PadLock Quick Reference

25th July 2008

LIFE SUPPORT POLICY

DETECTING PADLOCK

Issue **CPUID** with **EAX** = 0xC0000000. If it returns with **EAX** >= 0xC0000001, then a subsequent **CPUID** instruction with **EAX** = 0xC0000001 will return Centaur Extended Feature Flags in **EDX**, of which:

- **EDX**:2 - **RNG** instructions are present
- **EDX**:3 - **RNG** instructions are enabled
- **EDX**:6 - **AES** instructions are present
- **EDX**:7 - **AES** instructions are enabled
- **EDX**:10 - **SHA** instructions are present
- **EDX**:11 - **SHA** instructions are enabled

NOTE ON TERMINOLOGY

In the instruction descriptions on the following pages, all examples are given with 32-bit registers (**EAX**, etc). PadLock instructions in VIA Nano processors work correctly in 16-bit and 64-bit modes, and register usage should be modified accordingly (**AX** or **RAX**, etc).

RANDOM NUMBER GENERATION

Instructions:

XSTORE - 0F A7 C0 - store random bytes

REP XSTORE - F3 0F A7 C0 - store rep count random bytes

Register Usage: Input

ECX	For **REP XSTORE** only, contains the number of bytes to be stored
EDX	Entropy factor
EDI	Pointer to the memory buffer where the random bytes are stored. Assumes **ES** segment. The memory must be writable

Register Usage: Output

EAX	For **XSTORE** only, contains the number of bytes stored (0-16) in bits 4:0. All other bits are undefined for both **XSTORE** and **REP XSTORE**.
ECX	For **REP XSTORE**, **ECX** will be zero. Otherwise unchanged.
EDX	Bits 0:1 are unchanged, all higher order bits are zero.
EDI	Incremented by the number of bytes stored. The memory will contain the random bits generated by the PadLock hardware.

Notes:

If **ECX** is initially 0 for **REP XSTORE**, **EAX** and **EDX** are not modified.

If insufficient bits have accumulated in the hardware buffers, **XSTORE** will store no data. **REP XSTORE** will execute until the number of requested bytes are stored.

The **ES** segment for **EDI** may not be overridden with another segment prefix. Any such prefix will be ignored.

REP XSTORE is interruptible. As for other x86 **REP** instructions, the registers and memory are modified so that the instruction can continue execution, after the interrupt is serviced, in a manner transparent to the software.

EDX specifies the entropy of the output bytes:

- **EDX** = 0 : raw bits as generated by the hardware are stored.
- **EDX** = 1, 2, or 3 : raw bits are grouped into two blocks of 16 bytes, block 1 = **AES** key, block 2 = **AES** plaintext, and the first 8 bytes of **AES ECB** mode ciphertext are stored. This provides more entropy per bit and good statistical qualities.

ADVANCED ENCRYPTION STANDARD

Instructions:

REP XCRYPTECB - F3 0F A7 C8 - encrypt/decrypt **REP** count 16-byte blocks using electronic code book

REP XCRYPTCBC - F3 0F A7 D0 - encrypt/decrypt **REP** count 16-byte blocks using cipher block chaining

REP XCRYPTCTR - F3 0F A7 D8 - encrypt/decrypt **REP** count 16-byte blocks using counter mode

REP XCRYPTCFB - F3 0F A7 E0 - encrypt/decrypt **REP** count 16-byte blocks using cipher feedback

REP XCRYPTOFB - F3 0F A7 E8 - encrypt/decrypt **REP** count 16-byte blocks using output feedback

Register Usage: Input

EAX	For all **XCRYPT** instructions except **XCRYPTECB**: pointer to the Initialization Vector. Assumes **ES** segment. The memory must be writable.
EBX	Pointer to the Encryption Key. Assumes **ES** segment.
ECX	Number of 16-byte blocks to encrypt or decrypt
EDX	Pointer to the Control Word. Assumes **ES** segment
ESI	Input pointer. Assumes **ES** segment. When encrypting this is the plaintext, when decrypting it is the ciphertext
EDI	Output pointer. Assumes **ES** segment. When encrypting this is the ciphtertext, when decrypting it is the plaintext. The memory must be writable.

Register Usage: Output

EAX	The register is unchanged. The memory will contain the updated Initialization Vector, to be used with the next sequential input
EBX	Unchanged (register and memory)
ECX	0
EDX	Unchanged (register and memory)
ESI	Incremented by the number of bytes encrypted or decrypted. Memory unchanged.
EDI	Incremented by the number of bytes encrypted or decrypted, except when the Control Word specifies Message Authentication Code. The memory will contain the ciphertext (for encryption) or plaintext (for decryption) of the input.

Notes:

The control word fields are:

	Key Size	Encryption Mode	Intermediate Mode	Key Mechanism		Message Authentication Code	Number of Rounds
127:12	11:10	9	8	7	6:5	4	3:0
0	00 - 128 bits 01 - 192 bits 10 - 256 bits	0 - encrypt 1 - decrypt	0 - off 1 - on	0 - hardware generate 1 - load from memory	0	0 - off 1 - on	10 - 128b keys 12 - 192b keys 14 - 256b keys

The plaintext and ciphertext buffers may point to the same memory location.

The **ES** segment for **EAX, EBX, EDX, ESI,** and **EDI** may not be overridden with another segment prefix. Any such prefix will be ignored.

For **XCRYPTCTR**, the high-order 16 bits of the Initialization Vector will be incremented by one for every block processed. This counter is big-endian, compatible with the counter mode for Wireless specification 801.11i

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.

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

Message Authentication Code (MAC) is valid for **XCRYPTCBC** and **XCRYPTCFB** instructions only. This bit is ignored by all other **XCRYPT** instructions.

REP XCRYPT instructions are interruptible. As for other x86 **REP** instructions, the registers and memory are modified so that the instruction can continue execution, after the interrupt is serviced, in a manner transparent to the software

FIPS 197 specifies the indicated number of rounds for each of the supported key sizes. The **XCRYPT** instructions will operate on any number of rounds from 1 to 16 (specifying 0 rounds actually performs 16 rounds - think of it as an assembly language loop counter). For correct **AES** calculations, the number of rounds must be set as per the **FIPS** 197 specification.

SECURE HASH ALGORITHM

Instructions:

REP XSHA1 - F3 0F A6 C8 - Calculate **SHA1** as specified by **FIPS** 180-2

REP XSHA256 - F3 0F A6 D0 - Calculate **SHA256** as specified by **FIPS** 180-2

Register Usage: Input

EAX	[0] the **XSHA** instruction should perform the **FIPS** 180-2 specified padding of the input stream, and treat the **REP** count in **ECX** as a byte counter. [-1] the **XSHA** instruction does not perform the **FIPS** 180-2 padding, and treats the **REP** count in **ECX** as the number of 64-byte blocks of input stream on which to perform the **SHA** calculation.
ECX	Size of the input stream, in bytes (**EAX** = 0) or in 64-byte blocks (**EAX** = -1)
ESI	Pointer to the input stream. Assumes **ES** segment.
EDI	Pointer to a memory buffer with the initial hash constants. Assumes **ES** segment. Must be writable.

Register Usage: Output

EAX	If the input value was zero, it will contain the original **ECX** value. Otherwise it will be unchanged.
ECX	If the input value of **EAX** was zero, **ECX** will be unchanged. If the input value of **EAX** was -1, **ECX** will be zero.
ESI	Incremented by the number of bytes processed. Memory unchanged. If input **EAX** = 0, the increment is the input value in **ECX** If input **EAX** = -1, the increment is the input value in **ECX** * 64
EDI	Unchanged. The memory pointed to will contain the result of the secure hash calculation.

Notes:

The **ES** segment for **ESI** and **EDI** may not be overridden with another segment prefix. Any such prefix will be ignored.

REP XSHA instructions are interruptible. As for other x86 **REP** instructions, the registers and memory are modified so that the instruction can continue execution, after the interrupt is serviced, in a manner transparent to the software. For **XSHA** instructions with **EAX** initially zero, the value in **EAX** will be modified to indicate the number of bytes processed before the interrupt.

The hash result is loaded and stored as a sequence of 32-bit integers in little-endian format. FIPS 180-2 specifies big-endian format. The calling program will need to BSWAP each 32-bit DWORD of the hash if it is to be communicated to any other program, process, or user.