VIA Nehemiah
Random Number Generator
Programming Guide

Version 1.0

This is **Version 1.0** of the VIA Nehemiah RNG 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.

Table of Contents

1 INTRODUCTION....2

2 PROGRAMMING INTERFACE....3

2.1ARCHITECTURE 3
2.2CONFIGURING AND ENABLING....5
2.2.1CPUID INSTRUCTION IDENTIFICATION 5
2.2.2MSR 0X110B 6
2.3X86 XSTORE INSTRUCTION 7
2.3.1XSTORE: STORE RANDOM DATA INSTRUCTION....8
2.3.2XSTORE: NUMBER OF BYTES STORED 9
2.3.3REP XSTORE INSTRUCTION 9
2.4STARTUP 10
2.5MANUFACTURING TESTS....11
2.6DC BIAS....11

4 PERFORMANCE 16

4.1BIT GENERATION SPEED 16
4.2INSTRUCTION TIMING....16
4.3POWER MANAGEMENT 17
4.4RANDOMNESS 17

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 die*.

A new VIA Nehemiah processor (stepping 8) extends this feature by placing two separate RNG noise sources on the die, improving both performance and randomness. This core also includes Padlock's Advanced Cryptography Engine or ACE. See the separate programming guide for more details on ACE.

This guide defines the RNG function as implemented in the VIA Nehemiah processor. Earlier versions of the VIA Nehemiah processor, starting with stepping 3, had a single RNG noise source. Software should not use the RNG feature on VIA Nehemiah processors prior to stepping 3.

The CPUID identification of the first VIA Nehemiah processor containing the RNG feature is:

Vendor ID: CentaurHauls
Family: 6
Model: 9
Stepping: 3

The CPUID identification for the newer VIA Nehemiah core with two RNG noise sources is:

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

Summarizing Nehemiah family core (CentaurHauls Family 6 Model 9) steppings:

Steppings 0 through 2	Steppings 3 through 7	Steppings 8 and higher
No RNG support	One RNG	Two RNG's

It is theoretically possible that even with a correct stepping, the RNG feature may not be present. Thus one must still check for existence of the RNG feature as described in section 2.2.

For support, interested developers should contact: rng_support@centtech.com.

CHAPTER

2

PROGRAMMING INTERFACE

2.1 ARCHITECTURE

Figure 2.1 is a conceptual view of the VIA Nehemiah processor RNG. The major architectural components are:

Two bits in the *Centaur extended feature flags* returned by the CPUID instruction. These bits identify

(1) whether the RNG feature physically exists on the processor, and

(2) whether it is currently enabled.

A new RNG status and control Machine Specific Register (MSR). In addition to providing status information, this privileged register controls global options of the RNG.

A high-speed hardware mechanism for generating random bits.

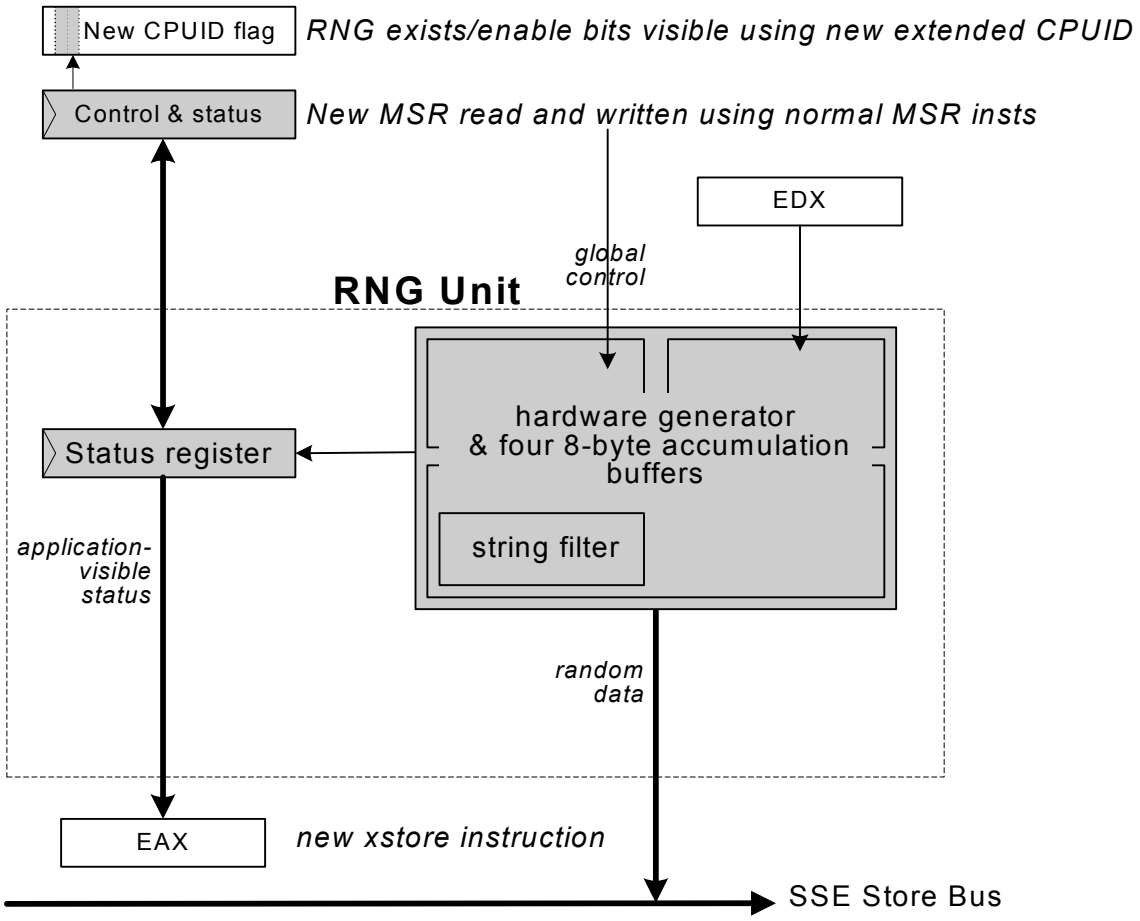
A hardware datapath mechanism that collects the generated bits into multiple buffers.

A non-privileged x86 instruction - `xstore` - which stores the collected random bits to memory, using the SSE store bus, as well as placing a copy of the status and control MSR in EAX. Applications can then verify that the RNG is configured acceptably.

An option to store only a specified subset of the generated bits, thus reducing the effective bit generation rate but improving the randomness of the data.

A REP version of the x86 instruction which allows easy collection of long contiguous strings of random data while allowing transparent interrupts.

Figure 1. Visible RNG Architecture



Subsequent sections in this chapter describe in detail these registers and instructions. The basic approach to use the VIA Nehemiah RNG is simple:

1. If special configuration is required (e.g., for testing purposes), set control values via the RNG MSR. This is a privileged operation and rarely needs to be done.

2. An application can use `xstore` at any time to store random data *and* the instantaneously accurate count of how many random bytes have been stored. This is an atomic instruction and can be used concurrently by any number of active tasks (and the operating system). Note, however, that any global options set in step 1 affect all users.

3. Examine the number of bytes returned in step 2, and repeat step 2, if necessary, until the desired number of bytes has been obtained.

   Or ...

4. Replace steps 2 and 3 with a REP xstore instruction specifying the exact number of bytes desired.

Paranoid (e.g., cryptographic) applications should be advised to verify that the RNG configuration is valid (or at least unchanged) after each xstore operation. The concern is that a different application running on the CPU could modify the RNG configuration. Centaur Technology recommends that paranoid applications not use the REP prefix.

2.2 CONFIGURING AND ENABLING

To use the VIA Nehemiah processor RNG, three conditions must be met:

1. The RNG feature 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. The RNG feature must be enabled and configured appropriately. The enabled state can be discovered from the Centaur Extended CPUID Feature Flags. The RNG feature can be enabled or disabled by writing to a new 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 CR4 enabling also enables the full set of SSE instructions on both Intel and on VIA Nehemiah processors. If CR4[9] is not set, the RNG behaves as if it were disabled via the RNG MSR, regardless of the setting of the RNG enable bit in the RNG MSR.

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

2.2.1 CPUID INSTRUCTION IDENTIFICATION

The VIA Nehemiah processors (Family=6, Model=9, Stepping=3 & higher) support the Centaur Extended Feature Flags. When the CPUID instruction is executed with EAX=0xC0000000, the processor returns 0xC0000001 in EAX. Note that earlier VIA 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 RNG:

EDX[2] == 0 RNG feature does not exist on this chip. A RDMSR/WRMSR of MSR 0x110B will cause a General Protection Fault, and the `xstore` instruction will always cause an Invalid Opcode Fault.

EDX[2] == 1 RNG feature exists, a RDMSR/WRMSR of MSR 0x110B is allowed. The behavior of `xstore` is dependent upon whether the RNG is enabled or not.

EDX[3] == 0 RNG is disabled. The `xstore` instruction will cause an Invalid Opcode Fault. This bit reflects the result of the setting of bit 6 of MSR 0x110B.

EDX[3] == 1 RNG is enabled (usually the RESET default). This bit reflects the result of the setting of bit 6 of MSR 0x110B. If 1, the `xstore` instruction can be used at any privilege level (provided that CR4[9] =1)

2.2.2 MSR 0X110B

A new MSR 0x110B provides both status and control of the RNG feature. In addition, a copy of the MSR contents as they are at the start of execution of an `xstore` instruction is placed in EAX as a result of executing an `xstore` instruction.

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

31:22	21:16	15:15	14:14	13:13	12:10	9:8	**7:7**	6:6	5:5	4:0
Reserved	String filter count	String filter failed	String filter enable	Raw bits enable	DC bias	Noise Source select	*Reserved*	RNG enable	*Reserved*	Current byte count

Reserved bits have undefined and unpredictable values. In the following detailed description, **(O)** means a read-only field output by the processor, **(I)** means the field can written by software. In addition, 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.

VIA recommends that when writing the MSR, all reserved bits should be set to zero. It is not possible to misconfigure the current version of the VIA Nehemiah processor RNG, but subsequent versions may define these reserved bits and software that improperly detects the processor stepping may misconfigure these later steppings.

Any write to MSR 0x110B causes a *load reset* of the RNG hardware. The load reset actions are:

- The internal hardware buffers are cleared to zero.
- The internal accumulation count in the string filter is reset to zero.
- The various options defined by the MSR are set to the value written to the MSR.

The detailed fields are:

Bits 4:0 Current Random Byte Count (O): 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 data using `xstore`, the number that appears may not be the same as the number stored on the next execution of `xstore`.

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.

Bit 6 RNG Enable (I): When set to 1, this bit enables the RNG. When set to 0, the `xstore` instruction becomes invalid, and the random number generator is internally disabled to reduce power consumption (see section 4.3). The extended CPUID feature flag that indicates whether or not the 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.

Bits 9:8 Noise Source Select (I): These bits control which of the two noise sources on the processor input bits to the accumulation buffers. Source "A" is the same as on earlier VIA Nehemiah processor steppings. Source "B" is a comparable noise source located elsewhere on the processor die.

00 Noise source "A" active

01 Noise source "B" active

10 Both noise sources active

11 Both noise sources active

On VIA Nehemiah processors prior to stepping 8, these bits are reserved and undefined. The default RESET state is for both bits to be zero; to operate like earlier VIA Nehemiah processors.

Bits 12:10 DC Bias (I): These bits control the DC bias supplied to the random number generator (see chapter 3). These bits may affect the speed of the generator and the randomness of the generated bits.

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 (see the hardware summary in chapter 3). A 0 in this bit selects the whitener function, which is required for producing truly random values. If it is desirable to sample the raw bits, then the string filter must also be disabled (bit 14) since the raw bits generated go through the string filter mechanism.

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 String Filter feature may be removed in future versions.***

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 String Filter feature may be removed in future versions.***

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 String Filter feature may be removed in future versions.***

2.3 x86 XSTORE INSTRUCTION

The VIA Nehemiah processor RNG feature implements a new x86 instruction: `xstore`. This instruction is enabled by the RNG enable bit in MSR 0x110B; an Invalid Instruction exception occurs if `xstore` is executed when not enabled.

2.3.1 XSTORE: STORE RANDOM DATA INSTRUCTION

This instruction stores zero, one, two, four, or eight random bytes in memory and places a copy of the current value of the RNG MSR 0x110B in EAX. This copy is referred to as the *status word*.

The opcode of `xstore` is an invalid opcode in normal x86 processors. When enabled, the format of the `xstore` instruction is:

0x0F	0xA7	0xC0

The `xstore` instruction requires that:

- ES:EDI points to the starting (low-order) memory address where the random data is to be stored. No segment override is possible.
- EDX specifies the rate at which the random data are returned, and the corresponding level of randomness. Only the lower two bits are meaningful, and the upper 30 bits will be set to zero by the instruction.

The `xstore` instruction has many similarities with the x86 `stos` instruction:

- The value in EDI is updated for the number of bytes stored.
- An address-size prefix affects the size of EDI used and updated.
- A REP prefix affects the behavior of the `xstore` instruction.
- The usual address exceptions (past the segment limit, etc.) can occur.

And many contrasts:

- A REPNE prefix causes an Invalid Instruction exception.
- In addition to storing the random bytes, the status word is placed in EAX.
- An operand size prefix causes an Invalid Instruction exception.
- 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.

Either zero, one, two, four or eight bytes are always stored to memory depending on the data rate specified by EDX and the status of the RNG hardware. Software must always provide an appropriate size writable area at the ES:EDI destination address.

Zero, one, or two 4-byte stores perform the store. 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 DS:EDI should be 4-byte aligned. Similarly, segment limit checks, protection check, page faults, etc. behave as if the data size is four bytes.

2.3.2 XSTORE: NUMBER OF BYTES STORED

The rules for the number of bytes stored by xstore are:

- The `xstore` instruction tests whether an eight-byte 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 (defined by EDX, more on this subsequently) 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.

Here is a summary:

EDX 1:0 value	Total bytes available	Total bytes stored	Random bytes stored	EAX[4:0] & EDI update byte count
N/A	< 8	0	0	0
00	>= 8	8	8	8
01	>= 8	4	4	4
10	>= 8	4	2	2
11	>= 8	4	1	1

2.3.3 REP XSTORE INSTRUCTION

The `REP` prefix on an `xstore` instruction performs as follows:

- The ECX value defines the total number of random bytes to be stored.
- The `xstore` function is repeatedly performed until the ECX number of random bytes are stored.
- Zero, one or two 4-byte stores are performed as for each iteration of `xstore` but ECX and EDI are updated automatically such that random bytes are concatenated into a contiguous string and iterations continue until the desired number of *random* bytes are stored.

The implications of storing fewer random bytes than the size of the store are that unaligned four-byte stores may be performed. These are slower than aligned stores and 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 stored. Adequate memory must be allocated based on the EDX value to contain the last one or two four-byte stores.

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

(1) ES:EDI is updated to address the next location for storing *random* byes

(2) ECX is updated to contain the number of random bytes remaining to be stored, and

(3) 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 run during an interrupt could, at least 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.

2.4 STARTUP

During RESET, *if the RNG unit is present:*

- MSR 0x110B is initialized with the value: 0x00000040. This enables the RNG unit (bit 6 = 1) and sets the default values of all control fields to 0.
- The RNG hardware accumulation buffers are cleared to 0 and the accumulation count 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 the RNG feature. 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], the RNG is indirectly disabled.

Start-up testing is a good idea in all environments. It is also a good idea to perform some test within an application before any collected random data is trusted. The FIPS 140-2 standard, in fact, requires start-up testing of RNG devices at the higher levels of certification.

Which startup test should be run is not obvious, however. There are two basic things that a startup test could be testing for:

- Whether the RNG feature is basically working or not (for example, does it have a stuck bit such that no data is ever returned).
- The randomness characteristics of the numbers generated.

In normal operation (the whitener enabled), it is theoretically possible that the `xstore` instruction generates no new bits for a long time. This theoretical delay has no guaranteed maximum, even in a correctly working part, in the same way that there is no limit to the number of sequential coin tosses that can come up heads. In practice, however, the likelihood of a long delay is negligible. This is based on (1) general statistical principles, (2) the inherent design of the random bit generator, and (3) our observations based on extensive testing. Our testing has never showed any deviation from the basic generation rate over long periods of operation (many months). The bit rate seems to be nearly constant from startup to shutdown. If startup tests run successfully (i.e., the hardware has not failed), applications should not need to detect timeouts.

Of course, a very conservative application can easily add a time-out check, since the atomic `xstore` instruction is usually used in a loop counting the bytes stored. It is easy to add another count that can detect **N** `xstore` instructions with no data returned. Alternatively, a time limit can be set and checked by the application.

The timeout detection works because the `xstore` instruction always immediately returns to execute the next instruction even if no random data is available. The `REP xstore` form, however, does not return control until the all of the random bytes requested have been stored. There is no built-in timeout check. If the RNG has died, the instruction will wait forever. Since `REP xstore` is interruptible, however, this case hangs the application, but not the operating system or other applications. It is also theoretically possible to detect a `REP xstore` timeout by intercepting an interrupt and examining the ECX count compared to some timer information about how long the `REP xstore` has been executing. This is very tricky and requires hooks into the operating system, so we recommend the following: do not use the `REP xstore` instruction if you are concerned about an RNG timeout condition.

2.5 MANUFACTURING TESTS

The RNG feature on every part is tested as part of manufacturing test. While it is impossible to perform a strenuous statistical test, a FIPS 140-2 monobits tests is executed (20,000 bits) at EDX=0 (the lowest quality setting). Passing this test verifies that the bit generator and the associated datapath are alive and working with no stuck bits, etc. It does not assure that the generator is statistically good enough to be considered truly random, but will generally detect catastrophic failures that result in highly biased or stuck output. (Actually, a failure on this FIPS test does not really mean that the generator is not random, but since the probability of a failure on one sample from a truly random generator is so low, the presumption is that a fail means broken.)

Important: *At this time,* if a processor fails the RNG test, or times out after a reasonable time without generating the expected number of bits, the random generator is permanently disabled for that particular part. This means that the RNG feature may be marked nonexistent on some VIA processors. VIA may change the RNG manufacturing tests at any time. Customers requiring RNG support may wish to contact VIA to ensure that all parts ordered have active RNGs.

2.6 DC BIAS

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.

The VIA Nehemiah processor RNG feature 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).

One might think that a boot time characterization could be used to determine the optimal DC bias setting for a particular part. But an *extensive* number of samples must be collected and analyzed statistically, and the

time required for a comprehensive determination of the correct DC bias is prohibitive for the boot process. Setting a non-standard DC bias may cause other applications to conclude that the RNG is misconfigured.

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

CHAPTER

3

HARDWARE DESCRIPTION

For ease in understanding, this discussion of the random bit generator describes the original VIA Nehemiah processor (stepping 3) with a singe set of oscillators. On VIA Nehemiah Steppings 8 and higher, there are two independent hardware generators, both of which feed into the datapath logic.

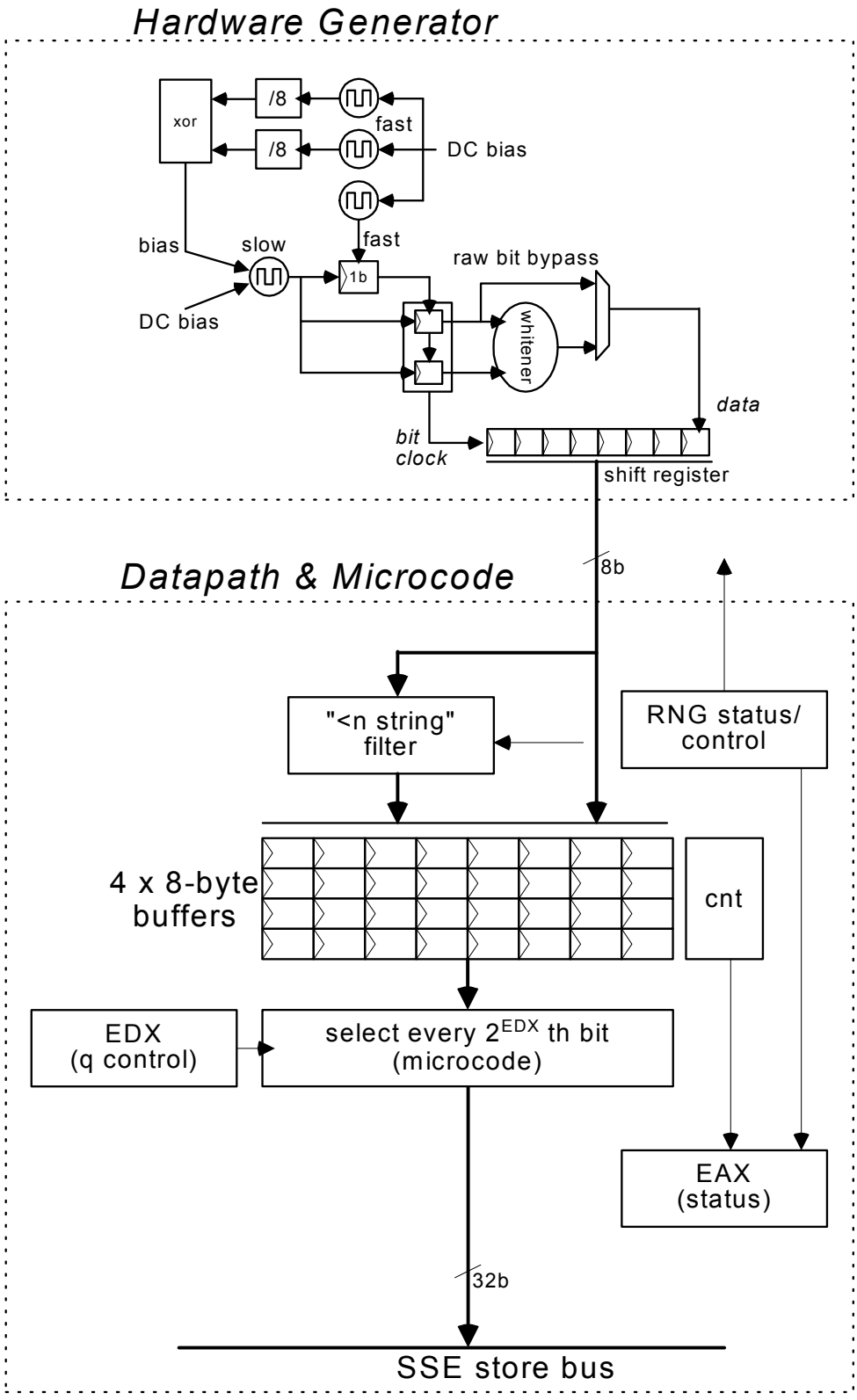
Figure 2 provides a pictorial overview of the RNG feature.

There are three fast oscillators, and one slow oscillator. These are free running ring oscillators; that is, their frequencies are not synchronized or stabilized and individually drift over time. The fast oscillator frequencies vary around a center of about 1 ns. The slow oscillator varies around a center of about 20 ns.

These oscillator frequencies are influenced by many factors:

- Random quantum effects of the transistor behavior: thermal noise in the channel, gate leakage caused by tunneling, etc.
- Dynamic voltage and temperature variations.
- Dynamic electromagnetic effects.
- Process differences in silicon dimensions. The oscillators are laid-out at right angles to each other to further accentuate silicon differences.
- Process differences in silicon characteristics such as implant characteristics.

Figure 2. RNG Hardware Overview



The operating voltage of all oscillators is partially controlled by a single DC bias which can be adjusted in eight steps via a programmable value.

The output of two of the fast oscillators is XOR'd to produce a variable signal. This variable signal further (in addition to the DC bias) modulates the voltage bias of the slow oscillator.

The output of the variably biased slow oscillator is used as the clock to sample a bit from the third free running fast oscillator.

Each two bit pair generated is optionally processed through a von Neumann whitener, which may output a single bit or nothing. If a bit is produced, it is then accumulated in an eight-bit shift register. This whitener function will reduce or eliminate any residual bias in the hardware generator. For example, the whitener should correct for biases in the third oscillator's duty cycle.

The input/output function of the whitener is

Input	Output
00	Nothing
01	0
10	1
11	Nothing

When eight new bits have been accumulated, a strobe causes the eight bits to be grabbed by the RNG data path (which is running using the synchronous processor clock).

On the Nehemiah Stepping 8, if both hardware noise generators have accumulated a byte during the same processor clock cycle, and both are selected, the datapath will accept only that byte from the "original" generator. The data from the second generator is lost. This is a relatively rare event.

Once eight bits have been generated they are delivered to the synchronous datapath. Each incoming byte is examined by the string filter function. If it is enabled, it may suppress the delivery of the byte to the accumulation buffers.

The incoming bytes are placed sequentially in the next consecutive position of four 8-byte buffers. The hardware maintains FIFO pointers and a count so that it can properly store the incoming bytes and access the requested bytes.

The `xstore` instruction itself is implemented in microcode. When executed, the microcode checks to see if eight new bytes are available in one of the buffers. If not, it returns with a zero byte count indicated in the

status word. When eight bytes are available, the microcode performs the one-of-N selection (specified by EDX), assembles the resulting data, and performs the store.

CHAPTER

4

PERFORMANCE

4.1 BIT GENERATION SPEED

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. For a VIA Nehemiah Stepping 8 processor, with *both noise generators selected*, our data show that depending on temperatures and DC bias settings, bits are delivered to the accumulation buffers at about 80 Mbs to 120 Mbs.

With the whitener selected, the rate will fall to between 12 Mbs and 20 Mbs.

Tests have also shown that there is typically a minor correlation between sequential bits delivered to the accumulation buffers. To reduce this correlation, the `xstore` instruction uses the value in the EDX register (mod 4) as a divider, and returns only every 2^{EDX} bit from the accumulation buffers.

4.2 INSTRUCTION TIMING

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` 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 information regarding 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 1 GHZ processor requires an average of about 11,000 clocks to fill a single 8-byte accumulation buffer.

Table 1. xstore Execution Clocks

EDX	Data Ready	Random Data 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 have 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 to the values in Table 1 .

For the more typical use of collecting a large number of bytes, the speed of `REP xstore` becomes the bit generation speed plus a few hundred clocks.

4.3 POWER MANAGEMENT

The RNG hardware is clock-enabled when the MSR enable bit is set. At this time, the random number bit generator is enabled. In addition, the RNG data path associated with accumulating the data, counting bytes, etc. is clock-enabled. Thus, dynamic power starts being consumed by the RNG unit 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 new `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.

4.4 RANDOMNESS

Evaluation of the statistical qualities of the bits produced by the VIA Nehemiah processors is a complex process. An independent evaluation has been performed by Cryptography Research, Inc., of San Francisco. A copy of their report is available at: www.cryptography.com/resources/whitepapers/VIA_rng.pdf

A comprehensive discussion of randomness, of Centaur's internal testing methodology and results, and recommendations for using the Nehemiah RNG feature in various environments is available in the *VIA Nehemiah Security Application Note*.