VIA Nehemiah Advanced Cryptography Engine

Programming Guide

Version 1.0

This is **Version 1.0** of the VIA Nehemiah Advanced Cryptography Engine Programming Guide.

© 2003 VIA Technologies, Inc. All Rights Reserved.
© 2003 Centaur Technology, Inc. All Rights Reserved.

VIA Technologies and Centaur Technology reserve the right to make changes in its products without notice in order to improve design or performance characteristics.

This publication neither states nor implies any representations or warranties of any kind, including but not limited to any implied warranty of merchantability or fitness for a particular purpose. No license, express or implied, to any intellectual property rights is granted by this document.

VIA Technologies and Centaur Technology make no representations or warranties with respect to the accuracy or completeness of the contents of this publication or the information contained herein, and reserves the right to make changes at any time, without notice. VIA Technologies and Centaur Technology disclaim responsibility for any consequences resulting from the use of the information included herein.

VIA and VIA C3 are trademarks of VIA Technologies, Inc.

CentaurHauls is a trademark of Centaur Technology, Inc..

Intel is a registered trademark of Intel Corporation.

Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

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.

Table of Contents

CHAPTER

1

INTRODUCTION

VIA is providing a suite of security technologies called Padlock in all new processors. The first Padlock technology, introduced in the VIA Nehemiah processor, provided a fast hardware random number generator (RNG) on the processor.

The updated VIA Nehemiah (Stepping 8 and higher) processor core, provides a high performance implementation of the Advanced Encryption Standard (AES), as specified in Federal Information Processing Standards Publication 197 (FIPS-197.) This new Padlock feature is called the *Advanced Cryptography Engine (ACE).*

NOTE: Throughout this document, a reference to encryption generally means both encryption and decryption. For example, "the ACE can encrypt" should be taken to mean "the ACE can encrypt or decrypt". Differences that exist between the encryption and decryption functions of the ACE will be explicitly discussed.

This guide defines the ACE feature as implemented in the newer VIA Nehemiah processor. The detailed CPUID identification of the first VIA processor containing the ACE is:

Vendor ID:	CentaurHauls
Type:	0
Family:	6
Model:	9
Stepping:	8

For support, developers should contact: ace_support@centtech.com.

CHAPTER

2

ARCHITECTURE

2.1 HARDWARE

The VIA Nehemiah Stepping 8 processor Advanced Cryptography Engine (ACE) is implemented in hardware with associated microcode. The hardware performs the encryption functions, and the microcode provides a well-architected x86 instruction interface providing transparent multitasking. This section describes the ACE hardware structure. This information is hidden from the x86 instruction interface and is not necessary for proper use of the ACE.

Figure 1 illustrates the conceptual architecture approach used in adding the ACE. A 128-bit wide *X-unit* is added in parallel with the SSE unit sharing instruction, load, and store buses. The X-unit has a discrete set of internal registers that are not visible to x86 instructions. X-unit instructions are issued, in order, to the SSE unit along with SSE instructions. Once the X-unit instruction is ready for execution, it is dispatched by the SSE unit to the X-unit hardware. Instruction execution in other units, such as integer, MMX, and floating point, can proceed in parallel to X-unit operations. SSE instructions, however, are blocked from execution until the X-unit operation is completed, because all X-unit operations end with storing data to memory. This store must be completed before subsequent SSE instructions can execute.

Figure 1. Adding Security Features to x86 Architecture

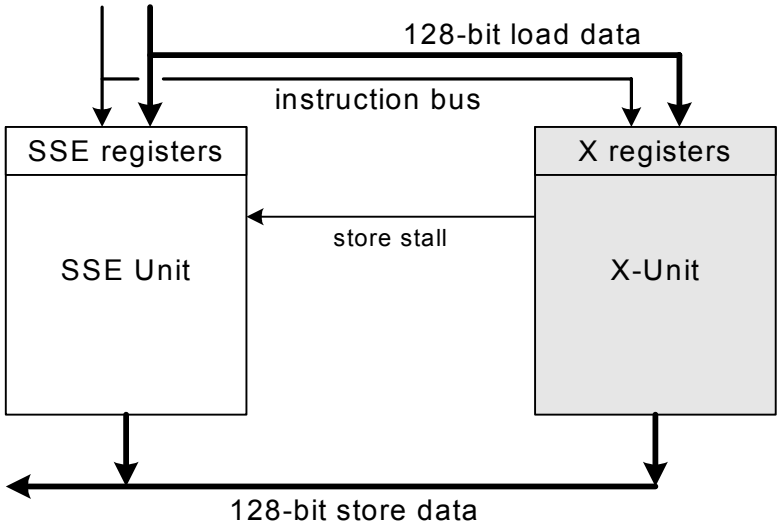
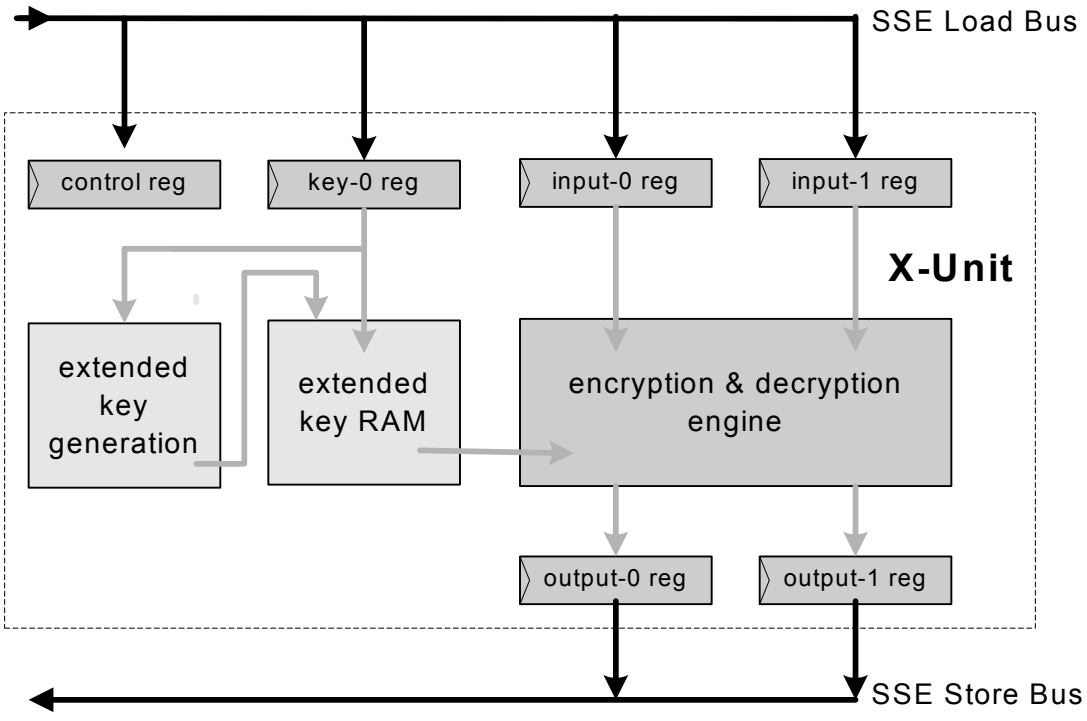


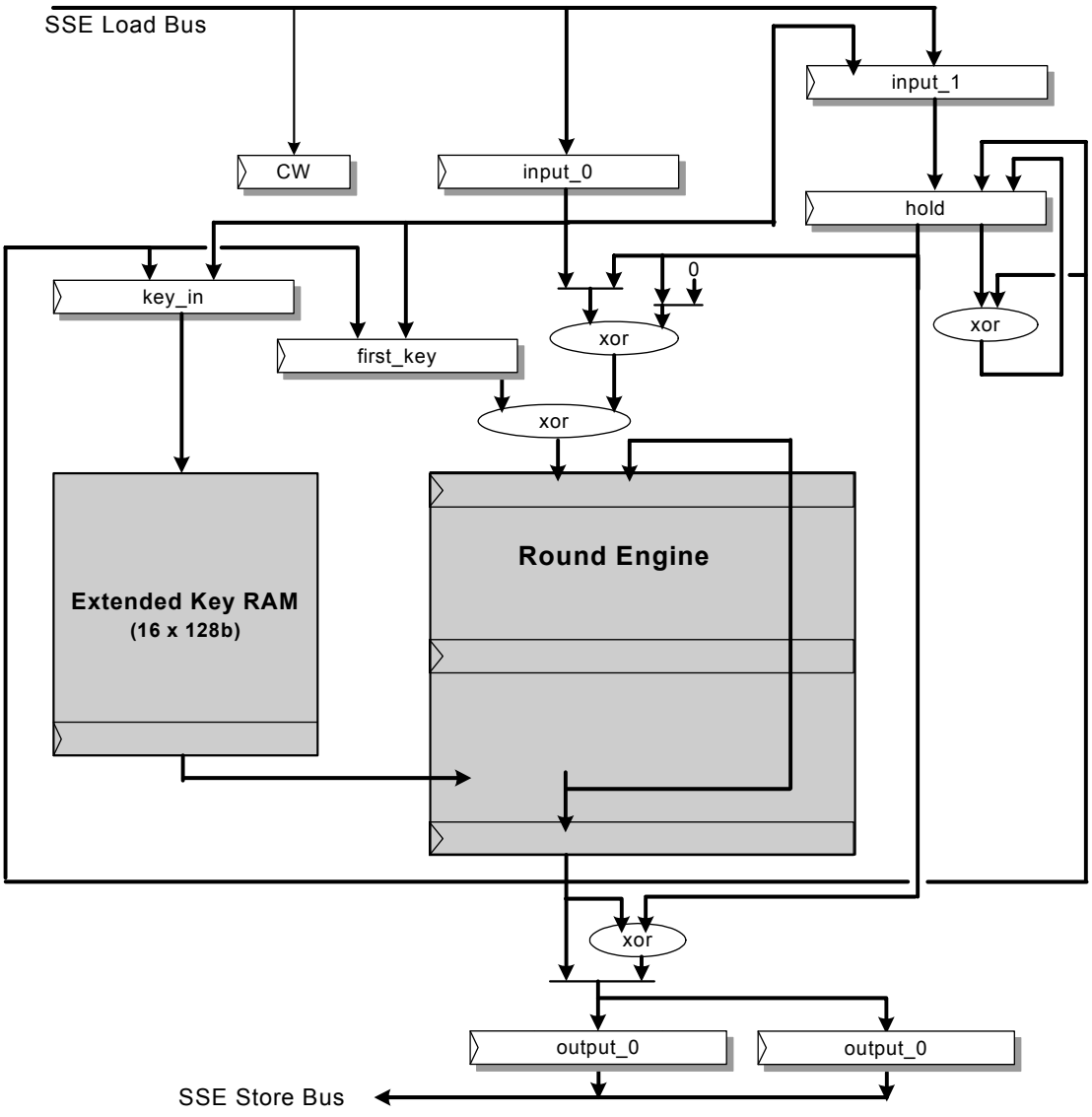
Figure 2 provides a further conceptual breakout of X-unit components. There are four input registers and two output registers. In addition, there is a 128-bit wide, 16-entry RAM that holds the extended keys for encryption. This RAM can be loaded from memory, or by an extended key generation unit that calculates the extended key sequence based on the primary key. The AES encryption engine is pipelined and can operate on two 128-bit data blocks concurrently.

Figure 2. Conceptual X-unit Hardware Overview



The X-unit is more complicated than shown in Figure 2 because the X-unit hardware directly supports several different operating modes. In addition, the extended-key generation hardware is not a distinct component, but rather is built into the encryption logic to minimize distinct circuitry. Figure 3 summarizes the VIA Nehemiah Stepping 8 processor X-unit hardware structure surrounding the round engine, which implements the AES encryption function.

Figure 3. X-unit Hardware



Consider the data path for a typical `xcrypt-ecb` instruction. If key loading is necessary, microcode directs the loading of the key and extended keys are generated into or loaded into the extended key RAM. The RAM is large enough to hold an extended key for 15 rounds plus the initial key. The hardware automatically accesses the appropriate 128-bit extended key sequence needed for each round.

Microcode then loads the second data block into `input_1` register, and the first data block into `input_0` register (ECB mode always works on two data blocks). A load into `input_0` automatically starts the encryption operation. In the first pipeline stage, the first data block is brought from the input 0 register, XOR'd with the first expanded key block (first key), and passed into the AES round engine. The collection of holding registers, muxes, and XORs are used by the various operating modes.

The AES round engine takes two clocks per round, but is pipelined to operate on two blocks at a time. Thus, on the clock after the first data block starts in the round engine, the second data block is XOR'd with the key and sent to the round engine. In our ECB example, the operation provides an effective throughput of one clock per round per 128-bit data block. The results of each round feedback to the next round within the round engine. The appropriate extended key for each round is output from the key RAM to the round engine. After all rounds are completed, the first-block result passes directly to `output_0` register for subsequent storing. The results for the second data block are passed to `output_1` register.

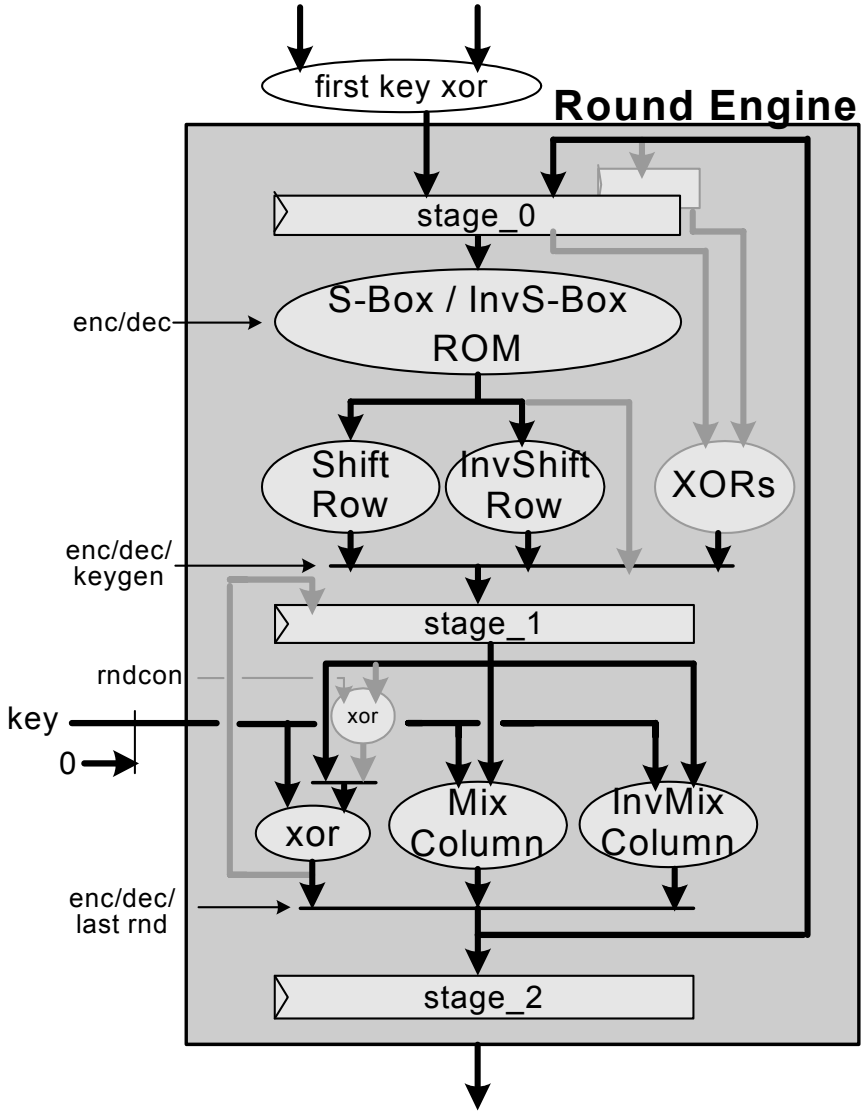
Figure 4 illustrates the internal design of the AES round engine. This logic directly implements the AES algorithm. The lightly drawn lines represent additional logic (added to the base round logic) used to calculate the extended key sequence. Each round, for a single block, requires two pipeline stages.

Encryption is performed as shown in section 5.1 of FIPS-197 as summarized in Figure 5 of that document. The first stage performs the *SubBytes* S-Box substitution followed by the *ShiftRows* operation. The S-Box is implemented as sixteen 2x256x8-bit ROMs (2x for separate encryption and decryption values). The second pipeline stage performs the *MixColumns* operation and the XOR of the extended key. The key XOR is included as another leg in the sequence of XORs performing the mix column function and a bypass of the mix column function exists for the special last-round function.

Decryption is performed using the equivalent inverse cipher algorithm as described in section 5.3.5 and summarized in Figure 15 of FIPS-197. The first stage implements the *InvSubBytes* S-box substitution followed by the *InvShiftRows* operation. The second stage implements the *InvMixColumns* function. This path is the slowest timing path in the round engine due to the increased number of XOR levels over mix column.

The hardware occupies about 0.6 mm^2 on the VIA Nehemiah Stepping 8 processor and runs at the processor clock frequency.

Figure 4. Round Engine Hardware

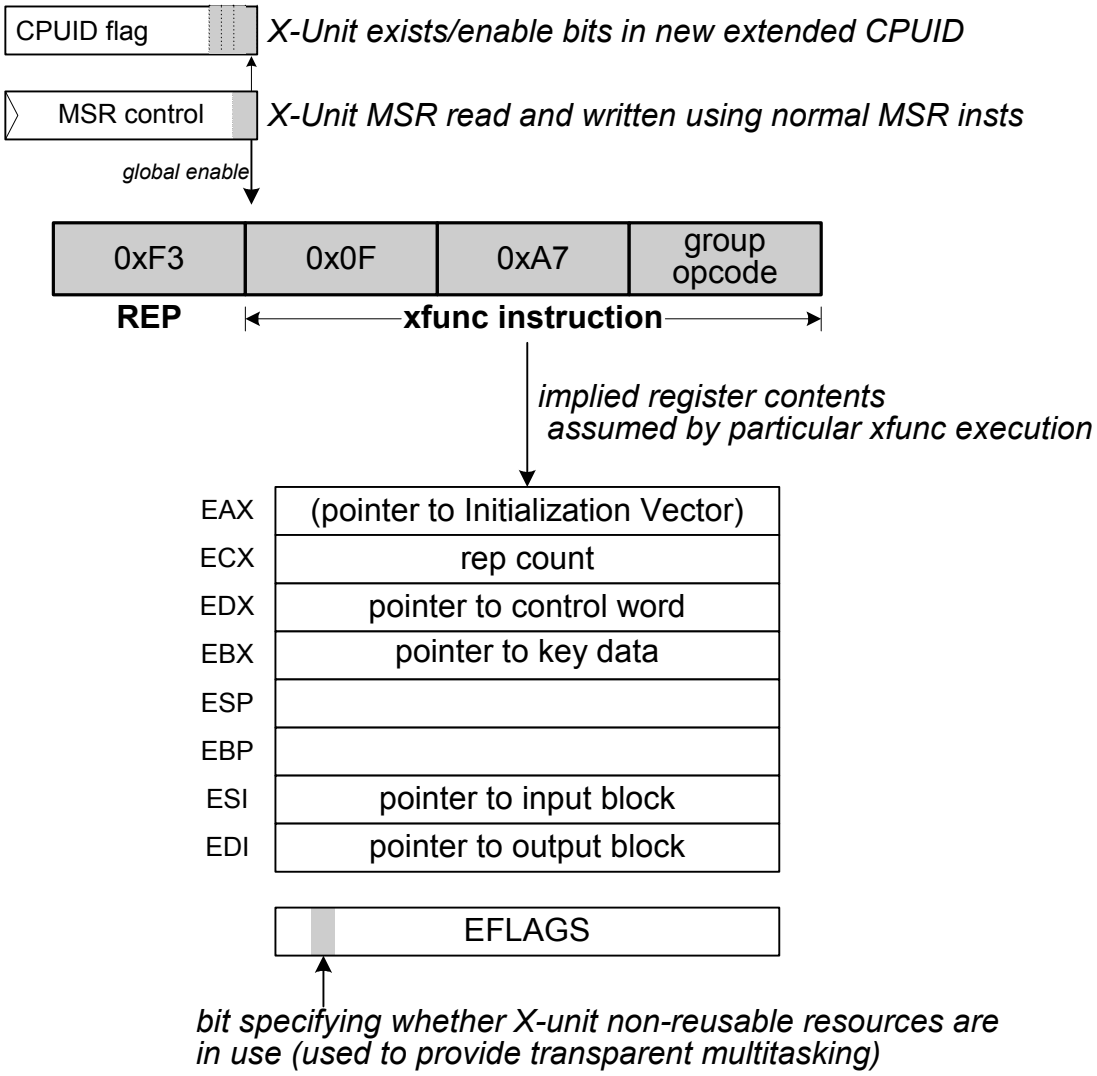


2.2 SOFTWARE

All X-unit functions are accessed using one new x86 primary opcode. This is a group opcode providing eight sub-opcode functions. X-unit instruction operands are defined by multiple memory values whose addresses are contained in general purpose registers. The approach is similar to x86 string instruction formats.

Figure 5 illustrates the instruction architecture and opcodes.

Figure 5. General X-unit Architecture Format

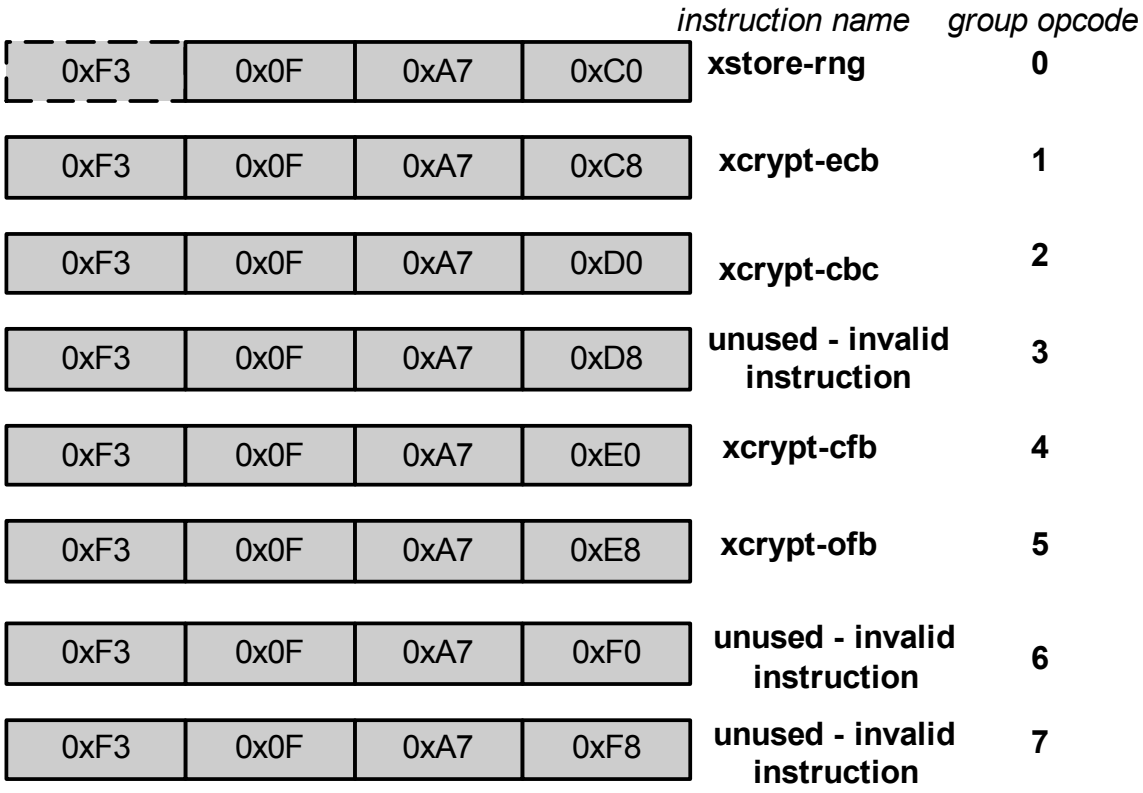


The major components are:

Two bits in the *Centaur extended feature flags* returned by the CPUID instruction. These bits identify (1) whether the ACE physically exists on the processor, and (2) whether it is currently enabled.

- A new bit in the existing FCR MSR that allows a privileged program to enable or disable the ACE.
- One non-privileged x86 opcode that provides the VIA Nehemiah Stepping 8 processor cryptographic features. This is a group format opcode that defines eight group instructions, as shown in **Figure 6**

Figure 6. X-unit Instruction Functions



The `xcrypt` instructions are atomic with respect to the encryption of a single data block (two data blocks in ECB mode); that is, once execution starts, the specified function completes each block before interrupts are allowed.

All `xcrypt` instructions exist in `REP` form only. Like other `REP` x86 instructions, they can be interrupted and restarted. They differ from other `REP` instructions in that the performance of encrypting N blocks of data with a `REP xcrypt` instruction is faster (as much as 50%) than the execution of N separate `xcrypt` instructions.

The `xcrypt` instructions define their operands in x86 general purpose registers.

An EFLAGS register extension in Bit 30, previously unused, allows multiple tasks to use `xcrypt` instructions in a fashion transparent to applications and to the operating system.

Figure 7 shows the detailed format of the `REP xcrypt-ecb` instruction. This performs encryption using the Electronic CodeBook (ECB) mode. The basic steps to use the instruction: [**Note: All addresses assume the ES segment selector, which may not be overridden**]

1. Check the Centaur extended CPUID flags to see if the ACE is present.
2. If necessary, enable the ACE using the FCR MSR. This is a privileged operation but rarely needs to be done, as the default state at RESET is that ACE is enabled.

3. Set bit 30 of EFLAGS to 0. This is interpreted by microcode to mean that the control word and extended keys provided by the subsequent `REP xcrypt-ecb` instruction are new to this task and thus need to be loaded into internal hardware registers. This can be done by a PUSHF instruction followed by a POPF instruction. (*Any* load of EFLAGS zeroes bit 30). **Be sure to do this whenever you use a new key.**

4. Load the effective address (offset) of either (1) the entire extended key sequence, or (2) the key itself into EBX. A bit in the control word indicates the distinction between these two key-extension options.

5. Load the effective address (offset) of the desired ACE control word into EDX. The control word specifies encryption or decryption, the number of rounds, where the extended key comes from, and so forth.

6. Load the effective address (offset) of the first data block to be encrypted into ESI.

7. Load the effective address (offset) of the start of the ciphertext result area into EDI. **The source data address and the result data address may be identical; the ACE supports encryption in place**.

8. Load the number of 128-bit data blocks to be encrypted into ECX. Note that each execution of the `REP xcrypt-ecb` function processes two data blocks. When ECX is odd, one extra block will be encrypted but the result *will not be stored to memory*.

9. Execute the `REP xcrypt-ecb` instruction, which performs the following steps:
   a. If ECX==0, continue at step 14: the REP instruction is complete.
   b. If EFLAGS:30 is 0, the extended keys and control word are loaded into internal hardware registers.
   c. EFLAGS:30 is set to 1.
   d. *Two* 128-bit input data blocks are loaded from memory into the hardware.
   e. The encryption operation starts.

10. The single encryption operation continues until the specified number of rounds has been completed, or an addressing exception is encountered. Each round takes two processor clocks. When processing of all rounds is complete, the result is placed in the appropriate memory location and both ESI and EDI are incremented by the size of the data stored (usually 32 bytes, but only 16 when ECX is equal to 1).

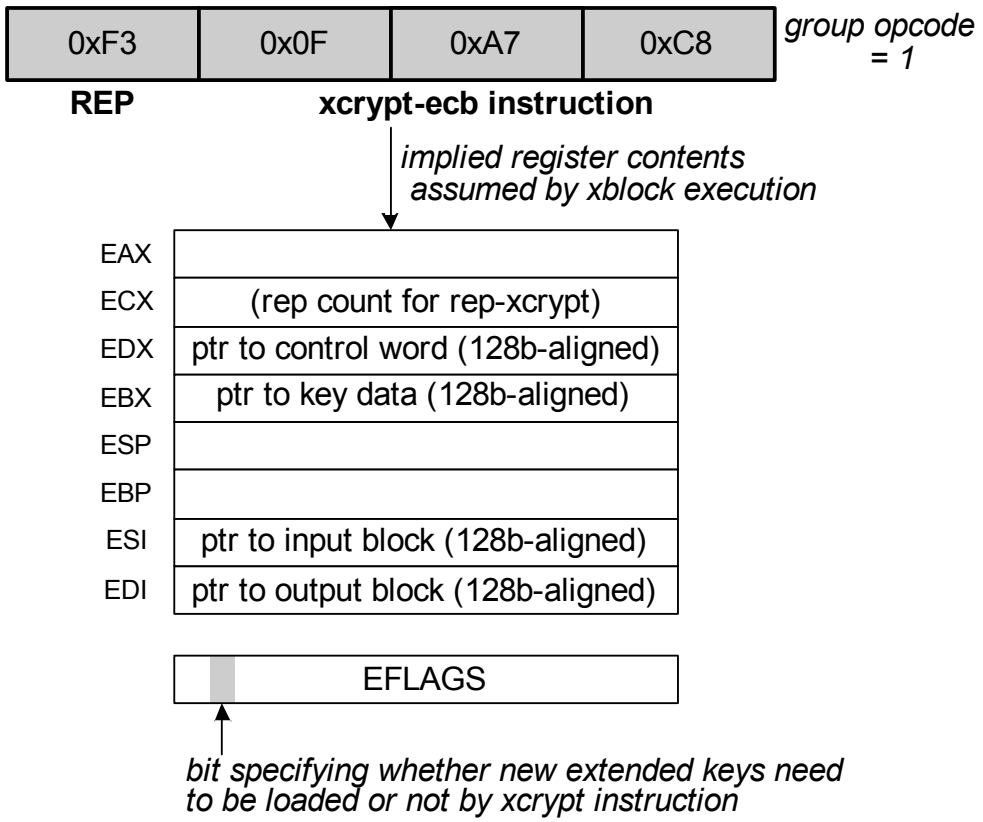
11. If an addressing exception occurs during the execution of `REP xcrypt-ecb` (for example, a page fault), the interrupted EIP and the register state saved by the exception allows the `REP xcrypt-ecb` instruction to be re-executed with correct results.

12. If the single operation has completed successfully, two is subtracted from ECX (except if ECX=1, when only 1 is subtracted) and execution continues at step 9 or at step 13 .

13. Every so often, after an encryption operation has been completed, the REP `xcrypt-ecb` instruction allows external interrupts to occur. If the execution is interrupted, the interrupted EIP and the register state saved by the exception allows the `REP xcrypt-ecb` instruction to be re-executed continuing the original sequence of encryptions.

14. Execution of the `REP xcrypt-ecb` instruction is now complete and interrupts are allowed before execution of the next x86 instruction starts.

Figure 7. `xcrypt-ecb` Instruction



2.3 MULTI-TASKING

The ACE architecture is specifically designed to address two major (and related) goals:

To allow multiple applications (including the operating system) to use the `xcrypt` functions without any operating systems support (such as a device driver), and

To allow multiple applications (including the operating system) to use the `xcrypt` functions without any awareness of, or visibility of, other tasks (and their data) that are also using `xcrypt` functions.

That is, if

(1) Task A executes an `xcrypt` instruction, and

(2) A task switch subsequently occurs to task B that

(3) Subsequently executes an `xcrypt` instruction, the following must be true:

- Any inter-instruction state needed to later restart task A and execute subsequent task A `xcrypt` instructions must be saved (when task A is suspended) and restored (when task A is restarted) by *existing* operating system code, and
- Any inter-instruction state required to execute task B's `xcrypt` instructions correctly must be restored (when task B is started) by *existing* operating system code, and

- Task B must not be able to see, touch or smell anything (such as the keys) relating to task A's use of the `xcrypt` instruction, and vice versa.

The ACE approach to satisfy these requirements is to make `xcrypt` self-contained. That is, all of the information needed to perform an encryption is contained within the `xcrypt` instruction by using pointers in GPRs 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 about as long as 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 `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 an `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 an `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 `xcrypt` behavior, the VIA Nehemiah Stepping 8 processor implements 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)

EFLAGS:30 cannot be set to 1 by any x86 instruction that causes EFLAGS to be loaded. Only `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):

Task 1 executes a first `xcrypt` instruction. Since EFLAGS:30 is 0, the `xcrypt` logic loads the control word and extended keys into the hardware, and sets EFLAGS:30 to 1.

Task 1 executes a second `xcrypt` instruction. Since EFLAGS:30 is 1, the `xcrypt` logic bypasses loading the control word and extended keys. This improves the performance of the second `xcrypt` instruction.

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 done by the built-in x86 exception or task switch mechanism. Regardless, this load causes task 2's EFLAGS:30 to be set to 0.

Task 2 executes an `xcrypt` instruction. Since its EFLAGS:30 is 0, the `xcrypt` logic loads the control word and extended keys into the hardware, and sets EFLAGS:30 to 1.

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 `xcrypt` execution will reload the task 1 control word and keys.

2.4 EXTENDED KEYS

An `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. The 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 all specifiable round counts. This generation is performed as a side effect of an `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.

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 an `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.

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 (128-bit key), 23 (192-bit key), or 31 (256-bit key).

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.)

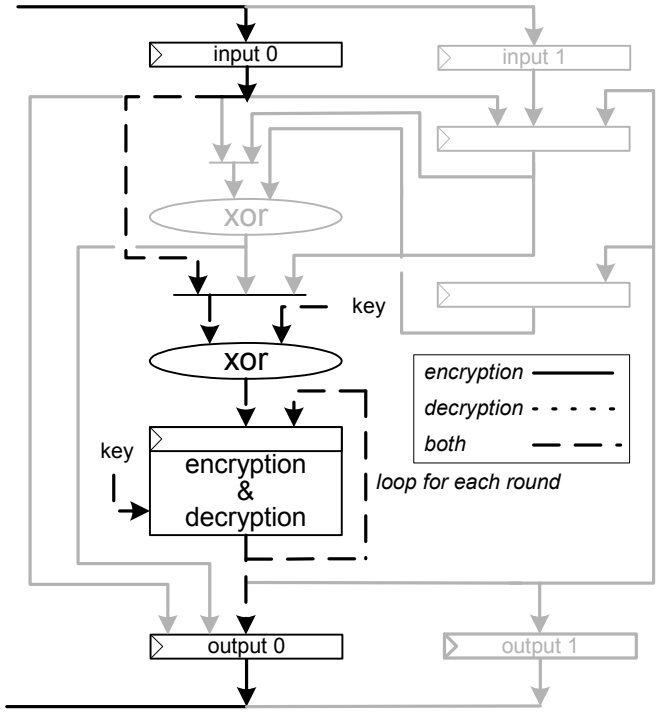
2.5 MODES OF OPERATION

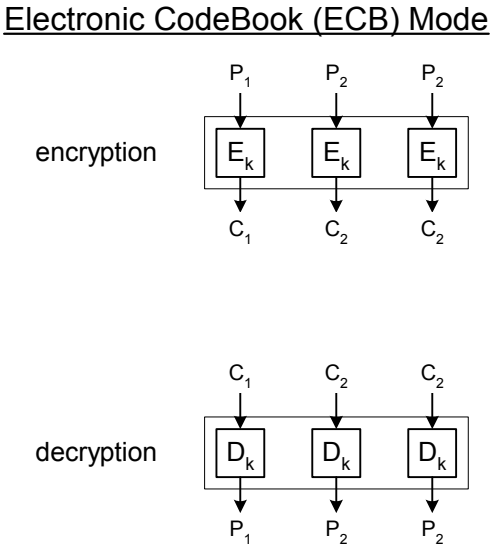
The supported modes are implemented by a collection of datapaths surrounding the basic AES round engine. The application program does not directly control these paths; instead, the application program merely specifies the operating mode to be used and the hardware manages the routing of data.

ECB MODE

In this mode, each data block is encrypted independently from other blocks. Two blocks are pipelined through the round engine. One block comes from input_0 register, the other from input_1 register (via the hold register). 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.

Figure 8. ECB Operating Mode Support





CBC MODE

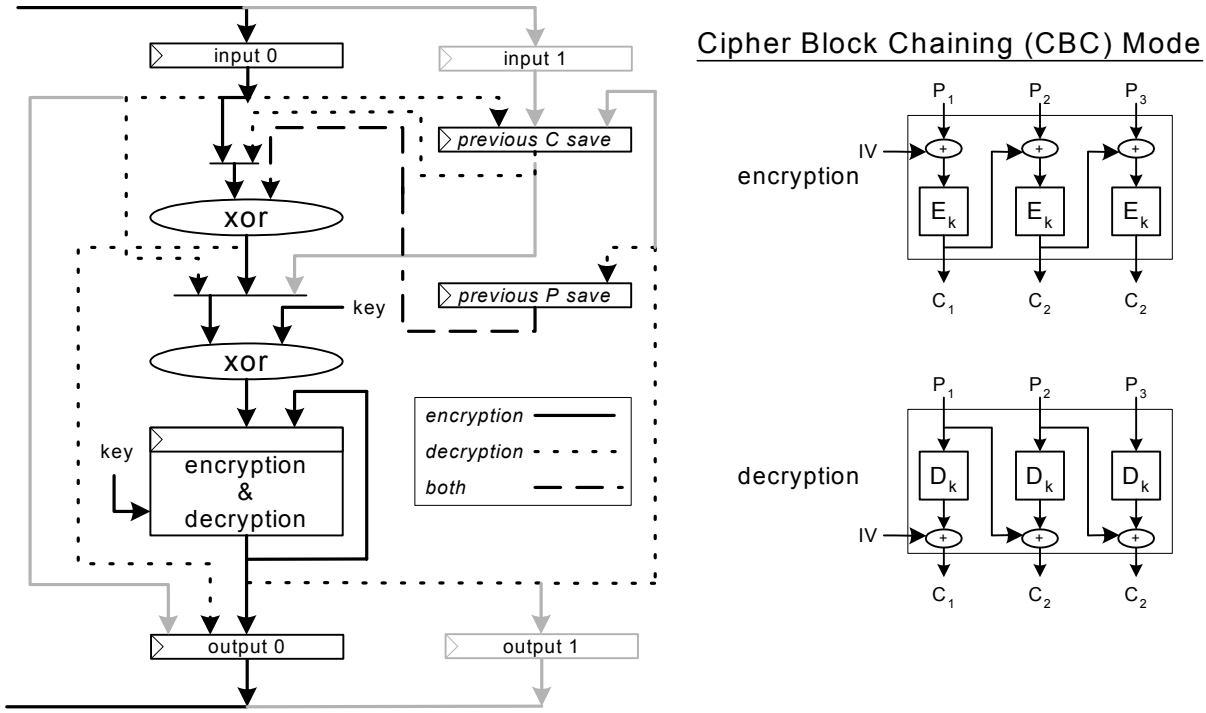
In this mode, the result of each encryption is XOR'd with the next incoming data block. This cipher block chaining (forwarding) operation insures that any single block of cipher text cannot be decrypted to plain text; only the entire file can be decrypted. This forwarding means that only one block is 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.

Unlike EBC, encryption and decryption have different routings through the X-unit.

For CBC **encryption**, the plain text comes from the *input_0* register, is then XOR'd with either the IV or the forwarded cipher text results from the last block (from the *hold* register), and the results proceeds to the key XOR and the round engine. The encryption results are sent to the output register and the hold register for use on the following encryption.

For CBC **decryption**, the incoming cipher block is passed on to the key XOR and the round engine for decrypting. The incoming cipher block is also moved into the hold register for subsequent XOR'ing with the results of the round engine operating on the next subsequent cipher block.

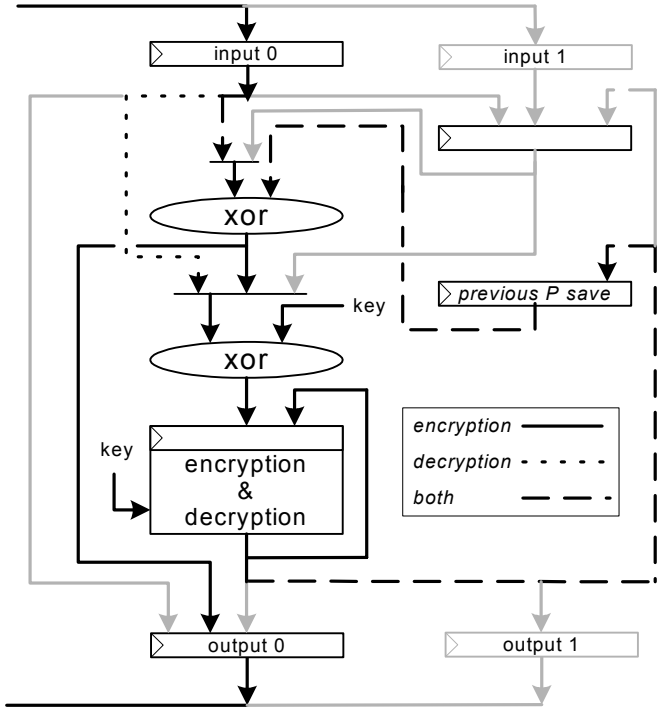
Figure 9. CBC Operating Mode Support

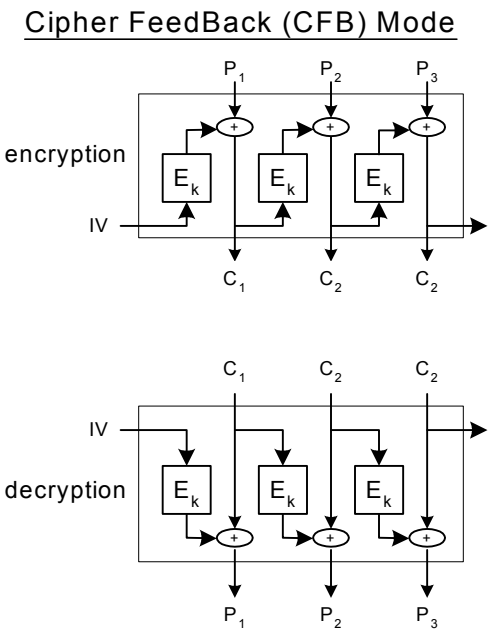


CFB MODE

For encryption, the IV is encrypted from the *hold* register, and then P_1 in routed from *input_0* register to the *hold* register. The final round results are XOR'd with P_1 in xor-b, with the result sent to the output, and in xor-b, with the result placed back in the *hold* register to feed the next encryption. Decryption is similar, except only xor-b is used and C_n is saved in the hold register for the subsequent cycle.

Figure 10. CFB Operating Mode Support

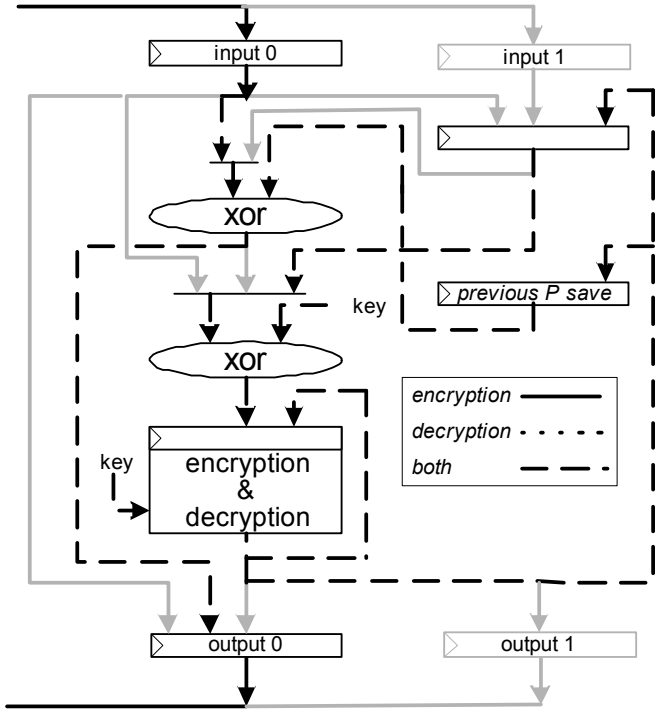


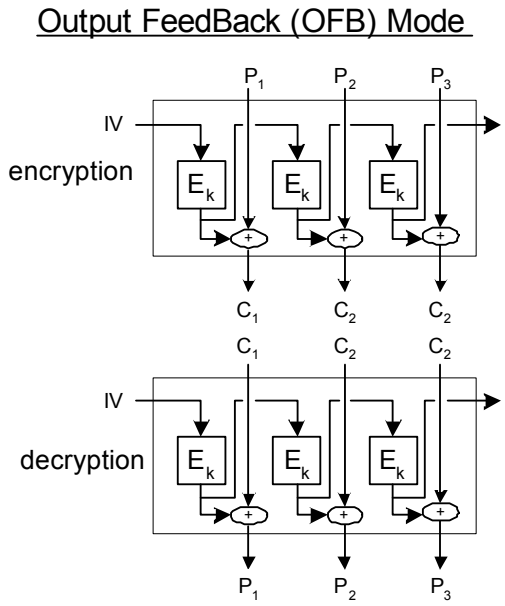


OFB MODE

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. A "one-time pad". This sequence is XOR'd with the plaintext to produce the ciphertext, or with the ciphertext to reconstruct the plaintext. It is thus critical that no 128-bit block of the encryped sequence be re-used.

Figure 11. OFB Operating Mode Support





CHAPTER

3

x86 INSTRUCTIONS

3.1 PARAMETERS

MEMORY FORMATS

All X-unit 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).

EDX: THE CONTROL WORD POINTER

DX or EDX in all `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.

The control word fields are:

127:32	31:12	11:10	9:9	8:8	7:7	6:4	3:0
Reserved	Reserved 0	Key size	Decrypt/ encrypt	Intermediate/ normal	key-generation type	algorithm type	round count

Bits 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 is specified by the AES, this standard value is not built into the hardware; the application program must specify the number of rounds.

Bits 6:4 Algorithm Type: This three-bit field specifies the cryptographic algorithm. The only algorithm currently supported is the AES, which has a field value of 0. Other values are reserved and the ACE behavior is undefined if this field is set to reserved values.

Bit 7 Key Generation Type: 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).

Bit 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.

Bit 9 Encrypt/Decrypt: This one-bit field specifies encryption (0) or decryption (1).

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

Bits 31:12 Reserved: These bits should be set to 0 and the contents of this field should not be used by the application.

Bits 127:32 Reserved: Although these bits are not used as part of the control word, they are modified during key generation. While it is possible for an application to use them as temporary scratch buffers, this use is not recommended.

ESI: SOURCE DATA POINTER

SI or ESI in all `xcrypt` instructions contains the effective address (offset) of the input 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.

At the end of instruction execution, or if an interrupt (or page fault, etc.) has occurred, SI or ESI 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.

EDI: RESULT DATA POINTER

DI or EDI in all `xcrypt` instructions contains the effective address (offset) of the result data generated by the instruction function. 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.

At the end of instruction execution, or if an interrupt (or page fault, etc.) has occurred, DI or EDI 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.

EBX: KEY POINTER

BX or EBX in all `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:

- Bit 7 = 0: The primary key. The extended key sequence will be generated by the hardware.
- Bit 7 = 1: The primary key followed by the application calculated extended key sequence.

ECX: REP COUNT

CX or ECX in all `rep xcrypt` instructions contains the count of 128-bit data blocks to be processed.

At the end of instruction execution, CX or ECX 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.

EAX: INITIALIZATION VECTOR

AX or EAX in chaining mode `rep xcrypt` instructions (all modes except ECB) contains the effective address (offset) of the initialization vector (IV). 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 IV is not the right one to use upon instruction restart. So the `xcrypt` logic changes either the address in AX/EAX, or changes the data value pointed to by AX/EAX to be the appropriate IV for the continuing 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.*

3.2 REP

The rep `xcrypt-` instructions have similarities with the x86 `movs` instruction:

The value in EDI is updated with the number of result bytes stored.

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.

And there are some different semantics:

- 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 an Invalid Instruction exception.
- 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.

CHAPTER

4

USING THE ACE

4.1 ENABLING

To use the VIA Nehemiah Stepping 8 processor ACE, three conditions must be met:

The ACE must physically exist on the chip. Its existence can be discovered from the Centaur Extended CPUID Feature Flags. This condition is permanently set as part of the manufacturing process.

The ACE must be enabled. The enabled state can be also discovered from the Centaur Extended CPUID Feature Flags. The ACE can be enabled/disabled by writing to the Function Control Register (FCR), MSR 0x1107. The RESET default is enabled. But, see the next condition.

SSE instructions must be enabled via the standard x86 method of enabling the FXSAVE/FXRSTOR instructions using CR4[9] This CR4 enabling also enables the full set of SSE instructions on Intel processors and VIA Nehemiah Stepping 8 processors.

The VIA Nehemiah Stepping 8 processor supports the Centaur Extended Feature Flags. When the CPUID instruction is executed with EAX=0xC0000000, the processor returns 0xC0000001 in EAX. Note that earlier VIA C3 Nehemiah processors do not support the Centaur Extended CPUID Functions and will return EAX=0xC0000000 or EAX=0x00000000 (see the VIA Nehemiah processor datasheet for details on the extended CPUID function).

If a CPUID with EAX=0xC0000000 returns a value in EAX >= 0xC0000001 then Centaur Extended Feature Flags are supported. A CPUID with EAX=0xC0000001 then returns the Centaur Extended Feature Flags in EDX. There are two bits in EDX that describe the ACE:

EDX[6] == 0 ACE does not exist on this chip. FCR[28] cannot be set to enable the ACE and the new `xcrypt` instructions cause an Invalid Opcode Fault.

EDX[6] == 1 ACE exists. The behavior of `xcrypt` instructions is dependent upon whether the ACE is enabled or not.

EDX[7] == 0 ACE is disabled. The new `xcrypt` instructions cause an Invalid Opcode Fault. If the ACE is present, FCR[28] can be set to enable the ACE

EDX[7] == 1 ACE is enabled (usually the RESET default. The `xcrypt` instructions behave as defined in this application note (provided CR4[9] =1)

All VIA C3 Nehemiah processors implement a Function Control Register (FCR), MSR 0x1107, which can be written by software to enable or disable various features. One bit controls the ACE - FCR[28]:

0=The ACE is not enabled. If EDX[6] indicates that ACE is present, a WRMSR can set this bit to 1 thus enabling the ACE.

1=The ACE is enabled. A WRMSR can set this bit to 0 thus disabling the ACE.

4.2 STARTUP

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 ACE actions are performed:

Any in-progress ACE instruction is immediately cancelled with undefined results being stored.

The FCR is reset to its default value. This may or may not directly change the ACE enable status.

The CR4 register is reset to zero. Since this resets the FXSAVE/FXRSTOR enable, (CR4[9]) the ACE is indirectly disabled.

EFLAGS:30 is cleared to zero.

The INIT signal is an interrupt that occurs between x86 instructions: INIT never halts instructions in the middle of their execution (except for REP string instructions, where INIT behaves like INTR). The ACE effect of INIT is:

The CR4 register is reset to zero. Since this resets the FXSAVE/FXRSTOR enable, (CR4[9]) the ACE is indirectly disabled.

EFLAGS:30 is cleared to zero.

MANUFACTURED STATE

The ACE is tested as part of the VIA Nehemiah Stepping 8 processor manufacturing process. Relative to potential fault coverage, however, this level of testing is not 100% complete. For certain sensitive applications, additional testing (by the application) may be desirable.

CHAPTER

5

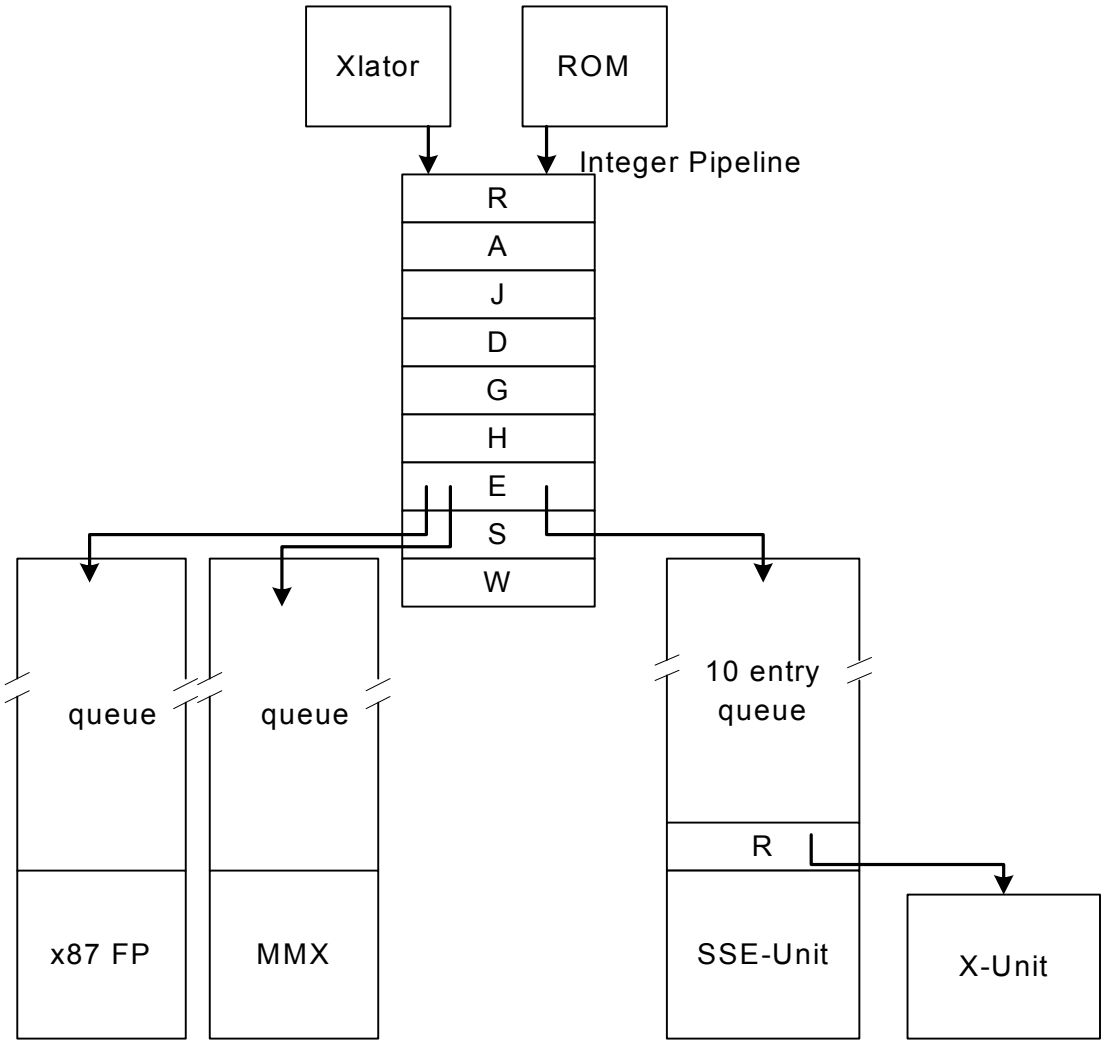
PERFORMANCE

As for all x86 instructions that reference memory, the timing of `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.

X-unit instruction timing is more complicated in that a large number of `xcrypt` execution clocks are overlapped with subsequent instruction execution clocks. To understand this, Figure 12 is a simple illustration of the VIA Nehemiah Stepping 8 processor pipeline structure (as it affects the ACE). x86 instructions are fetched and translated by the instruction fetch front-end stages (not shown). Internal micro-instructions are provided to the execution pipeline from either the x86-instruction translator or the associated micro-instruction ROM.

All types of microinstructions proceed through the integer pipeline. Media-type instructions (x87 floating-point, MMX, and SSE instructions) are passed from the E-stage of the integer pipeline to instruction and data queues feeding the appropriate media-execution engines. Once a SSE micro-instruction (for example) is placed in the SSE media queue, it can no longer cause an exception and the integer pipeline and other media units are free to continue executing other instructions while the SSE micro-instruction percolates down the queue. Thus, the execution of X-unit instructions (handled as SSE instructions) is overlapped with the execution of subsequent integer, MMX, or floating point instructions.

Figure 12. VIA Nehemiah Stepping 8 Execution Pipeline Overview



As an example, referring to the first entry in Table 1, we see that a rep `xcrypt-ecb` 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 `xcrypt-ecb` instruction. If the rep `xcrypt-ecb` instruction is followed by integer, MMX, or floating point instructions, however, the rep `xcrypt-ecb` instruction only appears to take 12 clocks total relative to these instructions. The missing 19 clocks of X-unit execution are overlapped with 19 clocks of execution of non-SSE instructions.

In the instruction timing tables below, the following terms are used:

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, the 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 the 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. It also means that the REP xcrypt instructions can be used as long NOP instructions if you need a delay in your program.

When ECX = 0, REP xcrypt instructions will between 22 and 27 clocks, depending on the state of the ELFAGS:30 bit 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 the X-unit. At the completion of the cipher operation, there will be a slight overlap where the integer unit can fetch subsequent instructions while the ACE completes the final block. This overlap is comparable to the extra clocks available when doing a single encryption or decryption.

ECB MODE XCRYPT			SSE Clocks		Integer Clocks	
Data Size	**EFLAGS bit 30**	**Extended Key Source**	**Base**	**Variable**	**Base**	**Variable**
128b	1		9	2*R	17	0
128b	0	Hardware	45	2*R	61	0
128b	0	Memory	76	2*R	84	0
2*128b	1		11	2*R	12	0
2*128b	0	Hardware	47	2*R	56	0
2*128b	0	Memory	78	2*R	79	0
N*128b	1		11	N*R		
N*128b	0	Hardware	47	N*R		
N*128b	0	Memory	78	N*R		

Notes

The REP `xcrypt-ecb` 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 since only the first block is written to memory.

CBC/CFB MODE XCRYPT			SSE Clocks		Integer Clocks	
Data Size	EFLAGS bit 30	Extended Key Source	Base	Variable	Base	Variable
128b	1		7	2*(R+1)	23	0
128b	0	Hardware	51	2*(R+1)	65	0
128b	0	Memory	72	2*(R+1)	88	0
N*128b	1		7	2*N*(R+1)		
N*128b	0	Hardware	51	2*N*(R+1)		
N*128b	0	Memory	72	2*N*(R+1)		

Notes

Add 11 clocks to the integer timing for DECRYPT of a single block.

Add 5 clocks to the base integer timing for XRCYPT-CFB

OFB MODE XCRYPT			SSE Clocks		Integer Clocks	
Data Size	EFLAGS bit 30	Extended Key Source	Base	Variable	Base	Variable
128b	1	n/a			48	0
128b	0	Hdw			90	0
128b	0	Appl			113	0
N*128b	1	N/a			4*R-26	45*N
N*128b	0	Hdw			4*R-26	45*N
N*128b	0	Appl			4*R-26	45*N

Notes

In OFB mode, a new IV must be calculated, and stored, for each 128-bit block processed. Thus the SSE clocks are fewer than the Integer clocks and are not separately recorded.

CHAPTER

6

VERIFICATION

Cryptography Research, Inc., of San Francisco has performed an independent evaluation of the ACE unit. A copy of their report will be available on the VIA website or ask a VIA representative for a copy.

6.1 ERRATA

Some functions documented in this note do not work properly in the current VIA Nehemiah processor stepping 8:

- The ability to generate extended keys for key sizes of 196- and 256-bits is not working. Attempts to use this function will cause incorrect and unpredictable results. Hardware extended-key generation for 128-bit keys works correctly.
- Intermediate round results for decryption require that the user provide the extended key schedule.

These errata may be fixed in later steppings.