Execution clocks

EDX	Data Ready	Random Bytes Stored	Execution Clocks
0-3	No	0	~30
0	Yes	8	~55
1	Yes	4	~310
2	Yes	2	~200
3	Yes	1	~120

REP XSTORE is executed as an internal microcode loop of individual XSTORE instructions until the specified number of bytes has been stored. If the requested count can be satisfied with a single internal XSTORE, and enough bits are in the hardware buffers, add 7 clocks.

For the more typical use of collecting a large number of bytes, the speed of REP XSTORE will be the bit generation speed plus a few hundred clocks.

RNG bit-generation hardware is clock-enabled when the MSR enable bit is set. In addition, the data path associated with accumulating bits, counting bytes, etc. is clock-enabled. Thus, dynamic power starts being consumed by RNG when it is enabled (normally by default at RESET).

Whenever all four 8-byte accumulation buffers are full, the random number bit generator clocks are temporarily disabled. When a buffer location becomes free, the generator is re-enabled. The associated RNG data path clocks are never disabled, however, until the MSR enable is cleared. Even though data path clocks are not disabled, the power consumption of this logic is very small.

Relative to the SSE unit power management, the XSTORE instruction behaves like SSE instructions. That is, when the XSTORE instruction is seen by the execution unit, the entire SSE unit is enabled until no further SSE instructions or XSTORE instructions are seen, at which point the SSE unit is disabled

Chapter 3

Advanced Cryptography Engine

PadLock Advanced Cryptography Engine (ACE) implements the Advanced Encryption Standard (AES) algorithm. The AES is a modern symmetric key encryption algorithm intended to replace the older Digital Encryption Standard, and has been selected by the National Institute of Standards and Technology (NIST) as the approved symmetric key algorithm for United States government use. Use of AES has been approved by the National Security Agency (NSA) for SECRET and TOP SECRET communications.

3.1 REP XCRYPT Instructions

REP XCRYPTECB: Electronic Code Book

0xF3	0x0F	0xA7	0xC8

	Input	Output
ECX	REP count	0
ES:[EDX]	ACE Control Word	
ES:[EBX]	Encryption Key	
ES:[ESI]	Encrypt:Plaintext / Decrypt:Ciphertext	
ES:[EDI]		Encrypt:Ciphertext / Decrypt:Plaintext

In this mode, each data block is encrypted independently from other blocks. Two blocks are pipelined through the round engine. Each data block is initially XOR'd with the first key block and the result feeds into the round engine. The result of each encryption round is feed back into the core for the next round. When all rounds have been completed, the final result is captured in one of the two output registers.

REP XCRYPTCBC: Cipher Block Chaining

0xF3	0x0F	0xA7	0xD0

	Input	Output
ECX	REP Count	0
ES:[EAX]	Initialization Vector	
ES:[EDX]	ACE Control Word	
ES:[EBX]	Encryption Key	
ES:[ESI]	Encrypt:Plaintext / Decrypt:Ciphertext	
ES:[EDI]		Encrypt:Ciphertext / Decrypt:Plaintext

In this mode, the result of each encryption is XOR'd with the next incoming data block. This cipher block chaining (forwarding) operation insures that any single block of cipher text cannot be decrypted to plain text; only the entire file can be decrypted. This forwarding means that only one block is encrypted at a time; no pipelining of the round engine occurs. On the first round, an Initialization Vector (IV) provided by the application is used in place of the forwarded cipher text.

REP XCRYPTCTR: Counter Mode

0xF3	0x0F	0xA7	0xD8

	Input	Output
ECX	REP Count	0
ES:[EAX]	Initialization Vector	
ES:[EDX]	ACE Control Word	
ES:[EBX]	Encryption Key	
ES:[ESI]	Encrypt:Plaintext / Decrypt:Ciphertext	
ES:[EDI]		Encrypt:Ciphertext / Decrypt:Plaintext

In this mode, the Initialization Vector is encrypted, and the result is XOR'd with the plaintext to produce the ciphertext, or with the ciphertext to reconstruct the plaintext. However, no data is forwarded to the next round.

CTR mode can be thought of as generating a one-time-pad. It is critical that no 128-bit block of the encrypted sequence be re-used. In CTR mode, this is accomplished by incrementing the CTR, or more accurately a specific bit-field of the CTR.

There are many possible counter modes. ACE implements a counter where the first 14 bytes of the counter (in normal ascending address order) are the nonce, a random value specific to the encryption of a particular message and which should never be re-used. The last two bytes are considered to be the 16-bit integer counter in big-endian format.

This counter mode is compatible with the CCM mode as specified by wireless protocol 802.11i and was chosen specifically by Centaur Technology and VIA to be compatible with that protocol.

REP XCRYPTCFB: Cipher Feedback Mode

0xF3	0x0F	0xA7	0xE0

	Input	Output
ECX	REP Count	0
ES:[EAX]	Initialization Vector	
ES:[EDX]	ACE Control Word	
ES:[EBX]	Encryption Key	
ES:[ESI]	Encrypt:Plaintext / Decrypt:Ciphertext	
ES:[EDI]		Encrypt:Ciphertext / Decrypt:Plaintext

In this mode, the Initialization Vector (IV) is encrypted and the result is XOR'd with the incoming plaintext (ciphertext) block. This is the ciphertext (plaintext) result, which is forwarded as the IV for the next source block. This block chaining (forwarding) operation insures that any single block of cipher text cannot be decrypted to plain text; only the entire file can be decrypted. This forwarding means that only one block is encrypted at a time; no pipelining of the round engine occurs. On the first round, an Initialization Vector (IV) provided by the application is used in place of the forwarded cipher text.

REP XCRYPTOFB: Output Feedback Mode

0xF3	0x0F	0xA7	0xE8

	Input	Output
ECX	REP Count	0
ES:[EAX]	Initialization Vector	
ES:[EDX]	ACE Control Word	
ES:[EBX]	Encryption Key	
ES:[ESI]	Encrypt:Plaintext / Decrypt:Ciphertext	
ES:[EDI]		Encrypt:Ciphertext / Decrypt:Plaintext

In this mode, the Initialization Vector is encrypted and the result is forwarded to the next round for further encryption, yielding a sequence of 128-bit blocks. This sequence is XOR'd with the plaintext to produce the ciphertext, or with the ciphertext to reconstruct the plaintext.

OFB mode is often used to pre-calculate a byte stream to XOR with a message as the encryption of the IV and the XOR with the plaintext can be separated in time. Thus OFB when used correctly is like a one time pad. It is thus critical that no 128-bit block of the encrypted sequence be re-used.

Memory Formats

Most ACE data loads and stores must be 16-byte aligned. Relative to AES notation, byte 0 is at the lowest address, byte 1 is at the next higher address, and so forth. That is, data is organized in the normal x86 little endian fashion such that AES byte 0 is the normal x86 architecture byte 0 (the linear address provided by an instruction operand).

For Nehemiah processors, all loads and stores must be 16-byte aligned. Esther and Ruth processors have a new algorithm (defined in bit 5 of the control word) that performs AES on plaintext and ciphertext with arbitrary alignment. This new algorithm is supported by all AES operating modes except ECB.

Performance of the new algorithm will still of course be best if the plaintext and ciphertext are nevertheless aligned on a 16-byte boundary. However, when the plaintext and ciphertext buffers are known to be aligned on a 16-byte boundary, the original aligned-only algorithm is preferred, as the performance is significantly better. On future VIA processors the performance difference may be reduced, but in all cases (where the plaintext and ciphertext buffers are aligned on 16-byte boundaries) the aligned-only algorithm will be faster.

Input (Plaintext) Data Pointer: ESI

ESI (or SI) in all REP XCRYPT instructions contains the effective address (offset) of the input data. This address is always relative to the segment specified in ES. If Control Word bit 5 = 0, the resulting linear address must be aligned on a 128-bit boundary, otherwise a General Protection exception occurs. If Control Word bit 5 = 1 this alignment requirement is relaxed except for ECB mode. It is possible to encrypt or decrypt a buffer directly, i.e. initially setting ESI == EDI is valid for all ACE modes, although it is not recommended for message digests.

Non 16-byte aligned data (Control Word bit 5 = 1) is not supported on Nehemiah processors.

For encryption operations, ESI (or SI) points to the plaintext. For decryption operations, ESI (or SI) points to the ciphertext.

At the end of instruction execution, or if an interrupt (or page fault, etc.) has occurred, ESI (or SI) is incremented by the number of bytes stored. The address is always incremented: EFLAGS.DF has no effect. Whether SI or ESI is used and updated is based on the effective address size for the executed instruction.

Output (Ciphertext) Data Pointer: EDI

EDI (or DI) in all REP XCRYPT instructions contains the effective address (offset) of the result data generated by the instruction. This address is always relative to the segment specified in ES. If the Control Word bit 5 = 0, the resulting linear address must be aligned on a 128-bit boundary, otherwise a General Protection exception occurs. If Control Word bit 5 = 1 this alignment requirement is relaxed except for ECB mode. It is possible to encrypt or decrypt a buffer directly, i.e. initially setting ESI == EDI is valid for all ACE modes, although it is not recommended for message digests.

Non 16-byte aligned data (Control Word bit 5 = 1) is not supported on Nehemiah processors.

For encryption operations, EDI (or DI) points to the ciphertext. For decryption operations, EDI (or DI) points to the plaintext.

At the end of instruction execution, or if an interrupt (or page fault, etc.) has occurred, EDI (or DI) is incremented by the number of bytes stored. The address is always incremented: EFLAGS.DF has no effect. Whether DI or EDI is used and updated is based on the effective address size for the executed instruction.

EXCEPTION: When calculating a message digest (ControlWord[4] = 1), and for CBC and CFB operating modes only, the Ciphertext Pointer in EDI (or DI) is never incremented.

Encryption Key Pointer: EBX

EBX (or BX) in all REP XCRYPT instructions contains the effective address (offset) of the key data. This address is always relative to the segment specified in ES. The resulting linear address must be aligned on a 128-bit boundary, otherwise a General Protection exception occurs. Whether BX or EBX is used is based on the effective address size for the executed instruction.

The key data takes one of two forms depending on bit 7 of the control word. If Bit[7]=0, EBX points to the primary key and the extended key sequence will be generated by the hardware. If Bit[7]=1, EBX points to the primary key followed by the application calculated extended key sequence, which is loaded in its entirety into the key ram.

Number of (128-bit) Blocks to Encrypt: ECX

ECX (or CX) in all REP XCRYPT instructions contains the count of 128-bit data blocks to be processed.

At the end of instruction execution, ECX (or CX) will be zero. If an interrupt (or page fault, etc.) has occurred, CX or ECX will be decremented by the number of blocks processed and stored by the instruction prior to the interrupt. Whether CX or ECX is used and updated is based on the effective address size for the executed instruction.

Initialization Vector Pointer: EAX

AX or EAX in chaining mode REP XCRYPT instructions (all modes except ECB) contains the effective address (offset) of the initialization vector (IV). In Counter (CTR) mode, the Initialization Vector is more accurately called the counter. This address is always relative to the segment specified in ES. The resulting linear address must be aligned on a 128-bit boundary, otherwise a General Protection exception occurs. The choice of whether AX or EAX is used is based on the effective address size for the executed instruction.

As for normal x86 string instructions, if an external interrupt occurs during the processing of the REP XCRYPT instruction, the registers are updated before the interrupt transition occurs such that the REP instruction may be restarted upon return from interrupt handling. The initialization vector presents a problem since the original initialization vector is not the right one to use upon instruction restart. So the REP XCRYPT logic changes either the address in AX/EAX, or changes the data value pointed to by AX/EAX to be the appropriate initialization vector for the subsequent iteration.

Thus, an application program must not assume that EAX is unchanged, or that the value pointed to by EAX is unchanged, after the execution of a chaining mode REP XCRYPT instruction. For OFB mode, and for decryption using CBC and CFB modes, the value pointed to by EAX will be changed. For encryption using CBC and CFB modes, the value in EAX itself is changed

For CTR mode, the high-order 16 bits of the value pointed to by EAX will be incremented by one for every block processed. Note also that this 16 bit counter is big-endian, compatible with the counter mode defined for Wireless specification 801.11i

REP Prefix

XCRYPT instructions must be used with the REP prefix.

- The value in EDI is updated with the number of result bytes stored (except for CBC-MAC and CFB-MAC modes, when EDI is unchanged).
- An address-size prefix affects the size of EAX, EBX, EDX, ESI, and EDI used
- The usual address exceptions (past the segment limit, page fault, etc.) can occur
- The value in ESI is updated with the number of result bytes stored. Note that in ECB mode with an initial odd value in ECX, one more data block from ESI is processed than is stored.
- A REPNE prefix causes undefined behavior.
- An operand size prefix causes an Invalid Instruction exception.
- The DF (direction flag) in EFLAGS has no effect. The operation always proceeds from low address to high address.

3.2 Configuring ACE

Extended Keys

A REP XCRYPT instruction always points to the encryption key. However, AES encryption requires an extended key sequence. This sequence provides a unique 128-bit value for each round of the algorithm. ACE supports two methods for providing these extended keys:

- Hardware Generated. The hardware can generate the extended key sequence for both encryption and decryption for 128-bit keys, for all specifiable round counts. This generation is performed as a side effect of a REP XCRYPT instruction that points to the base key and points to a control word that specifies hardware-generated extended keys. This option is the normal approach for 128-bit keys.
- Application Provided. The application can provide the entire extended key sequence instead of letting the hardware generate it. The load of this extended-key data is performed as a side effect of a REP XCRYPT instruction that points to the base key followed by the extended keys and points to a control word that specifies application-provided extended keys. This is the only supported option for 192-bit and 256-bit keys.

For hardware generated keys, the key data pointed to by EBX must start with byte 0 of the key (w[0] in AES notation) on a 16-byte aligned linear address, followed by byte 1 (w[1] in AES notation), up to byte 15.

For application-loaded keys for encryption, the extended key sequence starts with the normal key as described above followed immediately by the first byte of the extended key sequence, followed by the second byte, etc. for the entire sequence of extended keys. That is, the memory sequence w[0], w[1], etc. is as described in Figure 11 of FIPS-197.

For application-loaded keys for decryption, the equivalent inverse cipher extended keys must be provided in the inverse order (the real key comes last) as described in Figure 15 of FIPS-197. That is, the first addressed byte is defined in FIPS-197 Figure 15 as dw[0], the next is dw[1], and so forth.

For the application-loaded keys option, the hardware always loads sixteen 128-bit values from memory regardless of the specified key size. Any values beyond the normal extended key size are ignored and have no affect on the results, but that memory area must be accessible (within the segment limit, etc.)

NOTE: For calculation of intermediate results for decryption, for all key sizes, it is necessary for the application to load the extended key schedule.

EFLAGS[30]

For all REP XCRYPT instructions, bit 30 in EFLAGS specifies whether the processor needs to load the extended key sequence and the control word from memory into the hardware registers:

- If EFLAGS:30 is 0 when a REP XCRYPT instruction is executed, the control word and extended key sequence are loaded into the hardware registers. EFLAGS:30 is then set to 1.
- If EFLAGS:30 is 1 when a REP XCRYPT instruction is executed, the control word and extended key sequence in the hardware registers are presumed correct and are not reloaded. EFLAGS:30 is not changed.

In addition to this REP XCRYPT behavior, VIA processors implement changes to existing x86 instructions and operation that affect EFLAGS:

- EFLAGS:30 is set to 0 by any x86 instruction, interrupt, exception, task switch, etc. operation that causes EFLAGS to be stored (even if executed in 16-bit mode). Centaur Technology recommends using the instruction sequence PUSHFL; POPFL; prior to any REP XCRYPT instruction which uses a different key than the previous REP XCRYPT instruction.
- EFLAGS:30 cannot be set to 1 by any x86 instruction that causes EFLAGS to be loaded. Only REP XCRYPT instructions set this bit to 1.
- EFLAGS:30 is set to 0 by a SYSENTER instruction.

Control Word Pointer

EDX (or DX) in all REP XCRYPT instructions contains the effective address of the ACE control word. This address is always relative to segment ES. The resulting linear address must be aligned on a 128-bit boundary, otherwise a General Protection exception occurs.

Whether DX or EDX is used is based on the effective address size for the executed instruction.

When the new 16-byte alignment control bit is set (bit 5 = 1), REP XCRYPT instructions require that 112 bytes of scratch memory be allocated immediately after the control word. (A total of 128 bytes at ES:EDX) This memory is used by the microcode of REP XCRYPT instructions to move unaligned plaintext and ciphertext data to 16-byte aligned buffers when required by ACE hardware.

The control word fields are:

127:12	11:10	9	8	7	6	5	4	3:0
RESRV	KSIZE	CRYPT	INTER	KEYGN	CIPHR	ALIGN	DGEST	ROUND

ROUND CW[3:0] Round count: This four-bit field contains the number of rounds to be used for the encryption or decryption operation. Although the number of rounds for each key size of AES is specified by the FIPS standard, this value is not built into the hardware; the application program must specify the number of rounds. Note that a round count of 0 will encrypt or decrypt for 16 rounds. Assembly language programmers can think of the round count as a loop index, not a rep counter, in a 4-bit register.

For key lengths of 128 bits, the FIPS-197 specified number of rounds is 10.
For key lengths of 192 bits, the FIPS-197 specified number of rounds is 12.
For key lengths of 256 bits, the FIPS-197 specified number of rounds is 14.

DGEST CW[4] Digest mode: This one-bit field specifies whether the CBC and CFB modes should operate normally (bit 4 = 0), when each encrypted plaintext block (XORd per operating mode) is stored in a separate ciphertext block. In Digest mode (bit 4 = 1), the ciphertext buffer is not incremented and the AES calculates what is known as a Message Authentication Code.

On Nehemiah processors, a non-zero value in this field is undefined as the feature is not supported.

Message Authentication Code is defined by the standards only for encryption. However, ACE hardware will not increment the pointer specified in ES:EDI for the CBC and CFB operating modes regardless of the state of the encrypt/decrypt bit of the control word.

ALIGN CW[5] 16-byte alignment: This one-bit field specifies whether the plaintext and ciphertext buffers must be 16-byte aligned. When bit 5 = 0, which is required for VIA Nehemiah ACE, the plaintext and ciphertext buffers must be 16-byte aligned. When bit 5 = 1, this requirement is relaxed, and the VIA Esther ACE executes microcode that internally copies the non-aligned plaintext and ciphertext buffers to and from known aligned memory so that the hardware (which still does require 16-byte alignment) will operate correctly.

The 16-byte alignment for plaintext and ciphertext is still required for ECB mode, and REP XCRYPTECB will fault if bit 5 = 1 and either plaintext or ciphertext is not 16-byte aligned.

On Nehemiah processors, a non-zero value in this field is undefined as the feature is not supported.

CIPHR CW[6] Cipher Algorithm: This one-bit field specifies the cryptographic algorithm. The only algorithm currently supported is the AES, which has a field value of 0. ACE behavior is undefined if CW[6]=1.

The Nehemiah processor supported only one algorithm in ACE: AES with the plaintext and ciphertext buffers aligned on 16-byte boundaries. As non-zero values for bits 4 and 5 were undefined for Nehemiah ACE, existing software written for that processor will work as before and will execute the same microcode sequences optimized for (and assuming) 16-byte alignment of the plaintext and ciphertext buffers.

KEYGN CW[7] Key Generation: This one-bit field specifies whether the extended keys are to be generated by the hardware (0) or are to be loaded from memory (1). Hardware key generation is supported only for 128-bit keys, and setting bit 7 = 0 when bit 10 = 1 or bit 11 = 1 (setting key size to other than 128 bits) will result in undefined behavior by PadLock ACE.

INTER CW[8] Intermediate/Normal: This one-bit field specifies normal operation (0) or intermediate operation (1). Intermediate mode allows examination of the result of intermediate rounds.

CRYPT CW[9] Encrypt/Decrypt: This one-bit field specifies encryption (0) or decryption (1).

KEYSZ CW[11:10] Key size: This two-bit field specifies the key size: 128-bit (0), 192-bit (1), 256-bit (2), and reserved (3). ACE behavior is undefined if key size is set to the reserved value.

RESRV CW[127:12] Reserved: These bits should be set to 0 and the contents of this field should not be used by the application.

3.3 Performance and Timing

As for all x86 instructions that reference memory, the timing of REP XCRYPT instructions is non-deterministic due to cache misses, page faults, interrupts, load and store buffer stalls, SSE queue-full stalls, external DMA snoops, etc. Consistent with other published x86 instruction timings, this section ignores these variables and assumes a perfect environment: everything is in the cache, no interrupts, no snoops, no stalls, etc.

ACE instruction timing is more complicated in that a large number of REP XCRYPT execution clocks are overlapped with subsequent instruction execution clocks. As an example, referring to the REP XCRYPTECB TIMING Table, we see that a REP XCRYPTECB instruction requires 31 clocks to execute (for a round count of 10 and ECX=2) from the viewpoint of an SSE instruction that follows the REP XCRYPTECB instruction. If, however, the REP XCRYPTECB instruction is followed by integer, MMX, or floating point instructions, the REP XCRYPTECB instruction only appears to take 12 clocks relative to these instructions. The missing 19 clocks of ACE execution are overlapped with 19 clocks of execution of non-SSE instructions.

The following symbols are defined:

- R = the number of rounds specified
- N = the number of 128-bit blocks to be encrypted

Microcode is optimized for execution with the extended key and control word already loaded. Thus, ACE is started immediately on the assumption that the system is ready and only then is the EFLAGS bit tested to determine if keys should be loaded. If so, microcode resets ACE to halt the encryption, loads or generates the key, and restarts. Along the way, ECX is tested for zero. That means that unlike REP string instructions where an ECX value of zero is a NOP, it is possible for REP XCRYPT instructions to page fault when ECX = 0. When ECX = 0, REP XCRYPT instructions take between 10 and 27 clocks, depending on the state of ELFAGS:30, the processor and stepping, and the operating mode.

For large values of the REP count we do not provide separate integer timings. The integer pipeline quickly slips and stalls into sync with ACE and the associated SSE data paths. At the completion of the cipher operation, there will be a slight overlap where the integer unit can fetch subsequent instructions while ACE completes the final block. This overlap is comparable to the extra clocks available when doing a single encryption or decryption.

REP XCRYPTECB TIMING

Data	EFLAGS:30	Key Source	SSE Base Clocks	SSE Var Clocks	Integer Base Clocks
128b	1	N/A	9	2*R	17
128b	0	Hardware	45	2*R	61
128b	0	Memory	76	2*R	84
2*128b	1	N/A	11	2*R	12
2*128b	0	Hardware	47	2*R	56
2*128b	0	Memory	78	2*R	79
N*128b	1	N/A	11	N*R	N/A
N*128b	0	Hardware	47	N*R	N/A
N*128b	0	Memory	78	N*R	N/A

Notes:

The REP XCRYPTECB instruction always operates on two 128-bit blocks of data. However when the block count in ECX is odd, the extra block of data is not stored.

When ECX is odd and greater than 2, add an extra (R-2) clocks to the SSE Base clocks calculation. The extra R comes from the fact that ACE does an even number of rounds, but saves two clocks on the final store(s) since only the first block is written to memory.

Measured Performance

The following data were generated using the sample program provided in Chapter 6 of this manual. Please note that your mileage may vary! The clock() function is not particularly fine-grained and for programs that execute for only a few seconds, differences in timing measurements on the order of 10%-15% may be observed. You may also see differences on the order of 10%-15% (or more!) depending on the operating system used, the amount of memory available, the motherboard, other processes that may be running or otherwise consuming resources, and - well, you get the idea.

Note that for very large data blocks, performance is strictly limited by memory bandwidth as the data loads will almost always miss in the cache.

1.2 GHz Nehemiah with 16-byte aligned buffers (Gb/s)

Data Size	ECB	CBC	CFB	OFB
16B	3.28	3.37	2.98	1.86
32B	5.79	5.29	4.83	2.38
64B	10.67	5.82	5.82	2.65
128B	12.05	6.04	6.02	2.82
256B	12.88	6.19	6.10	2.91
512B	13.52	6.36	6.22	3.03
1KB	13.88	6.40	6.33	3.06
2KB	13.94	6.38	6.34	3.06
4KB	14.00	6.43	6.28	3.08
8KB	13.94	6.43	6.39	3.09
16KB	14.01	6.43	6.24	3.10
32KB	10.66	6.24	6.11	2.89
64KB	9.45	6.02	5.90	2.82
1MB	1.05	1.05	1.05	0.89

1.6 GHz Esther with 16-byte aligned buffers (Gb/s)

Data Size	ECB	CBC	CFB	OFB
16B	4.13	3.88	3.66	2.11
32B	6.83	6.03	5.75	2.96
64B	12.19	7.53	7.31	3.71
128B	14.11	8.36	8.19	4.27
256B	15.21	8.71	8.71	4.60
512B	16.07	8.80	8.80	4.82
1KB	16.72	8.90	8.90	4.93
2KB	16.83	8.98	8.96	4.96
4KB	17.03	8.93	8.93	5.00
8KB	17.03	8.93	8.93	5.00
16KB	17.03	8.98	8.93	5.02
32KB	17.20	8.97	8.89	5.04
64KB	14.56	8.67	8.67	4.72
1MB	3.09	2.69	2.69	2.56

1.6 Ghz Esther with non-aligned buffers (Gb/s)

Data Size	CTR	CBC	CFB	OFB
16B	1.52	1.56	1.49	1.42
32B	2.37	1.78	1.78	1.70
64B	3.27	2.31	2.28	2.18
128B	4.06	2.69	2.66	2.54
256B	4.10	2.95	2.93	2.77
512B	4.90	3.08	3.08	2.90
1KB	5.09	3.17	3.15	2.98
2KB	5.20	3.20	3.20	3.01
4KB	5.24	3.23	3.22	3.02
8KB	5.26	3.23	3.23	3.03
16KB	5.28	3.24	3.24	3.04
32KB	5.28	3.25	3.25	3.04
64KB	5.14	3.20	3.20	3.01
1MB	1.68	2.17	2.17	2.09

Chapter 4

Hash Engine

Federal Information Processing Standards Publication (FIPS) 180-2 defines the Secure Hash Standard. As part of the standard, FIPS 180-2 specifies four secure hash algorithms. These algorithms are used to provide a condensed representation, or message digest, of electronic data - an email or perhaps a file stored on your hard drive.

4.1 REP XSHA Instructions

REP XSHA1: Hash Function SHA-1

0xF3	0x0F	0xA6	0xC8

	Input	Output
EAX	0x00000000	REP count
ECX	REP count	
ES:[ESI]	Source input blocks	
ES:[EDI]	SHA-1 constants	Hash of input blocks

This instruction performs the SHA-1 algorithm, producing a 160-bit hash of the source as defined by FIPS 180-2.

REP XSHA256: Hash Function SHA-256

0xF3	0x0F	0xA6	0xD0

	Input	Output
EAX	0x00000000	REP count
ECX	REP count	
ES:[ESI]	Source input blocks	
ES:[EDI]	SHA-256 constants	Hash of input blocks

This instruction performs the SHA-256 Algorithm, producing a 256-bit hash of the source as defined by FIPS 180-2.

Memory Formats

Only Hash Engine (PHE) data stores must be 16-byte aligned. Like other Padlock functions, PHE uses SSE datapaths and logic, and most loads to and stores from Padlock registers must be 16-byte aligned. This is true for the initial constants and resulting hash pointed at by EDI.

In the Secure Hash Algorithm standard, as defined by the FIPS 180-2 specification, the input stream is defined as a sequence of 32-bit words in big-endian format. PHE is designed according to this big endian definition. However, VIA processors conform to the memory format of Intel x86 processors, which is little endian.

Therefore, the microcode of PHE automatically converts the input stream from little endian to big endian before loading PHE registers. As this process is equivalent to a 32-bit BSWAP instruction, data is loaded from memory by REP XSHA instructions in 32-bit chunks, which makes the 16-byte alignment a superfluous requirement for the input stream.

The FIPS 180-2 standard actually specifies that the input is a bit stream, not a byte stream. However, like most hardware implementations of the Secure Hash Algorithm of which we are aware, PHE operates at the byte level, that is the length L of the input bit stream must be such that: L mod 8 = 0

Padding

The FIPS 180-2 specified pre-processing of the input stream, appending the single 1 bit followed by a number of zero bits and finally the size (in bits) of the input stream, is handled automatically by the microcode for REP XSHA instructions. Not even the conventional 0x00 byte following a C-language string is required by REP XSHA instructions for correct processing.

Input Data Pointer: ESI

ESI (or SI) in all REP XSHA instructions contains the effective address (offset) of the input data. This address is always relative to the segment specified in ES.

At the end of instruction execution, or if an interrupt (or page fault, etc.) has occurred, ESI (or SI) is incremented by the number of bytes hashed. The address is always incremented: EFLAGS.DF has no effect. Whether SI or ESI is used and updated is based on the effective address size for the executed instruction.

The padding specified by the FIPS 180-2 standard for SHA-1 and SHA-256 is performed by the microcode associated with REP XHSA instructions. No special termination character for the input data is required. It is perfectly valid to hash any stream of bytes including an arbitrary number of NULL (0x00) bytes.

Output Data (Hash) Pointer: EDI

EDI (or DI) in all REP XSHA instructions contains the effective address (offset) of the result data generated by the instruction. This address is always relative to the segment specified in ES. The resulting linear address must be aligned on a 128-bit boundary (16-byte aligned); otherwise a General Protection exception occurs.

The total memory allocated at ES:EDI (or ES:DI) must be 128 bytes. REP XSHA microcode requires a 16-byte aligned buffer to load the PHE hardware. Only the hash pointer at EDI is guaranteed to be aligned.

At the start of a REP XSHA instruction, the memory pointed to by EDI (or DI) must contain the initial hash constants as specified by FIPS 180-2, stored in standard Intel little-endian format:

Memory	SHA-1	SHA-256
ES:[EDI]	0x67452301	0x6A09E667
ES:[EDI+4]	0xEFCDAB89	0BB67AE85
ES:[EDI+8]	0x98BADCFE	0x3C6EF372
ES:[EDI+12]	0x10325476	0xA54FF53A
ES:[EDI+16]	0xC3D2E1F0	0x510E527F
ES:[EDI+20]		0x9B05688C
ES:[EDI+24]		0x1F83D9AB
ES:[EDI+28]		0x5BE0CD19

After each 64-byte block of the input has been processed, the values at EDI (or DI) are written to from the PHE output registers with the current hash. At this point, with the pointer in ESI is incremented and the value in EAX is adjusted to reflect the total number of bytes processed. The REP XSHA instruction is interrupt and task- switch safe.

When the current (or final) hash result is stored into memory at ES:EDI, it is stored as a series of 32-bit integers in standard Intel little-endian format. However, correct representation of the hash, as specified by FIPS 180-2, is in big-endian format. Thus the calling program will need to BSWAP each 32-bit DWORD of the hash to produce the correct result if it is to be communicated to any other program, process, or user.

Total Number of Bytes to Hash: ECX

ECX contains the count of 8-bit data blocks (bytes) to be processed.

Unlike other x86 REP instructions, the value in ECX is never changed. But, also unlike other x86 REP instructions, it is a valid operation to take the hash of a null string (ECX = 0). So PadLock Hash Engine uses a second counter register (EAX) to keep track of instruction progress.

Bytes Hashed: EAX

EAX must be initialized to 0x00000000 at the beginning of a REP XSHA instruction. As 64 byte blocks are hashed, the value in EAX is incremented to reflect the state of the REP XSHA instruction, so that interrupts and task switches can be transparently supported. When the REP XSHA instruction completes, EAX == ECX.

REP Prefix

- The number of source bytes processed (hashed) is added to the pointer in ESI.
- An address-size prefix affects the size of ESI and EDI used.
- The usual address exceptions (past the segment limit, page fault, etc.) can occur.
- The value in EDI is never modified.
- The value in ECX is never modified.
- A REPNE prefix generates undefined behavior.
- An operand size prefix causes an Invalid Instruction exception.
- The DF (direction flag) in EFLAGS has no effect. The operation always proceeds from low address to high address.

4.2 Performance and Timing

As for all x86 instructions that reference memory, the timing of REP XSHA instructions is non-deterministic due to cache misses, page faults, interrupts, load and store buffer stalls, SSE queue-full stalls, external DMA snoops, etc. Consistent with other published x86 instruction timings, this section ignores these variables and assumes a perfect environment: everything is in the cache, no interrupts, no snoops, no stalls, etc.

PHE hardware completes one round every two clocks for SHA-1, and completes one round every three clocks for SHA-256. The FIPS 180-2 specification defines SHA-1 as requiring 80 rounds, and SHA-256 requires 64 rounds. Thus, for very large byte streams, the asymptotic performance of the PHE hardware engine is 3.2 bits/clock for SHA-1 and 2.66 bits/clock for SHA-256.

However, the performance of REP XSHA instructions is somewhat less. There is overhead in getting started and in converting from little endian format into big endian format. There is overhead in performing the pre-processing (which actually happens at the end of the instruction when all the actual source bytes have been processed). There is overhead in delivering the data to the hardware engine aligned on a 16-byte boundary (as required by the hardware engine) even though the x86 instruction does not require such alignment. And there is additional overhead in transitioning from the x86 instruction stream to the microcode. For all these reasons, and because of subtle timing interlocks, the performance of REP XSHA instructions is less than the hardware maximum. Our current estimates for the asymptotic performance is 2.66 bits/clock for SHA-1 and 2.26 bits/clock for SHA-256.

But remember that this includes everything required to generate a hash of an arbitrary byte stream. And it also assumes an infinite memory bandwidth, a delightful if unrealistic assumption.

The following performance data, repeatedly calculating the hash of a 100MB file, was measured on a 1.6 Ghz VIA Esther processor, and compared with results of hashing the same file with OpenSSL.

```
$ ls -l libgcj.a
-rw-r--r-- 1 root root 100757706 Oct 14 22:56 libgcj.a
$ openssl sha1 libgcj.a
SHA1(libgcj.a)= 73561e336c53975723b3c18fc3da8b9e74151301
$ time openssl sha1 libgcj.a
SHA1(libgcj.a)= 73561e336c53975723b3c18fc3da8b9e74151301
real 0m2.039s
user 0m1.719s
sys 0m0.316s
$ time openssl sha1 libgcj.a
SHA1(libgcj.a)= 73561e336c53975723b3c18fc3da8b9e74151301
real 0m2.048s
user 0m1.732s
sys 0m0.314s
$ time openssl sha1 libgcj.a
SHA1(libgcj.a)= 73561e336c53975723b3c18fc3da8b9e74151301
real 0m2.010s
user 0m1.719s
sys 0m0.289s
$ /home/crispin/phe libgcj.a
SHA1(libgcj.a)= 73561e336c53975723b3c18fc3da8b9e74151301
$ time /home/crispin/phe libgcj.a
SHA1(libgcj.a)= 73561e336c53975723b3c18fc3da8b9e74151301
real 0m0.837s
user 0m0.330s
sys 0m0.506s
$ time /home/crispin/phe libgcj.a
SHA1(libgcj.a)= 73561e336c53975723b3c18fc3da8b9e74151301
real 0m0.838s
user 0m0.330s
sys 0m0.506s
$ time /home/crispin/phe libgcj.a
SHA1(libgcj.a)= 73561e336c53975723b3c18fc3da8b9e74151301
real 0m0.838s
user 0m0.330s
sys 0m0.506s
```

4.3 On Big-Endian and Little-Endian Formats

As discussed earlier in this chapter, the Secure Hash Algorithms as specified in FIPS180-2 operate on unsigned 32-bit integers in big-endian format, which conflicts with the little-endian memory format of Intel x86 and compatible processors, such as the VIA Esther processor. You can guess, correctly, that this is an annoyance and has caused no end of difficulty for cryptographers and programmers.

In the context of PHE, the difficulty occurs when dealing with the resulting hash value and the initial constants loaded into memory at the address specified in ES:EDI (or ES:DI). The perceptive reader will have noticed that REP XSHA instructions require that the initial constants be stored in memory as the correct unsigned 32 bit integers, but in little-endian format rather than in big-endian format. This puts the data into the correct memory format for direct loading into the appropriate PHE registers.

This initialization technique causes no particular problems, and is in fact also used by most software SHA algorithm implementations. Typically these constants will be set by a compiler, and they may as well be organized in the most efficient byte order for the hardware.

With respect to storing the resulting hash at ES:EDI (or ES:DI), this little-endian/big-endian issue would be no problem for PHE if the REP XSHA instructions were atomic. There is no particular difficulty in storing the result of the hash, as unsigned 32-bit integers, in big-endian format, little-endian format, or any other format specified by the defining authority. There would be a very small, probably not detectable, increase in the time to execute a REP XSHA instruction as the microcode waits for the PHE output registers to be stored into memory (in little-endian format) and then does the BSWAP to make things simple for the x86 programmer.

However, REP XSHA instructions are not atomic. For very large input a single REP XSHA instruction may execute for tens or hundreds of millions of clocks. To run in an un-interruptible state for that length of time is not acceptable behavior.

Thus, after every 64-byte block of input data has been hashed, a REP XSHA instruction stores the current value of the hash directly to memory at ES:EDI, without any bit manipulation, in the hardware little-endian format. The bottom line is that the value of the hash (or the initial constants) must be in the same endian format when loaded at the start of a REP XSHA instruction, and when stored at completion of the instruction (or when the intermediate hash value is stored after each 64-byte input block).

If the unsigned 32-bit integers are stored in memory in little-endian format, the microcode and hardware of the PHE can operate at maximum efficiency. The setting of the initial constants is fundamentally a matter for the compiler, and has no performance implications. And adjusting the endian-ness of the final hash value, a few BSWAP instructions, can be performed as efficiently by the surrounding x86 code as by the microcode or hardware of PHE.

However, if the 32-bit integers are stored in the FIPS 180-2 big-endian format, it will be necessary for PHE to convert the values from big-endian to little-endian for every 64-byte data block when the results are stored, as well as during the initial load when the instruction starts. Not only that; in addition to the extra time involved in swapping the bits, this adds numerous stalls to the microcode pipeline. And makes the microcode significantly larger.

Alternately, it was (I suppose - Damn it Jim: I'm a programmer not a circuit engineer) possible to cross the wires in the hardware to handle little-endian to big-endian issues directly. But the reader will recall that PHE, like PadLock ACE, uses already existing SSE datapaths and buses, which would require not just crossing the existing wires but lots of extra transistors and a more complex, and confusing, hardware design.

Chapter 5

Montgomery Multiplier

Montgomery Multiplication is a sophisticated algorithm to speedup modulo multiplication of very large integers. Readers of this guide are assumed to understand the algorithm! If you do not, we recommend that at the very least you read the *VIA Security Application Note*, which provides more detail. A sample program in Chapter 6 provides an example of Montgomery Multiplication in C, tailored to the actual hardware of the VIA Esther processor.

5.1 REP MONTMUL

0xF3	0x0F	0xA6	0xC0

	Input	Output
EAX	0x00000000	Undefined
ECX	REP count	0
EDX		0
ES:[ESI]	MONTMUL context	

This instruction performs the Montgomery Multiplication algorithm, as described in the *VIA Security Application Note*, and as illustrated by sample program MM.C in Chapter 6, or see any competent mathematics text.

Memory Formats

All PMM data loads and stores must be 16-byte aligned. Relative to PMM notation, byte 0 is at the lowest address, byte 1 is at the next higher address, and so forth. That is, data is organized in the normal x86 little endian fashion such that PMM byte 0 is the normal x86 architecture byte 0 (the linear address provided by an instruction operand).

Thus the big integers that are pointed to in the Montgomery context must be 16-byte aligned. However, the pointer to the Montgomery context itself does not have this 16-byte alignment requirement.

Number of Bits in the Big Integers: ECX

ECX contains the number of bits in the big integers A, B, M, and T. The valid values for ECX are:

$$256 \ \le ECX \ \le 32768$$

$$ECX \; mod \, 128 = 0$$

At the end of instruction execution, ECX will be zero. If an interrupt (or page fault, etc.) has occurred, the value in ECX is not modified unless the instruction has completed. Information regarding the state of the interrupted instruction is stored in the EAX register.

Pointer to Montgomery Context: ESI

The Montgomery context is a structure that defines the operands used in the Montgomery multiplication, and is comparable to structures found in software implementations of the Montgomery algorithm. The four pointers (bigint*) A, B, T, and M, and the Scratch Buffer, must be aligned on 16-byte boundaries. The pointer in ESI is not so resticted.

All addresses assume the ES segment selector, which may not be overridden.

```
typedef struct _mmCTX {
    uint_32 mZeroPrime;
    bigint* A;
    bigint* B;
    bigint* T; /* Result buffer: initially must be all zero */
    bigint* M; /* Modulus  */
    uint_32[8]* Scratch /* aligned on a 16 byte boundary */
} mmCTX;
```

Big integers are defined as below. Note that there is no leading count specifying the number of bytes, or words, or uint_32 in the big integer. For the purposes of the multiply, that information is provided in the REP count in ECX.

```
typedef struct _bigint {
    uint_32 VALUE[MAX_WORDS+4];
} bigint;
```

NOTES:

It is possible for the partial product buffer to overflow in the Montgomery calculation. That is, the partial product can be a little bit larger than the modulus M. If this occurs in the final product, the software using REP MONTMUL will check for this and in that case the result will be T - M. This is well known to implementers of the Montgomery algorithm. (See the code samples in Chapters 6) What this means for PMM is that the data structure defining the big integer T (the partial product) must have an additional 128 bits of zeroes allocated to provide storage for the overflow should it occur.

Please see the illustrative code in Chapter 6 to learn how to calculate mZeroPrime.

REP MONTMUL State: EAX

EAX must be initialized to 0x00000000 at the start of a REP MONTMUL instruction. As a REP MONTMUL instruction can run for a long time (with ECX = 4096 the time is on the order of 67,000 clocks), it supports interrupts much like other Padlock instructions. However, as large integer modulo multiplication is a double loop, the value of ECX must be retained and state for restarting the instruction contained elsewhere. The REP MONTMUL instruction uses EAX to retain state regarding re-starting the instruction after an interrupt has been serviced. The partial product is continuously updated in memory so as to be correct when restarting after an interrupt.

Except for initializing EAX = 0x00000000 at the start of the instruction, the value in EAX should be considered undefined by your program. Centaur Technology may change the way this state information is preserved in the EAX register for microcode efficiency, or to reflect improvements in underlying PMM hardware.

REP MONTMUL Temp: EDX

EDX is used by the microcode of the REP MONTMUL instruction to store temporary data during the operation, as a precaution against certain interrupt conditions. The value in EDX is not used as input to the REP MONTMUL instruction, and the value is set to zero when the instruction completes.

REP Prefix

- An address-size prefix affects the size of ESI.
- The usual address exceptions (past the segment limit, page fault, etc.) can occur.
- The value in ESI is unchanged as it isn't a source register pointer
- EDI is not used, there is no destination pointer.
- A REPNE prefix causes undefined behavior.
- The DF (direction flag) in EFLAGS has no effect.
- The full 32 bit EAX and ECX registers are used regardless of any address or operand prefix.
- The value in ECX does not change until the operation completes, when it is set to zero.

5.2 Performance and Timing

As for all x86 instructions that reference memory, the timing of REP MONTMUL instructions is non-deterministic due to cache misses, page faults, interrupts, load and store buffer stalls, SSE queue-full stalls, external DMA snoops, etc. Consistent with other published x86 instruction timings, this section ignores these variables and assumes a perfect environment: everything is in the cache, no interrupts, no snoops, no stalls, etc.

The REP MONTMUL instruction is particularly complex. Our measurements (in simulation) are in line with our original estimates:

Let [M] be the number of bits in the big integers A, B, and M. (Including any necessary zeros padded at the left so that [M] is a multiple of 128.) Let N = [M] / 1024, that is N be the number of kilobits in the big integers. Then the number of clocks C to execute a REP MONTMUL instruction is

$$C = 1024 * (4 * N^2 + K_1 * N) + K_2$$

$$0.56 \le K_1 \le 1.06$$

$$K_2 \approx 35$$

Why is the K_1 indeterminate? There are three factors to consider:

First, the Montgomery algorithm can overflow, as you can see illustrated in the sample code. This is when it will be necessary to subtract the modulus (one time) from the result. In this case, the calculations require one extra (partial) execution of the inner loop for every pass through the outer loop once the overflow condition occurs.

Second, although the numbers are 16-byte aligned, at the conclusion of each outer loop the 33 bits of the carry are written to memory using a 64-bit MMX store instruction, which is not 16-byte aligned, but 4-byte aligned. There is no performance penalty for this store except when the store crosses a cache-line boundary.

Fortunately, these two conditions can never both occur. Whenever the extra calculation is required, the final store of the carry bits occurs on a 16-byte aligned boundary.

A third factor is the size of the big integers [M]. The microcode executes the inner loop of the algorithm, in the integer pipeline, in 15 clocks, while the PMM hardware and the associated MMX logic require 16 clocks for the inner loop. For smaller values of [M], these extra clocks for the REP MONTMUL calculation simply overlap some of the integer instructions necessary for the outer loop. For larger values of [M] the MMX pipeline will eventually fill and the integer pipeline stalls for one clock during the execution of the inner loop. In this latter case, however, the effective overlap of the integer pipeline with the REP MONTMUL calculation during the tail of the outer loop is at a maximum. The smallest measured value for K_1 (0.561) occurred during a simulation using numbers with [M] = 2304.

Of course, it is important to remember that the time to perform the REP MONTMUL instruction is not the total time to perform a modular multiplication.

Estimated RSA Performance

Based on the timing numbers above, we can estimate the performance of a VIA Esther processor for the RSA public-key encryption algorithm.

Assume a 2 Ghz VIA Esther processor and 1024-bit keys. Assume also that the time to perform the (many) multiplication operations dwarfs all other instruction overhead. Assume that the Chinese Remainder Theorem has been used so that the actual modexp calculations will be with 512-bit numbers rather than with 1024-bit numbers.

With those assumptions, a 2 Ghz VIA Esther processor will perform about 900 RSA signings per second, for 1024 bit keys. Similar assumptions for larger key lengths lead to estimates of about 130 RSA signings per second for 2048 bit keys, and about 18 signings per second for 4096 bit keys.

Measured MODEXP() Performance

Now, lets get back to the real world.

While the performance of the REP MONTMUL instruction as measured conforms closely to the theoretical estimates given above, the overhead of the other instructions in performing a modexp calculation is not negligible, especially for small values of the modulus.

Below we show representative output from the sample code of Chapter 6, which performs modexp calculations for various key lengths using both the GMP library and with code using the REP MONTMUL instruction. These values were calculated using a VIA Esther processor running at 1.6 Ghz.

```
LENGTH = 512 bits.
GMP: 250.0 ModExp/per sec
PMM: 1233.3 ModExp/per sec
LENGTH = 1024 bits.
GMP: 50.0 ModExp/per sec
PMM: 194.7 ModExp/per sec
LENGTH = 1536 bits.
GMP: 16.1 ModExp/per sec
PMM: 62.7 ModExp/per sec
LENGTH = 2048 bits.
GMP: 7.1 ModExp/per sec
PMM: 27.4 ModExp/per sec
```

Chapter 6

Sample Code

6.1 CRAP.C [Centaur RNG Analysis Program]

```
/*
    Centaur RNG Analysis Program
    Copyright (C) 2003-2005 Centaur Technology, Inc.
    All rights reserved.
    Redistribution and use in source and binary forms, with or without modification, are
    permitted provided that the following conditions are met:
    * Redistributions of source code must retain the above copyright notice, this list of
    conditions and the following disclaimer.
    * Redistributions in binary form must reproduce the above copyright notice, this list of
    conditions and the following disclaimer in the documentation and/or other materials
    provided with the distribution.
    * Neither the name of Centaur Technology nor the names of its contributors may be used to
    endorse or promote products derived from this software without specific prior written
    permission.
    THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS
    OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
    MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
    COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
    EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
    SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
    HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR
    TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
    SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
    */
    #include <stdio.h>
    #include <math.h>
    #include <stddef.h>
    #include <time.h>
    #include <stdlib.h>
    #include <string.h>
    #define TRUE 1
    #define FALSE 0
    #define XSTORE(SIZE, DIVIDER, DATA, X, Y, Z ) \
        asm __volatile__ (".byte 0xF3, 0x0F, 0xA7, 0xC0\n" \
        : "=c" (X), "=a" (Y), "=D" (Z) \
        : "c" (SIZE), "d" (DIVIDER), "D" (DATA));
    #define SHA_1(SIZE, DATA, HASH, X, Y) \
        asm __volatile__ (".byte 0xF3, 0x0F, 0xA6, 0xC8\n" \
        : "=a" (X), "=c" (Y), "=D" (HASH) \
        : "c" (SIZE), "a" (0), "S" (DATA), "D" (HASH));
```

```
/* Command line parameters */
/* -bias <N> Set the DC BIAS to the specified value [Requires wrmsr driver] */
int dc_bias = 0;
/* -edx <N> Value of the EDX divider */
int divider = 3;
/* -limit <X> Specify a limit at which to stop a biastest. Default 0.001 */
float limit = 0.001;
/* -mode Select which RNG device to use. [Requires wrmsr driver] */
int rng_mode = 0;
/* -o Output dir for TXT and RNG and other report files */
char * odir = "/rngdata";
/* -pvalue <X> Adjust chi-square pvalue to trigger full tests. Default 0.01 */
double trigger = 0.0;
/* -raw Evaluate raw bits [Requires wrmsr driver] */
int raw_bits = FALSE;
/* -rep <N> Repeat N times */
int repeat = 1;
/* -save Save the RNG sample to disk */
int out_flag = FALSE;
/* -sha <N> Pass bits thru SHA-1 hash. <N> bytes reduced to 20 */
int sha_flag = 0;
/* -size <N> Size RNG data sample - in megabytes. Default: 10MB */
int data_size = 0;
int megabytes = 0;
/* -speed Just collect bits as fast as possible. No tests */
int speed_path = FALSE;
/* -test <N> Tests each dc bias until or test limit [Requires wrmsr driver] */
int biastest = 0;
/* Other globals */
char * hostname = NULL;
char * this_date = NULL;
char * chip_ID = NULL;
char flags[20] = "";
char outfile[100] = "";
float rate = 0.0;
unsigned char * rng_buffer = NULL;
unsigned int * sha_buffer = NULL;
double ks_result = 0.0;
double chi_result = 0.0;
static FILE *random_out = NULL;
int iter;
int lines_written = 0;
int do_bias_set = FALSE;
int acmp(const void *a, const void *b) {
    if (* (double *)a < *(double *)b)
        return -1;
    if (* (double *)a > *(double *)b)
        return 1;
    return 0;
}
void asort(double *list, int n) {
    qsort(list, n, sizeof(double), acmp);
}
double kstest(double *y, int n) {
    double t, z;
    int i;
    asort(y, n);
    z = 0.0 - (double) n * n;
    for (i = 0; i < n; ++i) {
        t = y[i] * (1.0 - y[n-1 - i]);
        if (t < 1e-20) {
            t = 1e-20;
        }
        z -= (i + i + 1) * log(t);
    }
    z /= n;
    if (z < 0.01) {
        return 0.0;
    }
    if (z <= 2.0) {

        return exp(-1.2337/z)*2.0*(z/8.0 + 1.0 - z*0.04958*z / (z+1.325)) / sqrt(z);
    }
    if (z <= 4.0) {

        return 1.0 - exp(z * -1.091638) * 0.6621361 - exp(z * -2.005138) * 0.95059;
    }
    return 1.0 - exp(z * -1.050321) * 0.4938691 - exp(z * -1.527198) * 0.5946335;
}
int cleanup() {
    if (random_out)
        fclose(random_out);
    if (rng_buffer)
        free(rng_buffer);
    if (sha_buffer)
        free(sha_buffer);
}
void collect_and_hash() {
    int i,j;
    int T1, T2, T3;
    unsigned char temp_buffer[128];
    unsigned char * sha = (void *) sha_buffer;
    for (j = 0; j < data_size; j += 20) {
        sha_buffer[0] = 0x67452391;
        sha_buffer[1] = 0xefcdab89;
        sha_buffer[2] = 0x98badcfe;
        sha_buffer[3] = 0x10325476;
        sha_buffer[4] = 0xc3d2e1f0;
        XSTORE(sha_flag, divider, temp_buffer, T1, T2, T3);
        SHA_1(sha_flag, temp_buffer, sha_buffer, T1, T2);
        for (i=0; i<20; i++)
            rng_buffer[j+i] = sha[i];
    }
}
int collect_rng() {
    int T1, T2, T3;
    clock_t start = clock();
    if (sha_flag)
        collect_and_hash();
    else
        XSTORE(data_size, divider, rng_buffer, T1, T2, T3);
    return (int) ( clock() - start );
```

```
}
void processor_abort() {
    printf ("PROCESSOR ERROR. CANNOT GET HW RNG FROM THIS COMPUTER.\n");
    cleanup();
    exit (1);
}
void validate_processor() {
    /* Verify that we have the RNG and that it is enabled */
    asm ("pushl %ebx\n");
    asm ("pushl %ecx\n");
    asm ("pushl %edx\n");
    asm ("movl $0xC0000000, %eax\n");
    asm ("cpuid\n");
    asm ("cmpl $0xC0000000, %eax\n");
    asm ("jc processor_abort");
    asm ("movl $0xC0000001, %eax\n");
    asm ("cpuid\n");
    asm ("andl $0xC, %edx\n");
    asm ("cmpl $0xC, %edx\n");
    asm ("jnz processor_abort\n");
    asm ("popl %edx\n");
    asm ("popl %ecx\n");
    asm ("popl %ebx\n");
}
void process_command_line(int argc, char **argv) {
    int i, file_tests;
    char * c;
    for (i = 1; i < argc; i++) {
       c = argv[i];
       /* Set specific dc_bias [Requires ring 0 access/driver] */
       if (!strcmp(c,"-bias")) {
          dc_bias = atoi(argv[++i]);
          do_bias_set = TRUE;
       }
       /* Specify the EDX divider value. Default = 3 */
       if (!strcmp(c,"-edx"))
          divider = (atoi(argv[++i]) % 4);
       /* Specify a biastest limit. If the global chi-square or KS test approach the
       repective limits with this degree of error, terminate immediately rather than
       at the test maximum */
       if (!strcmp(c,"-limit"))
          limit = atof(argv[++i]);
       /* Which set of oscillators to use [Requires ring 0 access/driver] */
       if (!strcmp(c,"-mode")) {
          rng_mode = atoi(argv[++i]);
          do_bias_set = TRUE;
       }
       /* Specify directory for output reports and RNG saved data.
       Default: /rngdata */
       if (!strcmp(c,"-o"))
          odir = argv[++i];
       /* Specify a critical pvalue. If the chi-square test is less than this, save
       bits to the RNG file for later processing */
       if (!strcmp(c,"-pvalue"))
          trigger = atof(argv[++i]);
       /* Use the raw bits. [Requires ring 0 access/driver] */
       if (!strcmp(c,"-raw")) {
           raw_bits = TRUE;
           do_bias_set = TRUE;
        }
        /* Repeat the test(s) N times */
        if (!strcmp(c,"-rep"))
           repeat = atoi(argv[++i]);
        /* Save the RNG sample to a disk file */
        if (!strcmp(c,"-save"))
           out_flag = TRUE;
        /* Pass bits thru SHA-1. [Requires ESTHER processor] */
        if (!strcmp(c,"-sha"))
           sha_flag = atoi(argv[++i]);
        /* Repeat the test(s) N times */
        if (!strcmp(c,"-size"))
           megabytes = atoi(argv[++i]);
        /* Special speed path for power consumption testing */
        if (!strcmp(c,"-speed"))
           speed_path = TRUE;
        /* Run tests on all bias settings */
        if (!strcmp(c,"-test")) {
           biastest = atoi(argv[++i]);
           do_bias_set = TRUE;
        }
    }
    /* Validation and consistency checks */
    if (sha_flag > 128)
        sha_flag = 128;
    if (!do_bias_set)
        biastest = 0;
    if (sha_flag)
        strcat(flags, "H");
    if (raw_bits)
        strcat(flags, "R");
    if (biastest)
        strcat(flags, "T");
    if (repeat < 1)
        repeat = 1;
    if (limit > 0.01)
        limit = 0.01;
    if (biastest > 1000)
        biastest = 1000;
    /* BIAS values are allowed only in the range 0-7. */
    if ( (dc_bias < 0) || (dc_bias > 7) )
        dc_bias = 0;
    /* Make sure that we have a valid data_size for the test sequence. In MB */
    if (megabytes < 1)
        megabytes = 10;
    /* Convert to absolute number of bytes */
    data_size = megabytes * 1024 * 1024;
}
/*
Tweaking the RNG MSR requires access to a priveleged instruction. We have
a device driver that supports commnand line rdmsr/wrmsr instuctions. Any
way you want to do this is fine, just remember that the command line
```

```
parameters
-bias
-raw
-mode
-test
will cause the program to access this routine where writing the RNG MSR is
necessary for correct program operation
*/
int c5xl_bias() {
    char cmd[60];
    int temp;
    if (do_bias_set) {
        temp = 0x40 + (rng_mode<<8) + (dc_bias<<10) + (raw_bits<<13);
        sprintf(cmd,"wrmsr 0x110B 0x00000000%08X > /dev/null", temp);
        system (cmd);
    }
}
int datarate(float raw) {
    rate = raw / (8. * (float) data_size);
    rate = (float) CLOCKS_PER_SEC / rate;
}
#define log2of10 3.32192809488736234787
double chi_pvalue (double chi_sq, int N) {
    double p,t,a;
    int k;
    if ( chi_sq > 410. ) {
        return 1.0;
    }
    else {
        p = exp(-0.5 * chi_sq);
        k = N;
        p *= sqrt(2. * chi_sq / 3.14159265);
        while (k > 2) {
            p *= chi_sq / (double) k;
            k -= 2;
        }
        t = p;
        a = (double) N;
        while ( t > 0.0000000001 * p ) {
            a = a + 2.;
            t = t * chi_sq / a;
            p += t;
        }
        return p;
    }
}
static long long global_counts[256] = {0};
static long long full_size = 0;
double * chivalues;
static int chivalue_count = 0;
double chi_square_tests() {
    int i, k;
    int chi_counts[256];
    double prob[256];
    double expect, x, chisq;
    double ent;
    double p,t,a;
    full_size += data_size;
    for (k=0; k<256; k++)
        chi_counts[k] = 0;
    for (k = 0; k < data_size; k++)
        chi_counts[rng_buffer[k]]++;
    chisq = 0.0;
    ent = 0.0;
    expect = (double) data_size / 256.0;
    for (i=0; i<256; i++) {
        x = (double) chi_counts[i] - expect;
        chisq += x * x / expect;
        global_counts[i] += chi_counts[i];
    }
    p = 1.0 - chi_pvalue(chisq, 255);
    chivalues[iter-1] = p;
    printf("\t\tChi-square: %7.2f PVALUE: %.5f\n", chisq, p);
    chisq = 0.0;
    expect = (double) full_size / 256.0;
    for (i=0; i<256; i++) {
        x = (double) global_counts[i] - expect;
        chisq += x * x / expect;
    }
    chi_result = 1.0 - chi_pvalue(chisq, 255);
    printf("\t\t [Total] %7.2f %.5f\n",chisq,chi_result);
    ks_result = kstest(chivalues, iter);
    printf("\t\t KSTEST: %.5f\n", ks_result);
    lines_written += 6;
    return p;
}
int print_header() {
    printf("\tCENTAUR RNG ANALYSIS PACKAGE %s\n", this_date);
}
int print_details() {
    printf("\t%d MB samples from %s\n", megabytes, hostname);
    printf("\tDCBIAS=%d EDX DIVIDER=%d", dc_bias, divider);
    switch (rng_mode) {
        case 0:
            printf(" DEV=A");
            break;
        case 1:
            printf(" DEV=B");
            break;
        default:
            printf(" DEV=AB");
    }
    if (raw_bits)
        printf(" RAW BITS");
    if (sha_flag)
        printf(" SHA-1=%d", sha_flag);
    if (random_out)
        printf("\n\tRNG data saved to %s", outfile);
    printf("\n\t---------------------------------------------------\n");
}
int print_full_header() {
    print_header();
    print_details();
}
int formfeed() {
    printf("\f\n");
    lines_written = 0;
}
/* Global data needs to be reset when we switch bias settings */
int reset_vars() {
    int i;
    full_size = 0;
    chivalue_count = 0;
    for (i=0; i<256; i++)
        global_counts[i] = 0;
}
int rng_tests() {
    int k;
    int elapsed;
    time_t now;
    double p;
    now = time(NULL);
    this_date = ctime(&now);
    /* Get the RNG and time the operation */
    elapsed = collect_rng();
    /* Write RNG bits to disk here if requested. */
    if (out_flag)
        fwrite(rng_buffer, 1, data_size, random_out);
    datarate(elapsed);
    if (lines_written == 0)
        print_full_header();
    printf("\n\tSample %d. RNG datarate: %d kbits/sec\n", iter, (int) rate/1000);
    if (speed_path)
        return 0;
    p = chi_square_tests();
    printf("\t\t----------------\n");
    /* If the user has requested a critical p-value save the data now.
    The command-line parser makes sure that if out_flag == TRUE
    then trigger value = 0.0 */
    if ( p < trigger )
        fwrite(rng_buffer, 1, data_size, random_out);
    if (lines_written > 45)
        formfeed();
}
int main (int argc, char **argv) {
    time_t now;
    char s[12];
    char report_file[50];
    char * temp;
    int i;
    int mtbf[8] = {0};
    validate_processor();
    chivalues = (double *) valloc(1000000 * sizeof(double));
    process_command_line(argc, argv);
    rng_buffer = valloc(data_size + 8);
    sha_buffer = valloc(128);
    if (!rng_buffer || !sha_buffer) {
```

```
    fprintf(stdout,"Unable to allocate buffer for RNG\n\n");
    cleanup();
    exit (1);
  }
  hostname = getenv("HOSTNAME");
  /* Report file should be DATE.FFFFFF.txt where the F's are flags to indicate which
  tests are reported and test parameters */
  now = time(NULL);
  strftime (s, 12, "%y%j.%H%M", localtime(&now));
  sprintf(report_file, "%s/%s.%s.%d%s.TXT", odir, chip_ID, s, divider, flags);
  sprintf(outfile, "%s/%s.%s.%d%s.RNG", odir, chip_ID, s, divider, flags);
  /* Can we write the requested output file?? */
  if (out_flag || (trigger > 0.0) ) {
    random_out = fopen(outfile, "wb");
    if (!random_out) {

      fprintf(stderr, "Error creating RNG outfile %s\n\n", outfile);
      cleanup();
      exit (1);
    }
  }
  if ( !biastest ) {
    c5xl_bias();
    for (iter = 1; iter <= repeat; iter++)
      rng_tests();
  }
  else {
  for(rng_mode = 0; rng_mode < 3; rng_mode++) {
    for (i = 1; i <= repeat; i++) {
      for (dc_bias=0; dc_bias<8; dc_bias++) {
        c5xl_bias();
        for (iter = 1; iter <= biastest; iter++) {
          rng_tests();
          if((iter>3)&&((ks_result>(1.0-limit))||(chi_result<limit)))break;
        }
        mtbf[dc_bias] += iter;
        formfeed();
        reset_vars();
      }
      print_header();
      printf("\n\t--------------------------------------\n");
      printf("\n\n\n");
      for (dc_bias = 0; dc_bias < 8; dc_bias++)

        printf("\tMTBF DC BIAS = %d: %d\n",dc_bias, mtbf[dc_bias] / i );
      formfeed();
    }
    for (dc_bias = 0; dc_bias < 8; dc_bias++)
      mtbf[dc_bias] = 0;
    }
  }
  cleanup();
  exit(0);
}
```

6.2 BAES.C [Benchmark AES]

```
/*
baes.c -- benchmark/verification for VIA Esther ACE
Copyright (c) 2004-2005, Centaur Technology, Inc.
All rights reserved.
Redistribution and use in source and binary forms, with or without modification, are
permitted provided that the following conditions are met:
* Redistributions of source code must retain the above copyright notice, this list of
conditions and the following disclaimer.
* Redistributions in binary form must reproduce the above copyright notice, this list of
conditions and the following disclaimer in the documentation and/or other materials
provided with the distribution.
* Neither the name of Centaur Technology nor the names of its contributors may be used to
endorse or promote products derived from this software without specific prior written
permission.
THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS
OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR
TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
*/
/*
To build baes:
gcc -o baes baes.c
NOTE! The assembly language interface given in this program is written for maximum
clarity, especially since not all readers will be familiar with gcc syntax. So be
careful if you optimize code using this exact sequence of instructions, as the compiler
will likely be confused regarding register usage. Also, be careful to ensure that
buffers are aligned on 16-byte boundaries where so required.
It will be more efficient to create an assembly language subroutine that executes the
ACE instruction, properly saving and restoring the necessary registers. But the purpose
of this source is to illustrate how to use the XCRYPT instructions of the PadLock
Advanced Cryptography Engine, not how to write tight code. Since this program is not
optimized,the performance numbers generated for PadLock ACE should be treated as lower
bounds.
Alternately, experienced gcc programmers can create efficient macros that will instruct
the compiler regarding register usage by the XCRYPT instruction.
*/
#include <stdio.h>
#include <time.h>
/* Various control words*/
#define hardware 0x000
#define key128 0x000 + 0x00a
#define encrypt 0x000
#define decrypt 0x200
int HwkyEncrypt128[4] = {encrypt + hardware + key128, 0, 0, 0};
int HwkyDecrypt128[4] = {decrypt + hardware + key128, 0, 0, 0};
int ecb_monte_e128[4] = {0xAB7743A0, 0xD0B059E2, 0x402DBAB5, 0x1B9701A5};
int ecb_monte_d128[4] = {0x378BBFF5, 0x1F2E6F13, 0x576FEC6B, 0xBAE32120};
int cbc_monte_e128[4] = {0xBF4C842F, 0x0DA7EB78, 0x0196A4A7, 0xB61A8F38};
int cbc_monte_d128[4] = {0x1EB78F9B, 0xF9EF5C03, 0x4613FACB, 0xE0EFACE5};
/* Initial key for timing tests. We check for functionality, with the Monte Carlo test,
so we don't really care what we encrypt -- all bits look the same, performance-wise */
int ikey[8] = {0x5a5a5a5a};
/* Initialization vector - can be anything for timing */
```

```
int ivec[8] = {0};
/* Buffers for monte carlo tests */
int ctxt[8] = {0};
int dtxt[8] = {0};
int zero[8] = {0};
/* Main encryption buffer */
int buffer_ecb[1024*1024*8];
#define ECB asm (".byte 0xF3, 0x0F, 0xA7, 0xC8\n");
#define CBC asm (".byte 0xF3, 0x0F, 0xA7, 0xD0\n");
#define CTR asm (".byte 0xF3, 0x0F, 0xA7, 0xD8\n");
#define CFB asm (".byte 0xF3, 0x0F, 0xA7, 0xE0\n");
#define OFB asm (".byte 0xF3, 0x0F, 0xA7, 0xE8\n");
#define KEYS asm ("pushfl\n\tpopfl\n");
/* EAX assignment not needed for ECB but makes the macro easier */
#define REGISTERS asm \
    ("leal buffer_ecb, %esi\n" \
    "leal buffer_ecb, %edi\n" \
    "leal ikey, %ebx\n" \
    "leal ivec, %eax\n" \
    "movl buffersize, %ecx\n");
#define ENCRYPT asm \
    ("leal HwkyEncrypt128, %edx\n"); \
    REGISTERS
#define DECRYPT asm \
    ("leal HwkyEncrypt128, %edx\n"); \
    REGISTERS
#define MISALIGN asm \
    ("incl %esi\n" \
    "incl %edi\n");
#define UNALIGNED 0x20
#define MAC_MODE 0x10
int loopcounter = 0;
int buffersize = 0;
void reset_buffers() {
    int i;
    for (i=0; i<8; i++) {
        zero[i] = 0;
        dtxt[i] = 0;
        ctxt[i] = 0;
    }
}
int monte_carlo_cbc() {
    int j, k, n;
    reset_buffers();
    for (k=0; k < 400; k++) {
        KEYS
        for (j=0; j < 10000; j++) {
            asm ("leal HwkyEncrypt128, %edx\n");
            asm ("leal ctxt, %esi\n");
            asm ("leal ctxt + 16, %edi\n");
            asm ("leal dtxt, %ebx\n");
            asm ("leal zero, %eax\n");
            asm ("movl $1, %ecx\n");
            CBC
            for (n=0; n<4; n++) {
              ctxt[n] = zero[n];
              zero[n] = ctxt[4+n];
            }
        }
        for (n=0; n<4; n++)
            dtxt[n] = dtxt[n] ^ zero[n];
    }
    for (n=0; n<4; n++) {
        if (zero[n] != cbc_monte_e128[n]) {
            printf("CBC 128BIT ENCRYPT MONTE CARLO FAILED\n");
            exit(0);
        }
    }
    reset_buffers();
    for (k=0; k < 400; k++) {
        KEYS
        for (j=0; j < 10000; j++) {
            asm ("leal HwkyDecrypt128, %edx\n");
            asm ("leal ctxt, %esi\n");
            asm ("leal ctxt + 16, %edi\n");
            asm ("leal dtxt, %ebx\n");
            asm ("leal zero, %eax\n");
            asm ("movl $1, %ecx\n");
            CBC
            for (n=0; n<4; n++) {
              zero[n] = ctxt[n];
              ctxt[n] = ctxt[4+n];
            }
        }
        for (j=0; j<4; j++)
            dtxt[j] = dtxt[j] ^ ctxt[j];
    }
    for (j=0; j<4; j++) {
        if (ctxt[j] != cbc_monte_d128[j]) {
            printf("CBC 128BIT DECRYPT MONTE CARLO FAILED\n");
            exit(0);
        }
    }
}
int process_command_line(int argc, char **argv) {
    int i;
    char * c;
    int retval = 0;
    for (i = 1; i < argc; i++) {
        c = argv[i];
        if (!strcmp(c,"-rep"))
            loopcounter = atoi(argv[++i]);
        if (!strcmp(c,"-buffer"))
            buffersize = atoi(argv[++i]);
        if (!strcmp(c,"-mc"))
            retval = 1;
    }
    /* Default the encryption buffer to 8K */
    if (buffersize == 0 )
        buffersize = 512;
    /* And don't let it get bigger than 32M */
    if (buffersize > 1024*32*64)
        buffersize = 1024*32*64;
    /* Set the hardware default if the user hasn't specified a loop count */
    if (loopcounter == 0)
        loopcounter = 100000;
    return retval;
}
int main(int argc, char **argv) {
    clock_t start;
    int j;
    float datarate;
    /* Check the command line to possibly tweak the operation */
    if (process_command_line(argc, argv)) {
        printf("\nRunning Monte Carlo tests ...\n");
        monte_carlo_cbc();
        printf("PASSED\n");
    }
    printf("\nEncrypting %d byte block %d times.\n\n", buffersize * 16, loopcounter);
    #define TIME(MODE,METHOD,TEXT) \
        start = clock(); \
        for (j=0; j < loopcounter; j++) { \
        METHOD; \
        MODE; \
        } \
        datarate = (float) CLOCKS_PER_SEC * j * buffersize * 128. / (1000000000. *
        (float) (clock() - start)); \
        printf(TEXT); \
        printf(" rate: %.2f Gbs\n\n", datarate);
    #define MTIME(MODE,METHOD,TEXT) \
        start = clock(); \
        for (j=0; j < loopcounter; j++) { \
        METHOD; \
        MISALIGN; \
        MODE; \
        } \
        datarate = (float) CLOCKS_PER_SEC * j * buffersize * 128. / (1000000000. * \
        (float) (clock() - start)); \
        printf(TEXT); \
        printf(" rate: %.2f Gbs\n\n", datarate);
    TIME(CBC,ENCRYPT,"CBC mode encrypt")
    TIME(CBC,DECRYPT,"CBC mode decrypt")
    TIME(ECB,ENCRYPT,"ECB mode encrypt")
    TIME(ECB,DECRYPT,"ECB mode decrypt")
    TIME(CFB,ENCRYPT,"CFB mode encrypt")
    TIME(CFB,DECRYPT,"CFB mode decrypt")
    TIME(OFB,ENCRYPT,"OFB mode encrypt")
    TIME(OFB,DECRYPT,"OFB mode decrypt")
    TIME(CTR,ENCRYPT,"CTR mode encrypt")
    TIME(CTR,DECRYPT,"CTR mode decrypt")
    /* Set the control words for unaligned operation */
    HwkyEncrypt128[0] += UNALIGNED;
    HwkyDecrypt128[0] += UNALIGNED;
    MTIME(CBC,ENCRYPT,"CBC mode encrypt (unaligned)")
    MTIME(CBC,DECRYPT,"CBC mode decrypt (unaligned)")
    MTIME(CFB,ENCRYPT,"CFB mode encrypt (unaligned)")
    MTIME(CFB,DECRYPT,"CFB mode decrypt (unaligned)")
    MTIME(OFB,ENCRYPT,"OFB mode encrypt (unaligned)")
    MTIME(OFB,DECRYPT,"OFB mode decrypt (unaligned)")
    MTIME(CTR,ENCRYPT,"CTR mode encrypt (unaligned)")
    MTIME(CTR,DECRYPT,"CTR mode decrypt (unaligned)")
    /* Reset control words for aligned MAC */
    HwkyEncrypt128[0] += MAC_MODE - UNALIGNED;
    TIME(CBC,ENCRYPT,"CBC-MAC mode")
}
```

6.3 PHE.C [PadLock Hash Evaluator]

```
/*
PHE Test Program -- display the (SHA-1) hash of a file
Copyright (c) 2004-2005, Centaur Technololgy, Inc.
All rights reserved.
Redistribution and use in source and binary forms, with or without modification, are
permitted provided that the following conditions are met:
* Redistributions of source code must retain the above copyright notice, this list of
conditions and the following disclaimer.
* Redistributions in binary form must reproduce the above copyright notice, this list of
conditions and the following disclaimer in the  documentation and/or other materials
provided with the distribution.
* Neither the name of Centaur Technology nor the names of its contributors may be used to
endorse or promote products derived from this  software without specific prior written
permission.
THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS
OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR
TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
*/
/*
    Usage: phe <file>
*/
#include <stdlib.h>
#include <stdio.h>
/* Good enough for this context */
#define SHA_1(SIZE, DATA, HASH) \
    asm __volatile__ (".byte 0xF3, 0x0F, 0xA6, 0xC8\n" \
    : \
    : "c" (SIZE), "a" (0), "S" (DATA), "D" (HASH));
int main (int argc, char *argv[]) {
    int i;
    long filesize;
    char * data = NULL;
    /* Make sure this is at least 16-byte aligned */
    unsigned int * hash = (unsigned int *) valloc(128);
    FILE *f = fopen(argv[1], "r");
    if (!f) {
        printf("FILE ERROR\n");
    }
    else {
        fseek(f, 0, SEEK_END);
        filesize = ftell(f);
        data = malloc(filesize);
        rewind(f);
        fread(data, 1, filesize, f);
        fclose(f);
        hash[0] = 0x67452301;
        hash[1] = 0xefcdab89;
        hash[2] = 0x98badcfe;
        hash[3] = 0x10325476;
        hash[4] = 0xc3d2e1f0;
        SHA_1(filesize, data, hash)
        printf("SHA1(%s)= ", argv[1]);
        for (i=0; i<5; i++)
           printf("%08x", hash[i]);
        printf("\n");
        free(data);
    }
    return (0);
}
```

6.4 MMTEST.C [Montgomery Multiply Test]

```
/*
Montgomery Multiply Test Program (MMTEST.C)
Copyright (c) 2004-2005 Centaur Technology
All rights reserved.
Redistribution and use in source and binary forms, with or without modification, are
permitted provided that the following conditions are met:
* Redistributions of source code must retain the above copyright notice, this list of
conditions and the following disclaimer.
* Redistributions in binary form must reproduce the above copyright notice, this list of
conditions and the following disclaimer in the documentation and/or other materials
provided with the distribution.
* Neither the name of Centaur Technology nor the names of its contributors may be used to
endorse or promote products derived from this software without specific prior written
permission.
THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS
OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR
TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
NOTE REGARDING THE GNU MULTI-PRECISION ARITHMETIC LIBRARY
This test program links to routines provided by the GNU Multiple Precision Arithmetic
Library (GMP). The GMP is released under the terms of the GNU Lesser General Public License
(LGPL), a copy of which may be obtained at http://www.gnu.org/copyleft/lesser.html
Use of the GMP is not required to use the MONTMUL instruction of the VIA Esther processor.
However, should you choose to use code from the GMP in support of operations using MONTMUL,
be sure to observe the terms of that license.
To build this program:
    gcc -O3 -o mmtest mmtest.c -lgmp
*/
#include <stdio.h>
#include <gmp.h>
#include <time.h>
#define REPEAT 2
```

```
#define COUNTS 37
#define MAX_BITS 65536
#define MAX_WORDS ((MAX_BITS/32))
#define TRUE 1
#define FALSE 0
#define MONTMUL(TEMP, DUMMY, CONTEXT) \
    asm __volatile__ (".byte 0xF3, 0x0F, 0xA6, 0xC0\n" \
    : "=c" (TEMP), "=a" (DUMMY) \
    : "c" (TEMP), "a" (0), "S" (CONTEXT));
#define XSTORE(TEMP, DUMMY, DATA) \
    asm __volatile__ (".byte 0xF3, 0x0F, 0xA7, 0xC0\n" \
    : "=c" (TEMP), "=a" (DUMMY) \
    : "c" (TEMP), "d" (3), "D" (DATA));
typedef unsigned int uint_32;
typedef unsigned long long uint_64;
typedef struct _mmCTX {
    unsigned int mzeroprime;
    int *A;
    int *B;
    int *T;
    int *M;
    unsigned int *scratch;
} mmCTX;
/* Size of big integers in 32-bit (int) chunks.
Valid between 8 and 1024 in 128 bit increments. */
int LENGTH, BLENGTH;
uint_32 mZeroPrime;
mmCTX CONTEXT = {0};
uint_32 aPrime[MAX_WORDS] = {0};
uint_32 varA[MAX_WORDS] = {0};
uint_32 varB[MAX_WORDS] = {0};
uint_32 varC[MAX_WORDS] = {0};
uint_32 varM[MAX_WORDS] = {0};
uint_32 varRM[MAX_WORDS] = {0};
uint_32 Result[MAX_WORDS] = {0};
uint_32 One[MAX_WORDS] = {0};
uint_32 SCRATCH[8] = {0};
/* Inverse -modulus^(-1) mod B for odd modulus */
uint_32 bigIntInvMon(uint_32 *modulus){
    uint_32 i;
    uint_64 x = 2, y = 1;
    uint_64 mZero = (uint_64)(modulus[0]);
    for (i = 2; i <= 32; i++, x < <= 1) {
       if (x < ((mZero * y) & ((x << 1) - 1))) {
          y += x;
       }
    }
    return (uint_32)(x - y);
}
int bigIntGTE(uint_32 *A, uint_32 *B) {
    int i;
    for (i = LENGTH + 1; i >= 0; i--) {
       if (A[i] > B[i]) return TRUE;
       if (A[i] < B[i]) return FALSE;
    }
    return TRUE;
```

```
}
void bigIntSub(uint_32 *A, uint_32 *B) {
    int i,j;
    for (i = 0; i < LENGTH + 2; i++) {
        if (B[i] > A[i]) {
           j = i+1;
           A[j]--;
           while (A[j] == 0xFFFFFFFF)
             A[j++]--;
        }
        A[i] = A[i] - B[i];
    }
}
int MulMon(uint_32 *a, uint_32 *b, uint_32 *modulus, uint_32 *t) {
    int TEMP = LENGTH << 5;
    int DUMMY = 0;
    int i;
    uint_32 *temp = t;
    CONTEXT.A = a;
    CONTEXT.B = b;
    CONTEXT.T = t;
    for (i=0; i<8; i++)
        SCRATCH[i] = 0;
    for (i=0; i<LENGTH+2; i++)
        *temp++ = 0;
    MONTMUL(TEMP, DUMMY, &CONTEXT);
    if (bigIntGTE(t, modulus))
        bigIntSub(t, modulus);
}
uint_32 * X;
/* CORNER CASE!
If the first 32 bits of the exponent are zero this won't work correctly - but I'm not
gonna fix it as as this is just a sample to illustrate USE of the new MONTMUL
instruction, not a ModExp tutorial
*/
void ModExp() {
    int i,j;
    int index = LENGTH - 1;
    CONTEXT.mzeroprime = mZeroPrime;
    CONTEXT.M = varM;
    CONTEXT.scratch = &SCRATCH[0];
    uint_32 counter = 32;
    One[0] = 1;
    MulMon(varA, varRM, varM, aPrime);
    j = varB[index];
    while ( (j & 0x80000000) == 0) {
        counter--;
        j = ((unsigned int) j) << 1;
    }
    for (i=0; i<LENGTH+2; i++)
        Result[i] = aPrime[i];
    counter--;
    j = ((unsigned int) j) << 1;
    while (index >= 0) {
        while ( counter-- ) {
           MulMon(Result, Result, varM, varC);
           if (j & 0x80000000) {
              MulMon(varC, aPrime, varM, Result);
            }
            else {
              for (i=0; i<LENGTH+2; i++)
                Result[i] = varC[i];
            }
            j = ((unsigned int) j) << 1;
        }
        j = varB[--index];
        counter = 32;
    }
    MulMon(Result, One, varM, varC);
}
int main (int argc, char *argv[]) {
    int i,j,counter;
    int DUMMY = 0;
    int TEMP = 0;
    time_t gmp_ticks, pmm_ticks, ticks;
    float rate;
    mpz_t x,y,z,mod,product,r,temp;
    for (LENGTH = 8; LENGTH < 65; LENGTH += 4 ) {
        gmp_ticks = 0;
        pmm_ticks = 0;
        printf("LENGTH = %d bits.\n", LENGTH*32);
        for (counter = 0; counter < REPEAT; counter++) {

            /* Generate test case - get three random numbers from our hardware RNG */
            mpz_init2(x, 65536);
            mpz_init2(y, 65536);
            mpz_init2(z, 65536);
            mpz_init2(r, 65536);
            mpz_init2(mod, 65536);
            mpz_init2(product, 65536);
            mpz_init2(temp, 65536);
            for(i=0; i<MAX_WORDS; i++) {
              aPrime[i] = 0;
              varA[i] = 0;
              varB[i] = 0;
              varC[i] = 0;
              varM[i] = 0;
              varRM[i] = 0;
            }
            X = (unsigned int *) x;
            X[1] = LENGTH;
            TEMP = LENGTH*4;
            XSTORE(TEMP, DUMMY, X[2]);
            X = (unsigned int *) y;
            X[1] = LENGTH;
            TEMP = LENGTH*4;
            XSTORE(TEMP, DUMMY, X[2]);
            X = (unsigned int *) mod;
            X[1] = LENGTH;
            TEMP = LENGTH*4;
            XSTORE(TEMP, DUMMY, X[2]);
            if (mpz_cmpabs(x, mod) > 0)
              mpz_swap(x, mod);
            if (mpz_cmpabs(y, mod) > 0)
              mpz_swap(y, mod);
```

```
   /* If modulus is even, add 1, the algorithm requires an odd modulus */
   if (mpz_even_p(mod)) {
     mpz_add_ui(z, mod, 1);
     mpz_set(mod, z);
   }
   X = (unsigned int *) x;
   X = (unsigned int *) X[2];
   varA[LENGTH] = 0;
   for (i=0; i<LENGTH; i++)
     varA[i] = *X++;
   X = (unsigned int *) y;
   X = (unsigned int *) X[2];
   varB[LENGTH] = 0;
   for (i=0; i<LENGTH; i++)
     varB[i] = *X++;
   X = (unsigned int *) mod;
   X = (unsigned int *) X[2];
   varM[LENGTH] = 0;
   for (i=0; i<LENGTH; i++)
     varM[i] = *X++;
   /* Mod exp in software */
   ticks = clock();
   for (i=0; i<COUNTS; i++)
     mpz_powm(product,x,y,mod);
   gmp_ticks += clock() - ticks;
   /* Use the new hardware instruction -- get RM and R (mZeroPrime)
   because the hardware needs them and we can't do it any faster */
   X = (unsigned int *) temp;
   X[1] = LENGTH + 1;
   X = (unsigned int *) X[2];
   X[LENGTH] = 1;
   for (i = 0; i < LENGTH; i++)
     X[i] = 0;
   mpz_mul(z,temp,temp);
   mpz_mod(r,z,mod);
   X = (unsigned int *) r;
   X = (unsigned int *) X[2];
   varRM[LENGTH] = 0;
   for (i=0; i<LENGTH; i++)
     varRM[i] = *X++;
   mZeroPrime = bigIntInvMon(varM);
   ticks = clock();
   for (i=0; i<COUNTS; i++)
     ModExp();
   pmm_ticks += clock() - ticks;
   X = (uint_32 *) product;
   X = (uint_32 *) X[2];
   for (i=0; i<LENGTH; i++) {
     if ( varC[i] != *X ) {
       printf ("ERROR: Results differ\n\n");
       exit (0);
     }
     X++;
   }
}
rate = ((float) (COUNTS * REPEAT * CLOCKS_PER_SEC)) / gmp_ticks ;
printf(" GMP: %7.1f ModExp/per sec \n", rate);
```

```
        rate = ((float) (COUNTS * REPEAT * CLOCKS_PER_SEC)) / pmm_ticks ;
        printf(" PMM: %7.1f ModExp/per sec \n\n", rate);
    }
    return 0;
}
```

6.5 MM.C [Montgomery Algorithm Illustrated]

```
/*
Montgomery Multiply in C
Copyright (c) 2004-2005 Centaur Technology
All rights reserved.
Redistribution and use in source and binary forms, with or without modification, are
permitted provided that the following conditions are met:
* Redistributions of source code must retain the above copyright notice, this list of
conditions and the following disclaimer.
* Redistributions in binary form must reproduce the above copyright notice, this list of
conditions and the following disclaimer in the documentation and/or other materials
provided with the distribution.
* Neither the name of Centaur Technology nor the names of its contributors may be used to
endorse or promote products derived from this software without specific prior written
permission.
THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS
OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR
TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
*/
/*
This code illustrates the Montgomery Multiplication algorithm. During our
development we generated test cases using the Mathematica program, which
produced output in a format like the following:
-------------
n=2
m=10a4201d
a=a46aa5a
b=b4a3016
k=20
rm=a65c865
c=d71a93
--------------
n=4
m=f180d8c8b1ad8f1f
a=a78c9241808a4e34
b=9c8a755b17821505
k=40
rm=5cdcb8ef8a4e7317
c=3b5ab8b2823c75b5
Some sample test files had more than 200,000 test cases. This is the format of the test
cases in the input data file referenced by the code

Mathematica is a trademark of Wolfram Research, Inc.
*/
#include <stdio.h>
#include <stdlib.h>
```

```
#include <string.h>
#include <ctype.h>
#include <assert.h>
#define TRUE 1
#define FALSE 0
#define MAX_BITS 4352
#define MAX_WORDS ((MAX_BITS/16))
typedef unsigned short uint_16;
typedef unsigned int uint_32;
typedef unsigned long long uint_64;
typedef struct _bigInt {
    uint_16 length;
    uint_16 value[MAX_WORDS];
} bigInt;
int LENGTH;
/*--------------------------------------------------------------------------*/
bigInt *newBigInt(uint_32 initialValue) {
    bigInt *aBigInt = (bigInt *) malloc(sizeof(bigInt));
    int i;
    assert(aBigInt != NULL);
    aBigInt->value[0] = (uint_16) (initialValue & 0xFFFF);
    aBigInt->value[1] = (uint_16) (initialValue >> 16);
    aBigInt->length = aBigInt->value[1] ? 2 : 1;
    for (i=2; i<MAX_WORDS; i++) {
        aBigInt->value[i] = 0x0000U; /* Zero extend */
    }
    return aBigInt;
}
void deleteBigInt(bigInt *aBigInt) {
    assert(aBigInt != NULL);
    free(aBigInt);
}
/*--------------------------------------------------------------------------*/
void bigIntSub(bigInt *A, bigInt *B) {
    int i,j;
    for (i = 0; i < LENGTH + 2; i++) {
        if (B->value[i] > A->value[i]) {
           j = i+1;
           A->value[j]--;
           while (A->value[j] == 0xFFFFFFFF)
             A->value[j++]--;
           }
        A->value[i] = A->value[i] - B->value[i];
    }
}
/*--------------------------------------------------------------------------*/
/* Inverse -modulus^(-1) mod B for odd modulus */
uint_16 bigIntInvMon(bigInt *modulus){
    unsigned int i;
    uint_32 x = 2, y = 1;
    uint_32 mZero = (uint_32)(modulus->value[0]);
    for (i = 2; i <= 16; i++, x <<= 1) {
        if (x < ((mZero * y) & ((x << 1) - 1))) {
           y += x;
        }
    }
    return (uint_16)(x - y);
}
/*-------------------------------------------------------------------------*/
char hexString[MAX_WORDS*4];
char hex[] = "0123456789ABCDEF";
char *bigInt2HexString(bigInt *aBigInt) {
    int i, j;
    uint_16 a;
    for (i=aBigInt->length-1, j=0; i>=0; i--,j+=4) {
        a = aBigInt->value[i];
        hexString[j] = hex[(a >> 12) & 0X000F];
        hexString[j+1] = hex[(a >> 8 ) & 0X000F];
        hexString[j+2] = hex[(a >> 4 ) & 0X000F];
        hexString[j+3] = hex[(a ) & 0X000F];
    }
    hexString[j] = 0x00;
    return hexString;
}
/*-------------------------------------------------------------------------*/
uint_16 char2int(char c) {
    switch(c) {
        case ’0’ : return 0; break;
        case ’1’ : return 1; break;
        case ’2’ : return 2; break;
        case ’3’ : return 3; break;
        case ’4’ : return 4; break;
        case ’5’ : return 5; break;
        case ’6’ : return 6; break;
        case ’7’ : return 7; break;
        case ’8’ : return 8; break;
        case ’9’ : return 9; break;
        case ’a’ : return 10; break;
        case ’b’ : return 11; break;
        case ’c’ : return 12; break;
        case ’d’ : return 13; break;
        case ’e’ : return 14; break;
        case ’f’ : return 15; break;
        case ’A’ : return 10; break;
        case ’B’ : return 11; break;
        case ’C’ : return 12; break;
        case ’D’ : return 13; break;
        case ’E’ : return 14; break;
        case ’F’ : return 15; break;
        default:
        printf("Invalid hex character: %c\n", c);
        exit(0);
    }
}
/*-------------------------------------------------------------------------*/
uint_16 quad2int(char *aCP) {
    return((char2int(aCP[0]) << 12) | (char2int(aCP[1]) << 8) |
    (char2int(aCP[2]) << 4) | (char2int(aCP[3])));
}
/*-------------------------------------------------------------------------*/
void hexString2bigInt(char *theString, bigInt *b) {
    int len, lenm4;
    int i;
    char *aString;
    len = strlen(theString);
    lenm4 = (len % 4);
    if (lenm4) {
        lenm4 = 4 - lenm4;
    }
    aString = theString-lenm4;
    for (i=0; i<lenm4; i++) {
        aString[i] = ’0’;
    }
    len = strlen(aString);
    for (i=0; 4*i < len; i++) {
        b->value[i] = quad2int(aString+len-(4*i)-4);
    }
    b->length = i;
    for(; i < MAX_WORDS-1; i++) {
        b->value[i] = 0x0000U;
    }
}
/*--------------------------------------------------------------------------*/
int bigIntGTE(bigInt *A, bigInt *B) {
    int i;
    for (i=MAX_WORDS-1; i >= 0; i--) {
        if (A->value[i] > B->value[i]) return TRUE;
        if (A->value[i] < B->value[i]) return FALSE;
    }
    return TRUE;
}
/*--------------------------------------------------------------------------*/
int bigIntEQ(bigInt *A, bigInt *B) {
    int i;
    for (i=0; i<MAX_WORDS; i++) {
        if (A->value[i] != B->value[i]) return FALSE;
    }
    return TRUE;
}
/*--------------------------------------------------------------------------*/
uint_16 mZeroPrime;
bigInt *one;
bigInt *rSquared;
void setupMontgomery(bigInt *modulus, bigInt *rm) {
    mZeroPrime = bigIntInvMon(modulus);
    rSquared = rm;
}
/*--------------------------------------------------------------------------*/
int fiosInnerLoop(uint_16 *a, uint_16 bi, uint_16 *modulus, uint_16 *t, int len, int i) {

    uint_32 sum1, sum2, carry=0;
    uint_16 u;
    uint_64 sum3_64;
    int j, retVal = 0;
    /* First iteration of loop is unrolled so we can calculate u */
    sum1 = (((uint_32) a[0]) * ((uint_32) bi)) + ((uint_32) t[0]);
    u = (uint_16)((sum1 & 0x0000ffff) * ((uint_32) mZeroPrime));
    sum2 = ((uint_32) modulus[0]) * ((uint_32) u);
    sum3_64 = (uint_64) sum1 + (uint_64) sum2;
    carry = (uint_32)(sum3_64 >> 16);
    /* The rest of the loop iterations */
    for (j=1; j < len; j++) {
        sum1 = (((uint_32) a[j]) * ((uint_32) bi)) + ((uint_32) t[j]);
        sum2 = (uint_32) modulus[j] * (uint_32) u;
        sum3_64 = ((uint_64) sum1) + ((uint_64) sum2) + ((uint_64) carry);
        t[j-1] = (uint_16)(sum3_64 & 0x0000ffff);
        carry = (uint_32)(sum3_64 >> 16);
    }
    t[len-1] = (uint_16)(carry & 0xFFFF);
    t[len] = (uint_16)(carry >> 16);
    return(t[len]);
}
/*--------------------------------------------------------------------------*/
void hardware_mulmon(bigInt *a_l, bigInt *b_l, bigInt *modulus_l, bigInt *t_l) {

    uint_16 *a = a_l->value;
    uint_16 *b = b_l->value;
    uint_16 *modulus = modulus_l->value;
    uint_16 *t = t_l->value;
    uint_16 bj;
    int j;
    int my_length;
    /* clear partial product accumulator */
    for (j=0; j < MAX_WORDS; j++) {
        t[j] = 0x0000U;
    }
    t_l->length = LENGTH;
    my_length = LENGTH;
    for (j=0; j<LENGTH; j++) {
        if (fiosInnerLoop(a, b[j], modulus, t, my_length, j))
           my_length++;
    }
}
/*--------------------------------------------------------------------------*/
void modularMultiply(bigInt *A, bigInt *B, bigInt *modulus, bigInt *product) {
    bigInt *aPrime = newBigInt(0);
    hardware_mulmon(A, rSquared, modulus, aPrime);
    /* Sometimes the reduction is not complete */
    if (bigIntGTE(aPrime, modulus))
        bigIntSub(aPrime, modulus);
    hardware_mulmon(aPrime,B,modulus,product);
    /* Sometimes the reduction is not complete */
    if (bigIntGTE(product, modulus))
        bigIntSub(product, modulus);
    deleteBigInt(aPrime);
}
/*--------------------------------------------------------------------------*/
int main(int argc, char **argv) {
    bigInt *a, *b, *c, *m, *rm, *e;
    FILE *aFile;
    int lengthBits;
    int test_number = 1;
    char theString[5001], tempChar;
    char *aString = theString+3;
```

```
aFile = fopen(argv[1], "r");
assert(aFile != NULL);
/* Note that the first line of the file is a seperator */
while (fscanf(aFile, "%s\n", aString) != EOF) {
   /* Size of the numbers in WORDS (16 bit quantities) */
   fscanf(aFile, "n=%s\n", aString);
   sscanf(aString, "%x", &LENGTH);
   /* Modulus "m" */
   m = newBigInt(0);
   fscanf(aFile, "m=%s\n", aString);
   hexString2bigInt(aString, m);
   /* Trap for the special cases where we miscalculate m->length above */
   if (m->length < LENGTH)
      m->length = LENGTH;
   /* First mulitplicand "a" */
   a = newBigInt(0);
   fscanf(aFile, "a=%s\n", aString);
   hexString2bigInt(aString, a);
   /* Second multiplicand "b" */
   b = newBigInt(0);
   fscanf(aFile, "b=%s\n", aString);
   hexString2bigInt(aString, b);
   fscanf(aFile, "k=%s\n", aString);
   /* Length in bits - a sanity check from early testing */
   sscanf(aString, "%x", &lengthBits);
   /* RM as calculated by Mathematica for this modulus */
   rm = newBigInt(0);
   fscanf(aFile, "rm=%s\n", aString);
   hexString2bigInt(aString, rm);
   /* Result "c" */
   c = newBigInt(0);
   setupMontgomery(m, rm);
   modularMultiply(a, b, m, c);
   e = newBigInt(0);
   /* Mathematica's answer: validate our solution */
   fscanf(aFile, "c=%s\n", aString);
   hexString2bigInt(aString, e);
   printf("Test %d ", test_number++);
   if(!bigIntEQ(c, e)) {
      printf(" failed\n");
      printf("Calculated: %s\n",bigInt2HexString(c));
      printf(" Expected: %s\n",bigInt2HexString(e));
   } else {
      printf(" passed\n");
   }
   deleteBigInt(a);
   deleteBigInt(b);
   deleteBigInt(m);
   deleteBigInt(c);
   deleteBigInt(e);
   deleteBigInt(rm);
}
fclose(aFile);
```