NPS-MA-09-001

NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

Implementing AES on the CellBE

by

David Canright, George Dinolt, Simson Garfinkel,
Jonathan Herzog, Bruce Allen

20 January 2009

Approved for public release; distribution is unlimited

Prepared for: National Security Agency

THIS PAGE INTENTIONALLY LEFT BLANK

NAVAL POSTGRADUATE SCHOOL
Monterey, California 93943-5000

Daniel T. Oliver
President

Leonard A. Ferrari
Executive Vice President and
Provost

This report was prepared for the National Security Agency and funded by the National Security Agency.

This report was prepared by:

_____
David Canright
Associate Professor of Mathematics

Reviewed by:

_____
Carlos Borges
Department of Applied Mathematics

Released by:

_____
Karl A. van Bibber
Vice President and
Dean of Research

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY ***(Leave blank)***	**2. REPORT DATE** 20 January 2009	**3. REPORT TYPE AND DATES COVERED** Technical Report 1 January – 31 December 2008	
4. TITLE AND SUBTITLE: Implementing AES on the CellBE			**5. FUNDING NUMBERS**
6. AUTHOR(S) David Canright, George Dinolt, Simson Garfinkel, Jonathan Herzog, Bruce Allen			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) National Security Agency 9800 Savage Road, Ste. 6538 Fort Meade, MD 20755-6538			**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**
11. SUPPLEMENTARY NOTES The views expressed in this report are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.			**12b. DISTRIBUTION CODE** **A**
13. ABSTRACT (maximum 200 words) When the CellBE processor was introduced, the Advanced Encryption Standard (AES) was one of the benchmarks; IBM published throughput speeds for different modes but gave no details on the precise implementation. Our team has developed AES independently. For ECB encryption our version is slightly faster than that of IBM; for CBC encryption our version is significantly faster. This paper describes our development process and design tradeoffs, with emphasis on lessons learned. This could be useful for anyone wishing to develop high-speed applications on the CellBE.			
14. SUBJECT TERMS AES, Advanced Encryption Standard, CellBE, Cell Broadband Engine, SPU, Synergistic Processor Unit, SIMD, Assembly Language			**15. NUMBER OF PAGES** 38
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UU

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

When the CellBE processor was introduced, the Advanced Encryption Standard (AES) was one of the benchmarks; IBM published throughput speeds for different modes but gave no details on the precise implementation. Our team has developed AES independently. For ECB encryption our version is slightly faster than that of IBM; for CBC encryption our version is significantly faster. This paper describes our development process and design tradeoffs, with emphasis on lessons learned. This could be useful for anyone wishing to develop high-speed applications on the CellBE.

THIS PAGE INTENTIONALLY LEFT BLANK

1 Introduction

Our team has implemented authenticated encryption, using Galois Counter Mode (GCM)[6, 11], on the Cell Broadband Engine (CellBE) processor[3]. An essential part of GCM is a block cipher, here the Advanced Encryption Standard (AES)[8]. This paper details the process through which we developed AES on the CellBE, and were able to match and even surpass the speed benchmarks set by IBM[1].

1.1 CellBE Processor

The Cell Broadband Engine (CellBE) processor architecture was designed jointly by Sony, Toshiba, and IBM, as a versatile multi-processor suitable for a wide variety of applications[3]. It is best known as the processor inside the PlaySation3, which has been very successful.

The currently available CellBE chip includes a main PowerPC Processor Element (PPE) along with eight "Synergistic Processor Elements" (SPEs). The intent is that the PowerPC processor should run the operating system and farm out all the computationally intensive tasks to the SPEs.

The SPEs have a different instruction set using Single Instruction Multiple Data (SIMD) parallelism, with 128 registers, each 128 bits wide[4]. Each SPE includes a Synergistic Processor Unit (SPU, the central processor), Local Store memory (LS, 256 KB), and a Memory Flow Controller (MFC) that handles DMA to/from the LS. The SPU has two instruction pipelines, called even and odd, each of which handles specific instruction types. That is, any particular instruction is either even type (e.g. `xor`) or odd type (e.g. load).

One application area used to demonstrate the capabilities of this new processor was cryptography. In particular, IBM published speeds for the Advanced Encryption Standard (AES), given in terms of throughput for a single SPE. Unfortunately, IBM did not publish its code.

1.2 Advanced Encryption Standard

The Advanced Encryption Standard (AES) was specified in 2001 by the National Institute of Standards and Technology[8]. The purpose is to provide a standard algorithm for encryption, strong enough to keep U.S. government documents secure for at least the next 20 years. The earlier Data Encryption Standard (DES) had been rendered insecure by advances in computing power, and was effectively replaced by triple-DES. Now AES will largely replace triple-DES for government use, and has become widely adopted internationally for a variety of encryption needs, such as secure transactions via the Internet.

The AES algorithm, previously called the Rijndael algorithm[2], is a symmetric encryption algorithm, meaning encryption and decryption are performed by essentially the same steps. It is a block cipher, where the data is encrypted/decrypted in blocks of 128 bits. (The original Rijndael algorithm allows other block sizes, but the Standard only permits 128-bit blocks.) Each data block is modified by several "rounds" of processing, where each round involves four steps. Three different key sizes are allowed: 128 bits, 192 bits, or 256 bits, and the corresponding number of rounds for each is 10 rounds, 12 rounds, and 14 rounds. From the original key, a different "round key" is computed for each of these rounds.

There are several different modes in which AES can be used [7]. For some of these, such as Cipher Block Chaining (CBC), the result of encrypting one block is used in encrypting the next. These are called feedback modes, and the feedback effectively precludes processing several blocks in parallel. Other modes, such as the "Electronic Code Book" mode and "Counter" modes, do not require feedback. These non-feedback modes may be parallelized for greater throughput.

Here we give a brief description of the algorithm, to indicate the computations involved. The four steps in each round of encryption, in order, are called[8] *SubBytes* (byte substitution), *ShiftRows*, *MixColumns*, and *AddRoundKey*. Before the first round, the input block is processed by AddRoundKey (one could consider this round number zero). Also, the last round skips the MixColumns step. Otherwise, all rounds are the same, except each uses a different round key, and the output of one round becomes the input for the next. (For decryption, the mathematical inverse of each step is used, in reverse order; certain manipulations allow this to appear like the same steps as encryption with certain constants changed.)

The single *nonlinear* step is the *SubBytes* (byte substitution) step, where each byte (8 bits) of the input is replaced by the result of applying the "S-box" function to that byte. This nonlinear function involves finding the inverse of the 8-bit number, considered as an element of the Galois field $GF(2^8)$. This is not a

simple calculation, and so AES implementations typically use a precomputed S-box table, where the input byte is an index into the table to find the output. This table look-up method is fast, easy to implement, and only requires 256 bytes.

The other three steps, (*ShiftRows*, *MixColumns*, and *AddRoundKey*) are *linear*, in the sense that the output 128-bit block for such steps is just the linear combination (bitwise, modulo 2) of the outputs for each separate input bit.

The *ShiftRows* step considers the current 128-bit state as a 4×4 matrix of bytes (ordered as 4 columns). This step rotates each row of bytes left by the row index (0–3); it just moves bytes around.

The *MixColumns* step considers the state as 4 columns of 4 bytes each, and multiplies each column by a constant matrix:

$$\begin{pmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{pmatrix} \begin{pmatrix} C_0 \\ C_1 \\ C_2 \\ C_3 \end{pmatrix} \rightarrow \begin{pmatrix} D_0 \\ D_1 \\ D_2 \\ D_3 \end{pmatrix}$$

where byte multiplication and addition uses the Galois arithmetic of $GF(2^8)$. In this field, each byte can be considered the coefficient vector of a polynomial of (formal) degree 7: $\mathbf{a} = a_7x^7 + \cdots + a_1x + a_0$ where each coefficient a_i is a bit. Addition (mod 2) is then bitwise XOR. Multiplication is polynomial multiplication, modulo the irreducible polynomial $x^8 + x^4 + x^3 + x + 1$. Then in the matrix above, '2' (00000010) means the polynomial x, and $2 \times \mathbf{a} = a_7x^8 + \cdots + a_1x^2 + a_0x$, but modulo $x^8 + x^4 + x^3 + x + 1$, giving `( a << 1 ) ^ ( a7 * 0x11B )` in C notation. And $3 \times \mathbf{a} = \mathbf{a} + (2 \times \mathbf{a})$. So *MixColumns* really only requires Galois multiplication by 2.

The inverse *MixColumns* operation uses the inverse of the above matrix (shown below in hexadecimal):

$$\begin{pmatrix} E & B & D & 9 \\ 9 & E & B & D \\ D & 9 & E & B \\ B & D & 9 & E \end{pmatrix} \begin{pmatrix} D_0 \\ D_1 \\ D_2 \\ D_3 \end{pmatrix} \rightarrow \begin{pmatrix} C_0 \\ C_1 \\ C_2 \\ C_3 \end{pmatrix}$$

This is a bit more complicated, since it requires multiplication by 2, by 4, and by 8 (or repeated multiplication by 2)

These Galois multiplications may be replaced by table look-ups, and these table lookups can be combined with those for the *SubBytes* (as suggested by the developers of Rijndael[2]). That is, *ShiftRows* can be done first in each round (just a matter of indexing correctly), then for each byte in a column, *SubBytes* and *MixColumns* requires one table lookup of a 4-byte column, and those 4 columns are added (XOR) to give the output column. This approach requires 4 tables (a different table for each byte row position), each of 256 columns, for a total 4 KB of storage. All the fastest general software implementations of AES use this approach, which has been called the T-table approach.

Lastly, the *AddRoundKey* step is merely adding (bitwise XOR) the Round Key to the current state.

1.3 Analysis of IBM's Results

As one of the benchmarks for the CellBE processor, IBM published timing results for their implementations of AES[1]. These results are given for a single SPU processor in terms of throughput rates measured in Giga-bits per second. They give results for each of the three key sizes, both ECB and CBC modes, both encryption and decryption. We asked IBM for the code and was told that it would not be released.

We analyzed their numbers, based on a simple model for their unknown code. We assumed their code was structurally similar to ours, having an inner loop for each round, inside an outer loop for each block, where the block loop may be partially unrolled to process some small number of blocks in parallel (for non-feedback modes). Table 1 shows their rates and our loop models for them.

For each of the four modes (ECB/CBC, encrypt/decrypt) all we have to work with are three numbers. But based on this model, the reciprocals (time per bit) should fall on a straight line. We chose the axis units to be time in instruction clock cycles versus rounds per block. The slope of that line indicates the number of clock cycles needed for each round of each block, inside the round loop. The total number of clock cycles for one iteration of the round loop, processing some number b of blocks in parallel, must be an integer. So

Table 1: IBM's published throughput rates (in Gigabits/sec for one SPU, from [1]) are shown, along with our models of the loop structure of their code: we assume a small number of blocks is processed in parallel ('blks') inside the round loop, and give the clocks per round per block, as well as the extra clocks per block for the last round (usually negative). The last column shows the maximum relative error in our modeled rates.

IBM's published results (Gbits/sec)				loop model			
AES type	keysize			blks	clocks		max
	128	192	256		round	last	err
ECB encr.	2.059	1.710	1.462	4	20.25	−4	0.03%
CBC encr.	0.795	0.664	0.570	1	51	3	0.18%
ECB decr.	1.499	1.252	1.068	2	27.5	−3	0.21%
CBC decr.	1.507	1.249	1.066	4	28	−8.75	0.05%

the fractional part of the clocks/round/block should be a multiple of $1/b$. The intercept of the line indicates the extra clocks/block needed outside the round loop, that is, for the last round (and round 0); this number also should be a multiple of $1/b$. But if our model is wrong (say, if they fully unrolled the round loop) then the points are unlikely to lie on such a line.

The published rates give three (or a bit more) significant digits. The slopes for our least-squares fit lines should have similar precision, but the intercepts have less precision (from cancellation). The fractional part of the slope only has about one significant digit, but we used that to guess the number b of blocks processed in parallel. (For CBC encryption, the feedback requires that $b = 1$. For ECB decryption, the fraction was 0.5, consistent with either $b = 2$ or $b = 4$.)

Our loop models agree well with the published data. For ECB encryption and CBC decryption, our models reproduce the published throughput rates almost exactly. For ECB decryption, the three points do not fit a line so well (the rate for 192-bit keys seems relatively high); for CBC encryption, the points make a nice line but the slope is not exactly an integer. But even in those cases our models only give a small difference in the least significant digits of the rates, with a relative error of a fraction of a percent. The accuracy of these models gives strong support to our assumptions about the structure of their codes.

2 Code Development

Our goal was to implement AES on an SPE and optimize for speed. In particular, we needed the Counter Mode (CTR) of encryption, for incorporation into the authenticated Galois Counter Mode (GCM)[6]. In Counter Mode, a 128-bit counter is given an Initial Value (unique IV for each message for a given key). Then for each plaintext block, the counter is incremented and encrypted using AES with the secret Key; the result is added to the plaintext (as a stream cipher) to give the ciphertext block. Hence decryption in CTR mode is exactly the same process, and actual AES decryption is never required. (Later, for comparison with IBM's results, we also implemented Electronic Code Book [ECB] encryption and Cipher Block Chaining [CBC] encryption, a feedback mode.)

The registers in the SPU are 128 bits wide, perfect to hold the current state in the AES encryption. The SIMD instruction set includes operations on whole registers as a single "quad-word", or in parallel as 4 words (each 32 bits, one column of the AES state) or as 16 bytes (or even as 128 bits in parallel for such operations as XOR). So we started by implementing the basic round steps with SIMD parallelism.

The first design consideration was whether or not to use T-tables. The IBM Cell Broadband Engine Programming Handbook[4, 24.6.2] shows how to do 16 table lookups in parallel using the shuffle bytes command (`shufb`), and specifically uses the AES *SubBytes* step as an example. Briefly, `shufb` does lookups of bytes from tables in registers, based on the lowest 5 bits of the index byte; then each higher bit is used to successively select (`selb`) the correct result. However, the T-table approach requires using bytes to look up whole *words* (4-byte columns) rather than bytes. Doing this in parallel using `shufb` is infeasible (not enough registers) and anyway would be much less efficient than doing the lookups sequentially from tables in Local

Store memory. We tried both approaches, parallel SIMD or serial T-tables, and discuss the comparisons below. Table 2 summarizes the different versions of AES we developed, and shows the code refinement process.

2.1 SIMD Code

For the SIMD approach, an entire block is processed in parallel parts simultaneously, including: 128 parallel bit operations for *AddRoundKey*, 16 parallel byte operations for *SubBytes*, 4 parallel word operations for *MixColumns*, and a single quadword operation for *ShiftRows*. This parallelism requires replacing any instruction branching (based on data values) with selection operations. For example, in Galois multiplication by 2 (for *MixColumns*), after a left shift we add the modulo constant only if the leading bit was 1; for SIMD we compute both with and without the modulo constant, then bytewise choose (by `selb`) the correct result using a selector mask based on the leading bit of each byte.

(Note: The SPU Instruction Set[5] is limited since instructions are 32 bits wide and 7 bits are required to specify each register involved [up to four], so relatively few operation codes are available. Consequently, some instructions one might expect are not available. In particular, there are no instructions to rotate or shift bytes [only halfwords, words, and quadwords], which would be handy for the Galois multiplication by 2.)

Our initial SIMD code was a straightforward implementation of the steps of a round, in a loop for the rounds, inside a loop for each block (encrypted by Counter Mode). The *SubBytes* step was the most expensive computationally, *MixColumns* roughly half as expensive, and the other steps just one or two instructions. We call this version CTR0, and its speed is about one-quarter that of the IBM benchmarks. (The closest comparison for our CTR mode is IBM's ECB mode.)

The next version applied "instruction scheduling," where we move instructions around (within the limitations imposed by the algorithm). One goal here is to reduce or eliminate dependency stall, where an instruction waits for the result of a previous one. The other goal of instruction scheduling is to begin two instructions at once, one in each pipeline of the SPU; this is called dual-issue. This requires the two instructions to be of the correct types, in the correct order, aligned with the correct address parity (even, odd), with both instructions ready to commence: no waiting for earlier results. (Address alignment may be adjusted by inserting no-operation commands: `nop` or `lnop`; this may also be done with the assembler `.align` directive.) The ideal would be for all instructions to be dual-issued without any dependency stall, keeping both pipelines running nonstop. But the algorithm determines which instructions are required, so typically there are not equal numbers of instructions for each pipeline. Some operations may be achieved by different choices of instructions, so somtimes instructions for one pipeline can effectively be replaced by instructions for the other, to give a better balance for more dual-issues. Indeed, sometimes using more instructions to get a result may take less time through more dual-issues.

Another related improvement comes from providing branch hints in the code. (The SPU hardware does not automatically predict branches.) Without a branch hint, the SPU "assumes" that a branch instruction will *not* branch (even an unconditional branch instruction!); if the branch is actually taken, then the instruction queue must be flushed and refilled, with a penalty of 18 or 19 clock cycles, before execution resumes. A branch hint instruction predicts whether a later branch instruction will branch or not. (Only a single branch hint may be in effect at any time.) If the hint is correct and given early enough, then the hinted branch takes a single clock cycle and execution continues; if the hint was incorrect the usual branch penalty applies. So efficiency can be enhanced by eliminating branches where feasible (e.g., using selection operations `selb`) or correctly hinting branches.

Instruction scheduling our code greatly increased the amount of dual-issues and reduced dependency stalls. And we successfully hinted the branches for both the inner round loop and the outer block loop (except the last iteration of each loop does not branch, so suffers the penalty). These techniques nearly doubled the speed; we call the resulting code version CTR1.

Next we considered loop unrolling. If two or more iterations of a loop can be done together, then interleaving their instructions effectively reduces the data dependency stalls; the interleaved instructions can take advantage of what would otherwise just be waiting time. (But note that such interleaving may have little effect on dual-issue rates, as the balance of instructions between pipelines remains unchanged.) Furthermore, fully unrolling a loop, where feasible, can eliminate branch instructions and counter increments.

For AES, each round begins with the result of the previous round, so successive iterations of the round loop cannot be interleaved this way. However, for non-feedback encryption modes, such as CTR or ECB, the encryption of each block is independent of the other blocks. So the block loop may be partially unrolled to interleave instructions for two or more blocks. This makes the code more complicated and also requires using more registers (several for each block). At first we unrolled to do two blocks at once, which eliminated much of the dependency stall; this code is called CTR2. We later unrolled two more blocks, to process four blocks at a time, eliminated all the remaining dependency stall; this we call CTR4a. But this was still not as fast as IBM's benchmark ECB, though it was getting close.

The next improvement came from rethinking the *MixColumns* step. (Two versions were developed, one for feedback modes and one for the four-block unrolled loop, because they had different optimizations available.) One `xor` was saved by reorganizing the algebraic steps, particularly by adding rows 0 and 1 together before doing the Galois multiply by 2. And the scheduling was improved by combining *AddRoundKey* with the additions in *MixColumns*. Also, the dual-issue rate was improved by replacing some even pipeline commands by different odd pipeline ones. More specifically, some `roti` (rotate) instructions were replaced by `shufb` instructions, a `selb` (select) became two `shufb` instructions, and for one of the four blocks, a comparison instruction was replaced by *four* odd pipeline instructions.

Further instruction scheduling was applied in the four-block version, to take advantage of more dual-issues. This included preparing for the next iteration of blocks while finishing the last round of the current blocks, and interleaving some instructions from *MixColumns* for some blocks with the *SubBytes* for other blocks.

Finally, another improvement was dynamic branch hinting. By using a table of branch hint addresses, we could correctly hint even the last iteration of the round loop. This alone gave a further 3% speedup (in the one-block version).

At this point, we have a highly optimized version of AES in Counter mode, which encrypts four blocks at a time, called CTR4. Within the block and round loops (and mostly elsewhere): *every* odd-pipeline instruction is dual-issued (there are more even-pipeline instructions); there are *no* dependency stalls; *all* branches are correctly hinted (except the final iteration of the block loop).

The only further improvement we could see would be to fully unroll the round loop. This would *not* help the instruction scheduling any, since already there is no dependency stall and no more possibilities for dual issue. Also the branch itself is dual issued and properly hinted so takes no time. The one apparent improvement comes from eliminating the single (even-pipeline) instruction that increments the round counter itself. (The instructions that load and issue the branch hints for the round loop could also be eliminated, but since these are odd-pipeline instructions dual issued with essential even-pipeline commands, eliminating them would save no time.) Since we process four blocks at a time, this only helps by $\frac{1}{4}$ cycle/block/round. The downsides would be requiring three different versions of the encryption code, one for each key length, and each of these unrolled codes would be much longer (by roughly 4 to 6 times). So we have chosen not to unroll the round loop.

2.2 Other Encryption Modes

Besides Counter mode, we also developed code versions for other modes of encryption, primarily for direct comparison with IBM's results.

Electronic Codebook (ECB) mode is very similar to Counter mode, except the AES rounds are applied to the plaintext block, rather than to a counter. This saves two operations per block, relative to Counter mode: no counter block is incremented nor added to the plaintext. So our ECB code is slightly faster than our corresponding CTR code. And since each block is encrypted independently, we can partially unroll the block loop as in CTR mode. Hence our ECB encryption code is very similar to our CTR code.

We did not develop code for ECB decryption, nor any other mode requiring the AES decryption function, also called the inverse cipher. The inverse cipher is more complicated due to the larger factors in the inverse *MixColumns* matrix. (IBM's results show a decrease in throughput for ECB decryption.)

Cipher Block Chaining (CBC) mode begins encryption of a plaintext block by adding the ciphertext from the previous block (except the first block uses an Initial Value instead of the ciphertext block). This feedback increases security, but prevents any unrolling of the block loop. Since only a single block is processed at a time, opportunities for instruction scheduling are greatly limited, compared to the non-feedback modes. So

Table 2: Here we compare several different versions we have developed.

throughput results (Gbit/sec)				loop model		
code	keysize			blks	clocks	
	128	192	256		round	last
our SIMD CTR results:						
CTR0	0.496	0.411	0.351	1	85	−26
CTR1	0.867	0.731	0.631	1	44	31
CTR2	1.431	1.196	1.028	2	28	5.5
CTR4a	1.872	1.555	1.330	4	22.25	−4.25
CTR4	2.071	1.722	1.474	4	20	−2.75
our T-table CTR results:						
Tab1	0.827	0.692	0.596	1	48	14
Tab2	1.084	0.914	0.790	1	35	27
our CBC results:						
CBC1	0.898	0.752	0.647	1	44	15
CBC2	1.191	0.989	0.846	1	35	−7
our ECB results:						
ECB1	1.058	0.884	0.759	1	38	6
ECB4a	1.976	1.639	1.400	4	21.25	−5.75
ECB4	2.092	1.737	1.484	4	20	−4.75

the time per block is increased due to unavoidable data dependence waits and fewer dual issues; our resulting CBC code is roughly half as fast as the CTR version. (CBC decryption can process blocks in parallel, using the inverse AES cipher; we did not develop code for this.)

Besides ECB, CBC, and CTR modes, NIST has approved two other modes for security[7]. Cipher Feedback (CFB) mode and Output Feedback (OFB) mode both need the output of encrypting the previous block before they can begin encrypting the next block, so cannot encrypt blocks in parallel. Both also add (xor) the result of an AES encryption to the plaintext block to get the ciphertext. Hence, for decryption, both use only the forward AES algorithm, not the inverse cipher. CFB can decrypt blocks in parallel, but not OFB. (We did not develop codes for these modes, though they would be relatively simple modifications to versions we did develop.)

NIST has also approved three authentication modes based on block encryption: Cipher-based Message Authentication Code (CMAC)[9] essentially uses CBC encryption to generate an authentication hash; Counter with Cipher Block Chaining-Message Authentication Code (CCM)[10] combines CTR mode for encryption with CBC mode for authentication; Galois/Counter Mode (GCM)[11] uses CTR mode for encryption with a separate hash function not based on encryption. (Our main goal was to produce fast GCM encryption/decryption, which is why our main interest in AES is the CTR mode.) None of these authentication modes uses the inverse cipher.

2.3 T-table Code

Since fast software implementations of AES typically use the "T-table" approach (where table look-ups handle the combined *SubBytes* and *MixColumns* steps), we wanted to try this on the CellBE. So we developed a T-table code to investigate how the algorithmic parallelism of the T-table method compares with the SIMD parallelism available on the SPU.

In the usual software implementation, for each column (4 bytes = 1 word) of output, each of the four bytes of input indexes a different table of 256 words, and those four words are added (`xor`) together. This

requires 4 tables × 256 entries × 4 Bytes = 4KB of storage for tables. On the SPU, speed dictates that each lookup returns a quadword (16 bytes = 1 register = 1 block), since otherwise several more instructions would be required to get the desired word into the desired position in a register. So we set up 16 tables (four for each column of output, with zeros in the other column positions), and each of the 16 input bytes indexes one of those tables, with the 16 output quadwords getting summed for the result. Altogether this requires 16 tables × 256 entries × 16 Bytes = 64KB, or $\frac{1}{4}$ of the total Local Store memory of an SPE.

The lookups are done for each byte in serial fashion, which might normally suggest a loop over the 16 bytes. But we fully unrolled this (potential) byte loop, which allows us to replace the *ShiftRows* step by choosing the shifted index in each case. For each of the 16 table lookups in a round, the corresponding byte first must be moved to the correct position in the "preferred slot" of a register, with all higher bits of that word zeroed out. Different approaches to do this were combined to balance the two pipelines.

By the way, the exact same approach is used in the Galois Hash operation of GCM. There, the operation performs multiplication of a 128-bit data block with a known 128-bit constant H in the Galois field $GF(2^{128})$. Sixteen tables, each one block wide by 256 long, are precomputed from H to give the contribution to the product from each byte of the data block. Then this Galois multiplication consists of using each input byte to index a different table and adding up (xor) all 16 of the 128-bit contributions (by the distributive property of multiplication).

Our T-table implementation (of CTR mode) has no unrolling of the block loop (nor the round loop). The round loop requires 35 clocks per round; the last round takes longer. (Since the last round lacks the *MixColumns* step, the T-table method requires additional instructions to mask the table outputs.) Although we did not develop a multi-block version using T-tables, we can estimate how much improvement is possible: it appears the best we might achieve by partially unrolling the block loop would be over 27 clocks per round.

One other improvement for the T-table approach would be the rather obscure trick called "counter-mode caching." For 15 out of 16 blocks, only the least significant byte of the CTR changes from the previous value. Then for the *first* round, only that byte needs a table look-up; the rest can be cached from the last block's first round. (This trick doesn't help the SIMD approach, since all bytes are processed in parallel.) We have not implemented this, but estimate that counter-mode caching would improve the throughput rates by no more than 6% for the one-block version. (This caching trick would not be feasible for multi-block versions. But for GCM, only the four least significant bytes of the counter ever change, so the results of the first round for the remaining 12 bytes could be cached.)

So how does the T-table method compare to the SIMD approach? In terms of memory, T-tables require an extra 64 KB. The speed comparison depends on the mode. For non-feedback modes of encryption, such as CTR mode, our 4-block SIMD version is much faster than the T-table approach (about 45% faster than our estimate for a multi-block table version). Hence "counter-mode caching" is moot. For feedback modes of encryption, such as CBC mode, our 1-block SIMD version is slightly faster (about 8%) than the T-table approach. (Both approaches take 35 clocks/round in the round loop; the difference is in the last round. Conceivably one could graft T-table rounds to a SIMD last round to get a version just as fast as our pure-SIMD CBC code.)

But for AES *decryption*, the SIMD approach gets more complicated due to the larger factors in the inverse *MixColumns*, while the T-table approach remains essentially unchanged, except for using a different 64 KB set of tables. We did not implement decryption, but judging from IBM's results, SIMD decryption for non-feedback modes should take about 27.5 clocks/round, comparable to the T-table approach. But for decryption modes requiring feedback, we expect the T-table approach to be significantly faster than SIMD. However, *none* of the five security or three authentication modes approved by NIST use the inverse AES cipher with feedback in decrypting, so this potential advantage of T-tables might only apply to some non-standard mode. Therefore, T-tables offer no significant speed advantages for any standard modes on the SPU, yet carry the significant cost of using $\frac{1}{4}$ of the Local Store memory (or $\frac{1}{2}$ if both AES encryption and decryption are needed).

3 Results and Conclusions

We have successfully developed fast versions of AES for the Synergistic Processor Elements of the CellBE processor. Our main interest was CTR mode, as part of Galois Counter Mode authenticated encryption,

Table 3: We compare our measured throughput rates (for one SPU) with those published by IBM. Also shown (using our models for IBM's results): number of blocks processed in parallel, clocks per round per block, and extra clocks per block for the last round.

throughput results (Gbit/sec)				loop model		
who	keysize			blks	clocks	
	128	192	256		round	last
ECB encr (no feedback):						
ours	2.092	1.737	1.484	4	20	−4.75
IBM's	2.059	1.710	1.462	4	20.25	−4
CBC encr (feedback mode):						
ours	1.191	0.989	0.846	1	35	−7
IBM's	0.795	0.664	0.570	1	51	3

but we also developed versions for ECB and CBC encryption modes. Table 3 compares our results with the IBM benchmarks, for the two modes implemented by both teams. We measured the throughput rates for our code using the system clock to find the time taken for our subroutine to encrypt a buffer full of blocks.

Our implementation of ECB encryption is slightly faster than IBM's (1.6% for 128-bit keys). Compared to our loop model of their code, we were able to save one more instruction per four blocks in the round loop (by replacing an even pipeline instruction by four odd pipeline instructions as mentioned above). More importantly, we are willing to make our code public, which IBM is not.

And for CBC encryption, our implementation is 50% faster (for 128-bit keys), a significant improvement over the IBM benchmark. (We remain curious why there is such a difference for CBC mode.)

In developing our AES code, we compared the T-table approach (found in all the fastest standard C implementations of AES), which uses serial table lookups, with the SIMD approach of processing a whole block in parallel. For non-feedback encryption modes SIMD is much faster (approximately 45%). For feedback modes of encryption and non-feedback decryption modes, T-tables are basically the same speed[1] as SIMD but use up at least $\frac{1}{4}$ of the Local Store memory. There are no standard modes where AES decryption must be done using feedback, but if there were, T-tables would likely be faster than SIMD for those. So for all standard modes, there is no reason to use T-tables on an SPU.

The method we used to develop fast code follows the suggestions in the IBM documentation for programming the SPU[4]. While the IBM programming environment provides great support for writing in a high level language such as C, including ways to include particular assembly language instructions, we chose to develop the most time-intensive portions of GCM (including AES) directly in assembly language. The first step was to arrange the algorithm to take full advantage of the SIMD architecture of the SPU, including replacing data-dependent branching by selection operations. Then instructions were scheduled (moved around), with the help of partial loop unrolling where feasible, to reduce the number of cycles where one or both pipelines was idly waiting for a previous result. This included moving instructions from one pipeline to equivalent instructions on the other in order to balance the load, to get both pipelines done sooner. And correctly hinting the remaining branches as often as possible eliminated instruction cache waits.

Our independent development of AES on the CellBE makes fast encryption code publicly available, and adds more confirmation of the powerful capabilities of the CellBE architecture.

[1] based partly on IBM's results, assuming their decryption was SIMD

References

[1] T. Chen, R. Raghavan, J. N. Dale, and E. Iwata. Cell Broadband Engine architecture and its first implementation—a performance view. *IBM Journal of Reseach and Development*, 51(5):559–572, September 2007. http://researchweb.watson.ibm.com/journal/rd/515/chen.pdf.

[2] Joan Daemen and Vincent Rijmen. *The Design of Rijndael, AES - The Advanced Encryption Standard.* Springer-Verlag, 2002.

[3] IBM. Introduction to the cell Broadband Engine. http://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/D21E662845B95D4F872570AB0055404D, 31 October 2005.

[4] IBM. Cell Broadband Engine programming handbook, version 1.2. http://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/9F820A5FFA3ECE8C8725716A0062585F, 24 April 2007.

[5] IBM. Spu instruction set architecture, version 1.2. http://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/76CA6C7304210F3987257060006F2C44, 27 January 2007.

[6] David A. McGrew and John Viega. The Galois/counter mode of operation (GCM). http://csrc.nist.gov/groups/ST/toolkit/BCM/documents/proposedmodes/gcm/gcm-revised-spec.pdf, May 2005.

[7] NIST. Recommendation for block cipher modes of operation, December 2001. SP 800-38A.

[8] NIST. Specification for the ADVANCED ENCRYPTION STANDARD (AES), November 2001. FIPS PUB 197.

[9] NIST. Recommendation for block cipher modes of operation: The CCM mode for authentication and confidentiality, May 2004. SP 800-38C.

[10] NIST. Recommendation for block cipher modes of operation: The CMAC mode for authentication, May 2005. SP 800-38B.

[11] NIST. Recommendation for block cipher modes of operation: Galois/counter mode (GCM) and GMAC, November 2007. SP 800-38D.

A Optimization of *MixColumns*

Here we detail the steps by which we optimized the *MixColumns* step, including the relevant assembly language source code (taken out of context). This section shows most of the interesting optimizations of the round loop, since our implementation of *SubBytes* basically follows the SIMD table lookup given in the IBM Programming Handbook[4].

Considering the 128-bit state block (register) as a 4×4 matrix of bytes, then *MixColumns* performs the same operation on each of the 4 columns (words in the register). For an input column (r_0, r_1, r_2, r_3), the top output byte (#0) is given by $2 \times r_0 + 3 \times r_1 + r_2 + r_3$, and the other output bytes are the rotated equivalent (so output #1 $= 2 \times r_1 + 3 \times r_2 + \ldots$, etc.) The multiplication is in the Galois field of bytes, so to multiply by 2 one shifts left 1 bit then reduces modulo the field polynomial, represented by the nine-bit constant 0x11B. (If the most significant bit was initially 0, the result is the usual multiply by 2.) And as usual, $3 \times x = 2 \times x + x$, except each addition is bitwise xor.

The initial assembly version of this (in CTR0) was a direct SIMD implementation: clear msb of bytes then shift quadword left by 1 bit (this could be done in one step if there were a "shift byte" instruction); maybe add 0x1B, using byte selector based on msb (bit7), to get $2 \times x$; add original byte to get $3 \times x$; rotate columns and add rows. (Note: to aid readibility, our assembly source uses named registers, beginning `$R`; pipeline 0 instructions are flush left while pipeline 1 instructions are indented; dual-issued instruction pairs are indicated by braces.)

```
# SIMD version #0 of Mix Columns
  andbi         $Rtimes2, $Rstate, 0x7F                # ain't no "shift byte"; clear msb
        shlqbii $Rtimes2, $Rtimes2, 1                  # shift block 1 bit
  xorbi         $Rtimes2m, $Rtimes2, 0x1B              # mod field polynomial
  clgtbi        $Rbit7, $Rstate, 0x7F                  # if msb = 1
  selb          $Rtimes2, $Rtimes2, $Rtimes2m, $Rbit7  # now have byte x 2 in GF
  xor           $Rtimes3, $Rtimes2, $Rstate            # also byte x 3
  roti          $Rrow1, $Rtimes3, 8                    # rotate columns and add:
  xor           $Rcols, $Rtimes2, $Rrow1               # 2 x r0 + 3 x r1
  roti          $Rrow2, $Rstate, 16
  xor           $Rcols, $Rcols, $Rrow2                 #  + 1 x r2
  roti          $Rrow3, $Rstate, 24
  xor           $Rstate, $Rcols, $Rrow3                #  + 1 x r3, and done
```

The next version (in CTR1) was essentially the same steps, but in a different order (instruction scheduling), to get some dual issues and reduce data dependency stall:

```
# SIMD version  #1 of Mix Columns
  andbi         $Rtimes2, $Rstate, 0x7F                # no "shift byte"; clear msb
  clgtbi        $Rbit7, $Rstate, 0x7F                  # if msb = 1
# dual issue:
{ roti          $Rrow2, $Rstate, 16
        shlqbii $Rtimes2, $Rtimes2, 1                  # shift block 1 bit
# dual issue:
{ roti          $Rrow3, $Rstate, 24
        lqx     $Rroundkey, $Rroundkeys, $Rround       # get round key
  xorbi         $Rtimes2m, $Rtimes2, 0x1B              # mod field polynomial
  selb          $Rtimes2, $Rtimes2, $Rtimes2m, $Rbit7  # now have byte x 2 in GF
  xor           $Rtimes3, $Rtimes2, $Rstate            # also byte x 3
  roti          $Rrow1, $Rtimes3, 8                    # rotate columns and add:
  xor           $Rcols, $Rtimes2, $Rrow2               # 2 x r0 + 1 x r2
  xor           $Rcols, $Rcols, $Rrow3                 #  + 1 x r3
  xor           $Rstate, $Rcols, $Rrow1                #  + 3 x r1, and done
```

Partially unrolling the block loop allowed reduction (CTR2) or elimination (CTR4a) of the remaining data dependency stall, by interleaving instructions for 2 or 4 blocks to fill in the "wait" cycles. At this point,

we also reconsidered the overall approach to *MixColumns*. One change was adding rows 0 and 1 first, before the multiply by 2: so $2 \times r_0 + 3 \times r_1 + r_2 + r_3$ became $2 \times (r_0 + r_1) + r_1 + (r_2 + r_3)$; this eliminated one `xor` and one `roti`. Another improvement came from integrating *ShiftRows* and *AddRoundKey* in as well, for better instruction scheduling. The third change involved moving instructions from pipeline 0 (even), where most of them were, to pipeline 1 (odd), to allow more dual issues: the remaining two `roti` instructions were replaced by two `shufb` ones. Here some dual issues come from interleaving with other blocks, but we show only those in one block.

```
# SIMD version #2 & #4a of Shift Rows and Mix Columns and Add Round Key
        shufb   $Rrow1, $Rstate, $Rstate, $Rshiftrow1   # move bytes: row 1
{ xor           $Rrows, $Rrow1, $Rroundkey              # 1 + RK
{       shufb   $Rrow0, $Rstate, $Rstate, $Rshiftrows   # move bytes around: row 0
  xor           $Rrow01, $Rrow0, $Rrow1                 # (0+1)
{ clgtbi        $Rbit7, $Rrow01, 0x7F                   # mult 2*(0+1) in GF
{       shufb   $Rrow23, $Rrow01, $Rrow01, $Rrotrow2    # 2+3
{ xor           $Rrows, $Rrows, $Rrow23                 # 1+2+3 + RK
{       shlqbii $Rtimes2, $Rrow01, 1                    # shift 1
  andbi         $Rtimes2, $Rtimes2, 0xFE                # clear lsb (was msb)
  xorbi         $Rtimes2m, $Rtimes2, 0x1B               # mod field polynomial
  selb          $Rtimes2, $Rtimes2, $Rtimes2m, $Rbit7   # now have 2*(0+1) in GF
  xor           $Rstate, $Rrows, $Rtimes2               # 2*(0+1) + (1+2+3) + RK
```

By this point (CTR4a), all the pipeline 1 instructions were dual-issued (within the loops), though there were many pipeline 0 instructions left over. But judging by IBM's times, there was still room for improvement, by one more clock cycle per round per block. We couldn't find any way to eliminate more instructions. So the only option was to move more instructions from pipeline 0 to pipeline 1. Fortunately, we found ways to do this, using some of the quirky pipeline 1 instructions. The shuffle bytes `shufb` instruction does special things if the msb of the input byte is 1 (otherwise it picks a byte based on the 5 lowest bits); in particular, repeated application could give the sequence 0xFF $\rightarrow$ 0x80 $\rightarrow$ 0x00. In this way, we replaced one selection `selb` by two `shufb`s, though it required reversing the comparison `cgtbi`: if the msb was 0, the comparison gave 0xFF, but if the msb was 1 then 0x00; after two `shufb`s using a register full of the field polynomial byte, then the result byte was 0x00 or 0x1B respectively, the correct value to add for the Galois multiply. This saves one cycle per round per block, by eliminating a pipeline 0 command, basically matching IBM's timing. In our final version (CTR4), this approach applies for 3 of the 4 blocks each round:

```
# SIMD version #4 (3 of 4 blocks) of Shift Rows, Mix Columns, Add Round Key
        shufb   $Rrow1, $Rstate, $Rstate, $Rshiftrow1   # move bytes: row 1
{ xor           $Rrows, $Rrow1, $Rroundkey              # 1 + RK
{       shufb   $Rrow0, $Rstate, $Rstate, $Rshiftrows   # move bytes around: row 0
  xor           $Rrow01, $Rrow0, $Rrow1                 # (0+1)
{ cgtbi         $Rbit7, $Rrow01, -1                     # msb=0 -> FF; =1 -> 00
{       shufb   $Rrow23, $Rrow01, $Rrow01, $Rrotrow2    # 2+3
{ xor           $Rrows, $Rrows, $Rrow23                 # 1+2+3 + RK
{       shlqbii $Rtimes2, $Rrow01, 1                    # shift 1
# Note: in $Rmod each byte = 0x1B
{ andbi         $Rtimes2, $Rtimes2, 0xFE                # clear lsb
{       shufb   $Rbit7, $Rmod, $Rmod, $Rbit7            # FF -> 80, 00 -> 1B
{ xor           $Rrows, $Rrows, $Rtimes2                # 2*(0+1) + (1+2+3) + RK
{       shufb   $Rbit7, $Rmod, $Rmod, $Rbit7            # 80 -> 00, 1B -> 1B
  xor           $Rstate, $Rrows, $Rbit7                 # mod GF poly
```

And for our final magic trick, we were able to move one more instruction from pipeline 0, but only for one of the four blocks each round. The comparison instruction `clgtbi`, which generates a byte of all 0s or 1s based on the msb, can be replaced using "gather bits from bytes" `gbb` (gets all 16 lsb's) followed by "form select mask for bytes" `fsmb` (repeats each of those 16 bits 8 times). Since this uses the lsb rather than the msb, it must be done after the shift (which itself must become a quadword rotate instead), so requires another

quadword rotate back by a byte to put the mask back with its byte of origin. Also, since this does not reverse the sense of the comparison (as needed for the previous trick), one additional `shufb` is required to get the selection right. In short, one pipeline 0 instruction `clgtbi` of duration 2 cycles gets removed, and later *four* pipeline 1 instructions, each of duration 4 cycles, get inserted. This is why it was only possible for one out of four blocks: lots of other instructions were needed to fill in all that time; but with massive rescheduling of instructions, it worked out. This trick saved one cycle per round for every 4 blocks (and beat IBM). So for one block in CTR4, it looks like this (note that *all* pipeline 1 instructions get dual issued by interleaving with other blocks; again only dual issues within the block are shown):

```
# SIMD version #4 (1 of 4 blocks) of Shift Rows, Mix Columns, Add Round Key
        shufb   $Rrow1, $Rstate, $Rstate, $Rshiftrow1   # move bytes: row 1
        shufb   $Rrow0, $Rstate, $Rstate, $Rshiftrows   # move bytes around: row 0
  xor           $Rrow01, $Rrow0, $Rrow1                 # (0+1)
        rotqbii $Rtimes2, $Rrow01, 1                    # mul by 2
        gbb     $Rbit7, $Rtimes2                        # get lsb (was msb)
        fsmb    $Rbit7, $Rbit7                          # byte selector
        rotqbyi $Rbit7, $Rbit7, -1                      # rot back to source byte
{ xor           $Rrows, $Rrow1, $Rroundkey              # 1 + RK
\       shufb   $Rrow23, $Rrow01, $Rrow01, $Rrotrow2    # 2+3
# Note: in $Rmod each byte = 0x1B; in $Rzero each byte = 0x00
{ xor           $Rrows, $Rrows, $Rrow23                 # 1+2+3 + RK
\       shufb   $Rbit7, $Rmod, $Rmod, $Rbit7            # 00 -> 1B, FF -> 80
{ andbi         $Rtimes2, $Rtimes2, 0xFE                # clear lsb
\       shufb   $Rbit7, $Rmod, $Rmod, $Rbit7            # 1B -> 1B, 80 -> 00
{ xor           $Rrows, $Rrows, $Rtimes2                # 2*(0+1) + (1+2+3) + RK
\       shufb   $Rbit7, $Rmod, $Rzero, $Rbit7           # 1B -> 00, 00 -> 1B
  xor           $Rstate, $Rrows, $Rbit7                 # mod GF poly
```

B Initial AES CTR Assembly Code

This version was our first attempt to use the SPU Assembly language to implement AES encryption: CTR0. The SIMD instructions process all parts of a block in parallel. The *SubBytes* table lookup is based on that given in the IBM Programming Handbook. The rest is implemented in a direct manner, in a way that seems logical from a programmer's point of view, so this is fairly readable. But the instructions are not in the most efficient order from the machine's viewpoint: there is a lot of data dependency stall and no dual issues.

The format is as in the optimization examples above: named registers begin `$R` and statement labels begin `L`; pipeline 0 instructions are flush left while pipeline 1 instructions are indented.

```
## AES function, CTR mode, basic version (0)  2008 Mar 24 Mon 20:42:10
## 5 input parameters:        (NO error checking)
##     pointer to data buffer
##     pointer to Round Key buffer
##     number of data blocks (must be compatible with length of data buffer)
##     number of rounds (must be compatible with length of Round Key buffer)
##     counter value for first data block
## 1 output parameter:
##     counter value for next data block

        .file   "aes_ctr.s"
        .section        mydata,"a",@progbits
        .align  4
Sbox:
        .octa   0x637C777BF26B6FC53001672BFED7AB76
        .octa   0xCA82C97DFA5947F0ADD4A2AF9CA472C0
        .octa   0xB7FD9326363FF7CC34A5E5F171D83115
        .octa   0x04C723C31896059A071280E2EB27B275
        .octa   0x09832C1A1B6E5AA0523BD6B329E32F84
        .octa   0x53D100ED20FCB15B6ACBBE394A4C58CF
        .octa   0xD0EFAAFB434D338545F9027F503C9FA8
        .octa   0x51A3408F929D38F5BCB6DA2110FFF3D2
        .octa   0xCD0C13EC5F974417C4A77E3D645D1973
        .octa   0x60814FDC222A908846EEB814DE5E0BDB
        .octa   0xE0323A0A4906245CC2D3AC629195E479
        .octa   0xE7C8376D8DD54EA96C56F4EA657AAE08
        .octa   0xBA78252E1CA6B4C6E8DD741F4BBD8B8A
        .octa   0x703EB5664803F60E613557B986C11D9E
        .octa   0xE1F8981169D98E949B1E87E9CE5528DF
        .octa   0x8CA1890DBFE6426841992D0FB054BB16
ShiftRows:
        .octa   0x00050A0F04090E03080D02070C01060B
Incr:
        .octa   0x00000000000000000000000000000001
.text
        .align  3
        .global aes_ctr
        .type   aes_ctr, @function

        ##REGISTER DEFINITIONS##
        .set    Rin_dat, 3      # 1st param = ptr to block
        .set    Rin_key, 4      # 2nd param = ptr to keys
        .set    Rin_nb, 5       # 3rd param = number of blocks
        .set    Rin_nr, 6       # 4th param = number of rounds
        .set    Rin_ctr, 7      # 5th param = counter initial value
```

```
.set    Rout_ctr, 3     # output param = counter next value

.set    RTOP, 79        # last volatile reg
.set    Rnrounds, RTOP - 20     # # of Rounds
.set    Rincr, RTOP - 19        # increment for CTR
.set    Rdat, RTOP - 18         # 1st param = ptr to block
.set    Rroundkeys, RTOP - 17   # Keys Ptr (const)
.set    Rshiftrows, RTOP - 16   # ShiftRows (const)
.set    Rsbox0, RTOP - 15       # S-box Table (const)
.set    Rsbox1, RTOP - 14       # S-box Table (const)
.set    Rsbox2, RTOP - 13       # S-box Table (const)
.set    Rsbox3, RTOP - 12       # S-box Table (const)
.set    Rsbox4, RTOP - 11       # S-box Table (const)
.set    Rsbox5, RTOP - 10       # S-box Table (const)
.set    Rsbox6, RTOP -  9       # S-box Table (const)
.set    Rsbox7, RTOP -  8       # S-box Table (const)
.set    Rsbox8, RTOP -  7       # S-box Table (const)
.set    Rsbox9, RTOP -  6       # S-box Table (const)
.set    RsboxA, RTOP -  5       # S-box Table (const)
.set    RsboxB, RTOP -  4       # S-box Table (const)
.set    RsboxC, RTOP -  3       # S-box Table (const)
.set    RsboxD, RTOP -  2       # S-box Table (const)
.set    RsboxE, RTOP -  1       # S-box Table (const)
.set    RsboxF, RTOP -  0       # S-box Table (const)
.set    Rround,  2      # Round counter
.set    Rctr,    3      # CTR (3 = reg for return)
.set    Rsbox01, 4      #
.set    Rsbox23, 5      #
.set    Rsbox45, 6      #
.set    Rsbox67, 7      #
.set    Rsbox89, 8      #
.set    RsboxAB, 9      #
.set    RsboxCD, 10     #
.set    RsboxEF, 11     #
.set    Rstate,  12     # block State
.set    Ridx,    13     #
.set    Rblock,  14     # block counter
.set    Rbit5,   15     #
.set    Rbit6,   16     #
.set    Rbit7,   17     #
.set    NR, 15          # number of reg per block (unused)
.set    Rsbox03, Rsbox01        #
.set    Rsbox47, Rsbox23        #
.set    Rsbox8B, Rsbox45        #
.set    RsboxCF, Rsbox67        #
.set    Rsbox07, Rsbox03        #
.set    Rsbox8F, Rsbox47        #
.set    Rtimes2, Rsbox23        #
.set    Rtimes2m, Rsbox45       #
.set    Rtimes3, Rsbox67        #
.set    Rcols, Rsbox89          #
.set    Rrow1, RsboxAB          #
.set    Rrow2, RsboxCD          #
.set    Rrow3, RsboxEF          #
```

```
        .set    Rroundkey, Rbit5        #
        .set    Rdatablk, Rbit6         #

aes_ctr:
# load tables into registers
        lqr     $Rincr, Incr
        lqr     $Rshiftrows, ShiftRows
        lqr     $Rsbox0, Sbox+0x00
        lqr     $Rsbox1, Sbox+0x10
        lqr     $Rsbox2, Sbox+0x20
        lqr     $Rsbox3, Sbox+0x30
        lqr     $Rsbox4, Sbox+0x40
        lqr     $Rsbox5, Sbox+0x50
        lqr     $Rsbox6, Sbox+0x60
        lqr     $Rsbox7, Sbox+0x70
        lqr     $Rsbox8, Sbox+0x80
        lqr     $Rsbox9, Sbox+0x90
        lqr     $RsboxA, Sbox+0xA0
        lqr     $RsboxB, Sbox+0xB0
        lqr     $RsboxC, Sbox+0xC0
        lqr     $RsboxD, Sbox+0xD0
        lqr     $RsboxE, Sbox+0xE0
        lqr     $RsboxF, Sbox+0xF0
# setup so round reg counts up to zero from neg.
# then adjust pointer to roundkeys so sum points to round key
  shli          $Rnrounds, $Rin_nr, 4   # #rounds*16
  sfi           $Rnrounds, $Rnrounds, 0x10              # neg. of (#rounds-1)*16 to addr QW
  sf            $Rroundkeys, $Rnrounds, $Rin_key        # offset: roundkeys+round -> round key
# use similar count-up with block counter
  shli          $Rblock, $Rin_nb, 4     # #blocks*16
  sfi           $Rblock, $Rblock, 0                     # neg. of (#blocks)*16 to addr QW
  sf            $Rdat, $Rblock, $Rin_dat                # offset: dataptr+block -> data
  ori           $Rctr, $Rin_ctr, 0                      # move initial value to CTR
Lblockloop:
  ori           $Rstate, $Rctr, 0                       # move CTR to State
  a             $Rctr, $Rctr, $Rincr                    # increment CTR
  ori           $Rround, $Rnrounds, 0                   # initialize round counter
# ROUND 0:
# SIMD version of Add Round Key
        lqx     $Rroundkey, $Rroundkeys, $Rround        # get round key
  xor           $Rstate, $Rstate, $Rroundkey            # add it to state
Lroundloop:
  ai            $Rround, $Rround, 0x10                  # next round (*16)
# SIMD version of S-box
# presumes S-box table pre-loaded into sbox1 - sboxF
  andbi         $Ridx, $Rstate, 0x1F                    # lower 5 bits for partial lookup
        shufb   $Rsbox01, $Rsbox0, $Rsbox1, $Ridx       # partial lookup if 3 msb = 000
        shufb   $Rsbox23, $Rsbox2, $Rsbox3, $Ridx       # partial lookup if 3 msb = 001
        shufb   $Rsbox45, $Rsbox4, $Rsbox5, $Ridx       # partial lookup if 3 msb = 010
        shufb   $Rsbox67, $Rsbox6, $Rsbox7, $Ridx       # partial lookup if 3 msb = 011
        shufb   $Rsbox89, $Rsbox8, $Rsbox9, $Ridx       # partial lookup if 3 msb = 100
        shufb   $RsboxAB, $RsboxA, $RsboxB, $Ridx       # partial lookup if 3 msb = 101
        shufb   $RsboxCD, $RsboxC, $RsboxD, $Ridx       # partial lookup if 3 msb = 110
        shufb   $RsboxEF, $RsboxE, $RsboxF, $Ridx       # partial lookup if 3 msb = 111
```

```
  andbi         $Rbit5, $Rstate, 0x20                  # get next bit (#5)
  ceqbi         $Rbit5, $Rbit5, 0x20                   # form bytewise selector
  selb          $Rsbox03, $Rsbox01, $Rsbox23, $Rbit5   # partial lookup if 2 msb = 00
  selb          $Rsbox47, $Rsbox45, $Rsbox67, $Rbit5   # partial lookup if 2 msb = 01
  selb          $Rsbox8B, $Rsbox89, $RsboxAB, $Rbit5   # partial lookup if 2 msb = 10
  selb          $RsboxCF, $RsboxCD, $RsboxEF, $Rbit5   # partial lookup if 2 msb = 11
  andbi         $Rbit6, $Rstate, 0x40                  # get next bit (#6)
  ceqbi         $Rbit6, $Rbit6, 0x40                   # form bytewise selector
  selb          $Rsbox07, $Rsbox03, $Rsbox47, $Rbit6   # partial lookup if 1 msb = 0
  selb          $Rsbox8F, $Rsbox8B, $RsboxCF, $Rbit6   # partial lookup if 1 msb = 1
  clgtbi        $Rbit7, $Rstate, 0x7F                  # form selector based on msb (#7)
  selb          $Rstate, $Rsbox07, $Rsbox8F, $Rbit7    # finish table lookup
# SIMD version of shift rows
# presumes shiftrows reg pre-loaded to:
# 0x 00 05 0A 0F 04 09 0E 03 08 0D 02 07 0C 01 06 0B
        shufb   $Rstate, $Rstate, $Rstate, $Rshiftrows  # move bytes around
# SIMD version of Mix Columns
  andbi         $Rtimes2, $Rstate, 0x7F                # ain't no "shift byte"; clear msb
        shlqbii $Rtimes2, $Rtimes2, 1                  # shift block 1 bit
  xorbi         $Rtimes2m, $Rtimes2, 0x1B              # mod field polynomial
  clgtbi        $Rbit7, $Rstate, 0x7F                  # if msb = 1
  selb          $Rtimes2, $Rtimes2, $Rtimes2m, $Rbit7  # now have byte x 2 in GF
  xor           $Rtimes3, $Rtimes2, $Rstate            # also byte x 3
  roti          $Rrow1, $Rtimes3, 8                    # rotate columns and add:
  xor           $Rcols, $Rtimes2, $Rrow1               # 2 x r0 + 3 x r1
  roti          $Rrow2, $Rstate, 16
  xor           $Rcols, $Rcols, $Rrow2                 #  + 1 x r2
  roti          $Rrow3, $Rstate, 24
  xor           $Rstate, $Rcols, $Rrow3                #  + 1 x r3, and done
# SIMD version of Add Round Key
# assumes round reg has (round number - # rounds) x 16, keyaddr reg points to last key
# if fully unroll round loop, could also pre-load round keys into registers
        lqx     $Rroundkey, $Rroundkeys, $Rround       # get round key
  xor           $Rstate, $Rstate, $Rroundkey           # add it to state
        brnz    $Rround, Lroundloop                    # branch if not last round
  ai            $Rround, $Rround, 0x10                 # next round (*16)
# LAST ROUND
# SIMD version of S-box
# presumes S-box table pre-loaded into sbox1 - sboxF
  andbi         $Ridx, $Rstate, 0x1F                   # lower 5 bits for partial lookup
        shufb   $Rsbox01, $Rsbox0, $Rsbox1, $Ridx      # partial lookup if 3 msb = 000
        shufb   $Rsbox23, $Rsbox2, $Rsbox3, $Ridx      # partial lookup if 3 msb = 001
        shufb   $Rsbox45, $Rsbox4, $Rsbox5, $Ridx      # partial lookup if 3 msb = 010
        shufb   $Rsbox67, $Rsbox6, $Rsbox7, $Ridx      # partial lookup if 3 msb = 011
        shufb   $Rsbox89, $Rsbox8, $Rsbox9, $Ridx      # partial lookup if 3 msb = 100
        shufb   $RsboxAB, $RsboxA, $RsboxB, $Ridx      # partial lookup if 3 msb = 101
        shufb   $RsboxCD, $RsboxC, $RsboxD, $Ridx      # partial lookup if 3 msb = 110
        shufb   $RsboxEF, $RsboxE, $RsboxF, $Ridx      # partial lookup if 3 msb = 111
  andbi         $Rbit5, $Rstate, 0x20                  # get next bit (#5)
  ceqbi         $Rbit5, $Rbit5, 0x20                   # form bytewise selector
  selb          $Rsbox03, $Rsbox01, $Rsbox23, $Rbit5   # partial lookup if 2 msb = 00
  selb          $Rsbox47, $Rsbox45, $Rsbox67, $Rbit5   # partial lookup if 2 msb = 01
  selb          $Rsbox8B, $Rsbox89, $RsboxAB, $Rbit5   # partial lookup if 2 msb = 10
  selb          $RsboxCF, $RsboxCD, $RsboxEF, $Rbit5   # partial lookup if 2 msb = 11
```

```
  andbi         $Rbit6, $Rstate, 0x40                  # get next bit (#6)
  ceqbi         $Rbit6, $Rbit6, 0x40                   # form bytewise selector
  selb          $Rsbox07, $Rsbox03, $Rsbox47, $Rbit6   # partial lookup if 1 msb = 0
  selb          $Rsbox8F, $Rsbox8B, $RsboxCF, $Rbit6   # partial lookup if 1 msb = 1
  clgtbi        $Rbit7, $Rstate, 0x7F                  # form selector based on msb (#7)
  selb          $Rstate, $Rsbox07, $Rsbox8F, $Rbit7    # finish table lookup
# SIMD version of shift rows
# presumes shiftrows reg pre-loaded to:
# 0x 00 05 0A 0F 04 09 0E 03 08 0D 02 07 0C 01 06 0B
        shufb   $Rstate, $Rstate, $Rstate, $Rshiftrows  # move bytes around
# SIMD version of Add Round Key
# assumes round reg has (round number - # rounds) x 16, keyaddr reg points to last key
# if fully unroll round loop, could also pre-load round keys into registers
        lqx     $Rroundkey, $Rroundkeys, $Rround        # get round key
  xor           $Rstate, $Rstate, $Rroundkey           # add it to state
# use similar count-up with block counter
        lqx     $Rdatablk, $Rdat, $Rblock               # get next block of data
  xor           $Rdatablk, $Rstate, $Rdatablk          # add it to encrypted CTR
        stqx    $Rdatablk, $Rdat, $Rblock               # overwrite block of data
  ai            $Rblock, $Rblock, 0x10                 # next block
        brnz    $Rblock, Lblockloop                    # branch if not last block
        bi      $lr             # return
        .ident  "DRC"
```

C Final AES CTR Assembly Code

Here is the final version of the CTR4 code. This has been painstakingly optimized. (As a result, it is pretty much unreadable.) Within the block and round loops: *every* odd-pipeline instruction is dual-issued; there are *no* data dependency stalls; *all* branches are correctly hinted (except the final iteration of the block loop). The same is true in the setup (before the block loop), except the hint table loop has some data dependency stalls and its last iteration branch is unhinted.

The format is as in the optimization examples above: named registers begin `$R` and statement labels begin `L`; pipeline 0 instructions are flush left while pipeline 1 instructions are indented; dual-issued instruction pairs are indicated by braces.

```
## Revised AES function, CTR mode, 4-block version
## 2009 Jan 8 Thu 14:25:44 modified to take # bytes, not blocks
## 5 input parameters:        (NO error checking)
##     pointer to data buffer
##     pointer to Round Key buffer
##     number of data BYTES (was BLOCKS)
##     number of rounds
##     counter value for first data block
## 1 output parameter:
##     counter value for next data block

        .file   "aes_ctr.s"
        .section        mydata,"a",@progbits
        .align  4
Sbox:
        .octa   0x637C777BF26B6FC53001672BFED7AB76
        .octa   0xCA82C97DFA5947F0ADD4A2AF9CA472C0
        .octa   0xB7FD9326363FF7CC34A5E5F171D83115
        .octa   0x04C723C31896059A071280E2EB27B275
        .octa   0x09832C1A1B6E5AA0523BD6B329E32F84
        .octa   0x53D100ED20FCB15B6ACBBE394A4C58CF
        .octa   0xD0EFAAFB434D338545F9027F503C9FA8
        .octa   0x51A3408F929D38F5BCB6DA2110FFF3D2
        .octa   0xCD0C13EC5F974417C4A77E3D645D1973
        .octa   0x60814FDC222A908846EEB814DE5E0BDB
        .octa   0xE0323A0A4906245CC2D3AC629195E479
        .octa   0xE7C8376D8DD54EA96C56F4EA657AAE08
        .octa   0xBA78252E1CA6B4C6E8DD741F4BBD8B8A
        .octa   0x703EB5664803F60E613557B986C11D9E
        .octa   0xE1F8981169D98E949B1E87E9CE5528DF
        .octa   0x8CA1890DBFE6426841992D0FB054BB16
ShiftRows:
        .octa   0x00050A0F04090E03080D02070C01060B      # standard (row 0)
        .octa   0x050A0F00090E03040D02070801060B0C      # row 1 on top
RotRow2:
        .octa   0x02030001060704050A0B08090E0F0C0D      # rotate row 2 to top
# Note: to rotate word by bytes using shufb:
# 000102030405060708090A0B0C0D0E0F
# 0102030005060704090A0B080D0E0F0C
# 02030001060704050A0B08090E0F0C0D
# 03000102070405060B08090A0F0C0D0E
SaveReg:                                                # to save registers
        .fill   4*4, 4, 0                               # (size cannot exceed 8)
BranchHints:                                            # for dynamic br. hints
```

```
        .fill   16*4, 4, 0                              # (size cannot exceed 8)
.text
        .global aes_ctr
        .type   aes_ctr, @function

        ##REGISTER DEFINITIONS##
# in/out params
        .set    Rin_dat,        3       # 1st param = ptr to block
        .set    Rin_key,        4       # 2nd param = ptr to keys
        .set    Rin_nb,         5       # 3rd param = number of bytes
        .set    Rin_nr,         6       # 4th param = number of rounds
        .set    Rin_ctr,        7       # 5th param = counter initial value
        .set    Rout_ctr,       3       # output param = counter next value
# per block values
        .set    Rsbox01,         2      #
        .set    Rsbox23,        13      #
        .set    Rsbox45,         4      #
        .set    Rsbox67,         5      #
        .set    Rsbox89,         6      #
        .set    RsboxAB,         7      #
        .set    RsboxCD,         8      #
        .set    RsboxEF,         9      #
        .set    Rbit5,          10      #
        .set    Rbit6,          11      #
        .set    Rbit7,          12      #
        .set    Rctr,            3      # CTR = output (=1st input)
        .set    Rstate,         Rbit7   # block State
        .set    Ridx,           RsboxEF #
        .set    Rsbox03,        Rsbox01 #
        .set    Rsbox47,        Rsbox45 #
        .set    Rsbox8B,        Rsbox89 #
        .set    RsboxCF,        RsboxCD #
        .set    Rsbox07,        Rsbox01 #
        .set    Rsbox8F,        Rsbox89 #
        .set    Rrow0,           2      #
        .set    Rrow1,          13      #
        .set    Rrow01,          4      #
        .set    Rrow23,          5      #
        .set    Rrows,           6      #
        .set    Rtimes2,         7      #
        .set    Rzero,           8      # temporary zero reg
        .set    Rdat,           Rsbox23 # ptr to data for block
        .set    Rdatablk,       Rbit5   #

        .set    NR,             12      # number of reg per block

# independent of block:
        .set    Rblockout,      Rsbox01 # temporary copy of block counter
        .set    Rhint,          51      # branch hint
        .set    Rhints,         52      # branch hint table
        .set    Rroundkey0,     53      #
        .set    Rblock,         54      # block counter (0th block of set)
        .set    Rround,         55      # Round counter
        .set    Rroundkey,      56      #
```

```
# constant values:
        .set    Rmod,           50      # for mod GF poly
        .set    Rblkpad,        57      # block pad
        .set    Rnrounds,       58      # # of Rounds
        .set    Rroundkeys,     59      # Keys Ptr (const)
        .set    Rincr,          60      # increment for CTR
        .set    Rshiftrows,     61      # ShiftRows (const)
        .set    Rshiftrow1,     62      #
        .set    Rrotrow2,       63      #
        .set    Rsbox0,         64      # S-box Table (const)
        .set    Rsbox1,         65      # S-box Table (const)
        .set    Rsbox2,         66      # S-box Table (const)
        .set    Rsbox3,         67      # S-box Table (const)
        .set    Rsbox4,         68      # S-box Table (const)
        .set    Rsbox5,         69      # S-box Table (const)
        .set    Rsbox6,         70      # S-box Table (const)
        .set    Rsbox7,         71      # S-box Table (const)
        .set    Rsbox8,         72      # S-box Table (const)
        .set    Rsbox9,         73      # S-box Table (const)
        .set    RsboxA,         74      # S-box Table (const)
        .set    RsboxB,         75      # S-box Table (const)
        .set    RsboxC,         76      # S-box Table (const)
        .set    RsboxD,         77      # S-box Table (const)
        .set    RsboxE,         78      # S-box Table (const)
        .set    RsboxF,         79      # S-box Table (const)

        .align  3
aes_ctr:
# setup so round reg counts up to zero from neg.
# then adjust pointer to roundkeys so sum points to round key
# use similar count-up with block counter
# for 4 blocks at once, keep track of padding at end
# load tables into registers
{ shli          $Rnrounds, $Rin_nr, 4                   # *16 to address quadwords
  hbrr          Lhinttabloop_end, Lhinttabloop          # hint for hint loop
{ il            $Rincr, 1
  lqr           $Rsbox0, Sbox+0x00
{ ai            $Rblkpad, $Rin_nb, 15                   # round up to whole blocks
  lqr           $Rsbox1, Sbox+0x10
{ sfi           $Rblock, $Rin_nb, 0                     # -(#bytes)
  rotqmbyi      $Rincr, $Rincr, -12                     # move to rightmost word
{ sfi           $Rnrounds, $Rnrounds, 0x10              # neg. of (#rounds-1)*16
  lqr           $Rsbox2, Sbox+0x20
{ andi          $Rblkpad, $Rblkpad, 48                  # [ (# blocks) % 4 ] * 16
  lqr           $Rsbox3, Sbox+0x30
{ sf            $Rroundkeys, $Rnrounds, $Rin_key        # roundkeys+round -> round key
  lqr           $Rsbox4, Sbox+0x40
{ andi          $Rblock, $Rblock, -64                   # round up to (4-block)s, neg.
  lqr           $Rsbox5, Sbox+0x50
{ ai            $Rroundkeys, $Rroundkeys, 0x10          # adjust since lookup before incr
  lqr           $Rsbox6, Sbox+0x60
{ sf            $Rdat, $Rblock, $Rin_dat                # dataptr+block -> data
  lqr           $Rsbox7, Sbox+0x70
```

```
{ ai            $Rctr, $Rin_ctr, 0                      # move CTR (clobber Rin_dat!)
\       biz     $Rin_nb, $lr                            # return if no bytes
{ ai            $(Rdat + NR), $Rdat, 0x10               # data ptr for block 1
\       lqr     $Rsbox8, Sbox+0x80
{ a             $(Rctr + NR), $Rin_ctr, $Rincr          # increment CTR for block 1
\       lqr     $Rsbox9, Sbox+0x90
{ ai            $(Rdat + 2*NR), $Rdat, 0x20             # data ptr for block 2
\       lqr     $RsboxA, Sbox+0xA0
{ a             $(Rctr + 2*NR), $(Rctr + NR), $Rincr    # increment CTR for block 2
\       lqr     $RsboxB, Sbox+0xB0
{ ai            $(Rdat + 3*NR), $Rdat, 0x30             # data ptr for block 3
\       lqr     $RsboxC, Sbox+0xC0
{ a             $(Rctr + 3*NR), $(Rctr + 2*NR), $Rincr  # increment CTR for block 3
\       lqr     $RsboxD, Sbox+0xD0
{ rotmi         $Rblkpad, $Rblkpad, -4                  # save info on (# blocks) % 4
\       lqr     $RsboxE, Sbox+0xE0
{ shli          $Rincr, $Rincr, 2                       # shift incr for 4 blocks
\       lqr     $RsboxF, Sbox+0xF0
{ ilh           $Rmod, 0x1B1B                           # 00 -> 1B -> 1B
\       lqd     $Rroundkey0, 0($Rin_key)                # get round key #0
{ ila           $Rhints, BranchHints
\       lqr     $Rshiftrows, ShiftRows
{ ila           $Rhint, Lroundloop
\       lqr     $Rshiftrow1, ShiftRows+0x10
{ sf            $Rhints, $Rnrounds, $Rhints             # hints+round -> round hint
\       lqr     $Rrotrow2, RotRow2
{ ai            $Rround, $Rnrounds, 0                   # initialize round counter
\       stqr    $Rdat, SaveReg+0x00                     # save data ptr
{ ila           $8, Lroundloop_end + 4                  # address not to loop
\       stqr    $(Rdat + NR), SaveReg+0x10              # save data ptr
        stqr    $(Rdat + 2*NR), SaveReg+0x20            # save data ptr
Lhinttabloop:
        stqx    $Rhint, $Rhints, $Rround                # put hint for each round
  ai            $Rround, $Rround, 0x10                  # next round (*16)
Lhinttabloop_end:
        brnz    $Rround, Lhinttabloop                   # branch if not last round
        .align  3
{ xor           $(Rstate + NR), $(Rctr + NR), $Rroundkey0
\       stqx    $Rhint, $Rhints, $Rround                # put hint for next round loop
{ xor           $Rstate, $Rctr, $Rroundkey0             # add RK0 to CTR
\       stqd    $8, -32($Rhints)                        # store hint not to loop
{ xor           $(Rstate + 2*NR), $(Rctr + 2*NR), $Rroundkey0
\       shlqbyi $Rround, $Rnrounds, 0                   # initialize round counter
# ROUND 0 for first set of blocks:
{ xor           $(Rstate + 3*NR), $(Rctr + 3*NR), $Rroundkey0
\       stqr    $(Rdat + 3*NR), SaveReg+0x30            # save data ptr
Lblockloop:
        .align  3
# initialize:
  a             $Rctr, $Rctr, $Rincr                    # increment CTR
  a             $(Rctr + NR), $(Rctr + NR), $Rincr
  a             $(Rctr + 2*NR), $(Rctr + 2*NR), $Rincr
  a             $(Rctr + 3*NR), $(Rctr + 3*NR), $Rincr
Lroundloop:
```

```
# SIMD version of S-box
        .align  3
{ andbi         $Ridx, $Rstate, 0x1F                    # lower 5 bits (0-4) for lookup
  hbr     Lroundloop_end, $Rhint                        # hint for round loop
{ andbi         $(Ridx + NR), $(Rstate + NR), 0x1F
  lqx     $Rhint, $Rhints, $Rround                      # get hint for next round
{ andbi         $(Ridx + 2*NR), $(Rstate + 2*NR), 0x1F
  shufb   $Rsbox01, $Rsbox0, $Rsbox1, $Ridx             # partial lookup if 3 msb = 000
{ andbi         $(Ridx + 3*NR), $(Rstate + 3*NR), 0x1F
  shufb   $Rsbox23, $Rsbox2, $Rsbox3, $Ridx             # partial lookup if 3 msb = 001
{ andbi         $Rbit5, $Rstate, 0x20                   # get next bit (#5)
  shufb   $Rsbox45, $Rsbox4, $Rsbox5, $Ridx             # partial lookup if 3 msb = 010
{ andbi         $Rbit6, $Rstate, 0x40                   # get next bit (#6)
  shufb   $Rsbox67, $Rsbox6, $Rsbox7, $Ridx             # partial lookup if 3 msb = 011
{ ceqbi         $Rbit5, $Rbit5, 0x20                    # form bytewise selector
  shufb   $Rsbox89, $Rsbox8, $Rsbox9, $Ridx             # partial lookup if 3 msb = 100
{ ceqbi         $Rbit6, $Rbit6, 0x40                    # form bytewise selector
  shufb   $RsboxAB, $RsboxA, $RsboxB, $Ridx             # partial lookup if 3 msb = 101
{ clgtbi        $Rbit7, $Rstate, 0x7F                   # form selector based on msb (#7)
  shufb   $RsboxCD, $RsboxC, $RsboxD, $Ridx             # partial lookup if 3 msb = 110
{ selb          $Rsbox03, $Rsbox01, $Rsbox23, $Rbit5    # partial lookup if 2 msb = 00
  shufb   $RsboxEF, $RsboxE, $RsboxF, $Ridx             # partial lookup if 3 msb = 111
{ selb          $Rsbox47, $Rsbox45, $Rsbox67, $Rbit5    # partial lookup if 2 msb = 01
  shufb   $(Rsbox23 + NR), $Rsbox2, $Rsbox3, $(Ridx + NR)
{ selb          $Rsbox8B, $Rsbox89, $RsboxAB, $Rbit5    # partial lookup if 2 msb = 10
  shufb   $(Rsbox23 + 2*NR), $Rsbox2, $Rsbox3, $(Ridx + 2*NR)
{ selb          $Rsbox07, $Rsbox03, $Rsbox47, $Rbit6    # partial lookup if 1 msb = 0
  shufb   $(Rsbox67 + NR), $Rsbox6, $Rsbox7, $(Ridx + NR)
{ selb          $RsboxCF, $RsboxCD, $RsboxEF, $Rbit5    # partial lookup if 2 msb = 11
  shufb   $(Rsbox89 + 2*NR), $Rsbox8, $Rsbox9, $(Ridx + 2*NR)
{ andbi         $(Rbit5 + NR), $(Rstate + NR), 0x20
  shufb   $(Rsbox01 + NR), $Rsbox0, $Rsbox1, $(Ridx + NR)
{ selb          $Rsbox8F, $Rsbox8B, $RsboxCF, $Rbit6    # partial lookup if 1 msb = 1
  shufb   $(RsboxAB + 3*NR), $RsboxA, $RsboxB, $(Ridx + 3*NR)
{ andbi         $(Rbit5 + 2*NR), $(Rstate + 2*NR), 0x20
  shufb   $(Rsbox01 + 2*NR), $Rsbox0, $Rsbox1, $(Ridx + 2*NR)
{ selb          $Rstate, $Rsbox07, $Rsbox8F, $Rbit7     # finish table lookup
  shufb   $(RsboxCD + 3*NR), $RsboxC, $RsboxD, $(Ridx + 3*NR)
{ andbi         $(Rbit5 + 3*NR), $(Rstate + 3*NR), 0x20
  shufb   $(Rsbox01 + 3*NR), $Rsbox0, $Rsbox1, $(Ridx + 3*NR)
{ ceqbi         $(Rbit5 + NR), $(Rbit5 + NR), 0x20
  shufb   $Rrow1, $Rstate, $Rstate, $Rshiftrow1   # move bytes: row 1
{ ceqbi         $(Rbit5 + 2*NR), $(Rbit5 + 2*NR), 0x20
  shufb   $Rrow0, $Rstate, $Rstate, $Rshiftrows   # move bytes around: row 0
{ ceqbi         $(Rbit5 + 3*NR), $(Rbit5 + 3*NR), 0x20
  shufb   $(Rsbox23 + 3*NR), $Rsbox2, $Rsbox3, $(Ridx + 3*NR)
{ andbi         $(Rbit6 + NR), $(Rstate + NR), 0x40
  shufb   $(Rsbox45 + NR), $Rsbox4, $Rsbox5, $(Ridx + NR)
{ andbi         $(Rbit6 + 2*NR), $(Rstate + 2*NR), 0x40
  shufb   $(Rsbox45 + 2*NR), $Rsbox4, $Rsbox5, $(Ridx + 2*NR)
{ xor           $Rrow01, $Rrow0, $Rrow1                 # (0+1)
  lqx     $Rroundkey, $Rroundkeys, $Rround        # get round key
{ andbi         $(Rbit6 + 3*NR), $(Rstate + 3*NR), 0x40
  shufb   $(Rsbox45 + 3*NR), $Rsbox4, $Rsbox5, $(Ridx + 3*NR)
```

```
{ ceqbi         $(Rbit6 + NR), $(Rbit6 + NR), 0x40
|       rotqbii $Rtimes2, $Rrow01, 1                    # mul by 2
{ ceqbi         $(Rbit6 + 2*NR), $(Rbit6 + 2*NR), 0x40
|       shufb   $(Rsbox67 + 2*NR), $Rsbox6, $Rsbox7, $(Ridx + 2*NR)
{ ceqbi         $(Rbit6 + 3*NR), $(Rbit6 + 3*NR), 0x40
|       shufb   $(Rsbox67 + 3*NR), $Rsbox6, $Rsbox7, $(Ridx + 3*NR)
{ clgtbi        $(Rbit7 + NR), $(Rstate + NR), 0x7F
|       shufb   $(Rsbox89 + NR), $Rsbox8, $Rsbox9, $(Ridx + NR)
{ clgtbi        $(Rbit7 + 2*NR), $(Rstate + 2*NR), 0x7F
|       gbb     $Rbit7, $Rtimes2                       # get lsb (was msb)
{ clgtbi        $(Rbit7 + 3*NR), $(Rstate + 3*NR), 0x7F
|       shufb   $(Rsbox89 + 3*NR), $Rsbox8, $Rsbox9, $(Ridx + 3*NR)
{ selb          $(Rsbox03 + NR), $(Rsbox01 + NR), $(Rsbox23 + NR), $(Rbit5 + NR)
|       shufb   $(RsboxAB + NR), $RsboxA, $RsboxB, $(Ridx + NR)
{ selb          $(Rsbox03 + 2*NR), $(Rsbox01 + 2*NR), $(Rsbox23 + 2*NR), $(Rbit5 + 2*NR)
|       shufb   $(RsboxAB + 2*NR), $RsboxA, $RsboxB, $(Ridx + 2*NR)
{ selb          $(Rsbox03 + 3*NR), $(Rsbox01 + 3*NR), $(Rsbox23 + 3*NR), $(Rbit5 + 3*NR)
|       fsmb    $Rbit7, $Rbit7                         # byte selector
{ selb          $(Rsbox47 + NR), $(Rsbox45 + NR), $(Rsbox67 + NR), $(Rbit5 + NR)
|       shufb   $(RsboxCD + NR), $RsboxC, $RsboxD, $(Ridx + NR)
{ selb          $(Rsbox47 + 2*NR), $(Rsbox45 + 2*NR), $(Rsbox67 + 2*NR), $(Rbit5 + 2*NR)
|       shufb   $(RsboxCD + 2*NR), $RsboxC, $RsboxD, $(Ridx + 2*NR)
{ selb          $(Rsbox47 + 3*NR), $(Rsbox45 + 3*NR), $(Rsbox67 + 3*NR), $(Rbit5 + 3*NR)
|       shufb   $(RsboxEF + NR), $RsboxE, $RsboxF, $(Ridx + NR)
{ selb          $(Rsbox8B + NR), $(Rsbox89 + NR), $(RsboxAB + NR), $(Rbit5 + NR)
|       rotqbyi $Rbit7, $Rbit7, -1                     # rot back to source byte
{ selb          $(Rsbox8B + 2*NR), $(Rsbox89 + 2*NR), $(RsboxAB + 2*NR), $(Rbit5 + 2*NR)
|       shufb   $(RsboxEF + 2*NR), $RsboxE, $RsboxF, $(Ridx + 2*NR)
{ selb          $(Rsbox8B + 3*NR), $(Rsbox89 + 3*NR), $(RsboxAB + 3*NR), $(Rbit5 + 3*NR)
|       shufb   $(RsboxEF + 3*NR), $RsboxE, $RsboxF, $(Ridx + 3*NR)
  selb          $(RsboxCF + NR), $(RsboxCD + NR), $(RsboxEF + NR), $(Rbit5 + NR)
  selb          $(Rsbox07 + NR), $(Rsbox03 + NR), $(Rsbox47 + NR), $(Rbit6 + NR)
  selb          $(Rsbox8F + NR), $(Rsbox8B + NR), $(RsboxCF + NR), $(Rbit6 + NR)
  selb          $(RsboxCF + 2*NR), $(RsboxCD + 2*NR), $(RsboxEF + 2*NR), $(Rbit5 + 2*NR)
  selb          $(Rstate + NR), $(Rsbox07 + NR), $(Rsbox8F + NR), $(Rbit7 + NR)
  selb          $(Rsbox07 + 2*NR), $(Rsbox03 + 2*NR), $(Rsbox47 + 2*NR), $(Rbit6 + 2*NR)
        .align  3
{ selb          $(Rsbox8F + 2*NR), $(Rsbox8B + 2*NR), $(RsboxCF + 2*NR), $(Rbit6 + 2*NR)
|       shufb   $(Rrow1 + NR), $(Rstate + NR), $(Rstate + NR), $Rshiftrow1
{ selb          $(RsboxCF + 3*NR), $(RsboxCD + 3*NR), $(RsboxEF + 3*NR), $(Rbit5 + 3*NR)
|       shufb   $(Rrow0 + NR), $(Rstate + NR), $(Rstate + NR), $Rshiftrows
  selb          $(Rstate + 2*NR), $(Rsbox07 + 2*NR), $(Rsbox8F + 2*NR), $(Rbit7 + 2*NR)
  selb          $(Rsbox07 + 3*NR), $(Rsbox03 + 3*NR), $(Rsbox47 + 3*NR), $(Rbit6 + 3*NR)
{ selb          $(Rsbox8F + 3*NR), $(Rsbox8B + 3*NR), $(RsboxCF + 3*NR), $(Rbit6 + 3*NR)
|       shufb   $(Rrow1 + 2*NR), $(Rstate + 2*NR), $(Rstate + 2*NR), $Rshiftrow1
{ xor           $(Rrows + NR), $(Rrow1 + NR), $Rroundkey
|       shufb   $(Rrow0 + 2*NR), $(Rstate + 2*NR), $(Rstate + 2*NR), $Rshiftrows
  selb          $(Rstate + 3*NR), $(Rsbox07 + 3*NR), $(Rsbox8F + 3*NR), $(Rbit7 + 3*NR)
  xor           $(Rrow01 + NR), $(Rrow0 + NR), $(Rrow1 + NR)
{ xor           $(Rrows + 2*NR), $(Rrow1 + 2*NR), $Rroundkey
|       shufb   $(Rrow1 + 3*NR), $(Rstate + 3*NR), $(Rstate + 3*NR), $Rshiftrow1
{ xor           $(Rrow01 + 2*NR), $(Rrow0 + 2*NR), $(Rrow1 + 2*NR)
|       shufb   $(Rrow0 + 3*NR), $(Rstate + 3*NR), $(Rstate + 3*NR), $Rshiftrows
# SIMD version of Shift Rows and Mix Columns and Add Round Key
```

```
{ ai            $Rround, $Rround, 0x10                  # next round (*16)
{       fsmbi   $Rzero, 0
{ xor           $Rrows, $Rrow1, $Rroundkey              # 1 + RK
{       shufb   $Rrow23, $Rrow01, $Rrow01, $Rrotrow2    # 2+3
  xor           $(Rrows + 3*NR), $(Rrow1 + 3*NR), $Rroundkey
  xor           $(Rrow01 + 3*NR), $(Rrow0 + 3*NR), $(Rrow1 + 3*NR)
{ cgtbi         $(Rbit7 + NR), $(Rrow01 + NR), -1
{       shufb   $(Rrow23 + NR), $(Rrow01 + NR), $(Rrow01 + NR), $Rrotrow2
{ cgtbi         $(Rbit7 + 2*NR), $(Rrow01 + 2*NR), -1
{       shufb   $(Rrow23 + 2*NR), $(Rrow01 + 2*NR), $(Rrow01 + 2*NR), $Rrotrow2
{ cgtbi         $(Rbit7 + 3*NR), $(Rrow01 + 3*NR), -1
{       shufb   $(Rrow23 + 3*NR), $(Rrow01 + 3*NR), $(Rrow01 + 3*NR), $Rrotrow2
{ xor           $Rrows, $Rrows, $Rrow23                 # 1+2+3 + RK
{       shufb   $Rbit7, $Rmod, $Rmod, $Rbit7            # 00 -> 1B, FF -> 80
{ xor           $(Rrows + NR), $(Rrows + NR), $(Rrow23 + NR)
{       shlqbii $(Rtimes2 + NR), $(Rrow01 + NR), 1
{ xor           $(Rrows + 2*NR), $(Rrows + 2*NR), $(Rrow23 + 2*NR)
{       shlqbii $(Rtimes2 + 2*NR), $(Rrow01 + 2*NR), 1
{ xor           $(Rrows + 3*NR), $(Rrows + 3*NR), $(Rrow23 + 3*NR)
{       shlqbii $(Rtimes2 + 3*NR), $(Rrow01 + 3*NR), 1
{ andbi         $Rtimes2, $Rtimes2, 0xFE                # clear lsb
{       shufb   $Rbit7, $Rmod, $Rmod, $Rbit7            # 1B -> 1B, 80 -> 00
{ andbi         $(Rtimes2 + NR), $(Rtimes2 + NR), 0xFE
{       shufb   $(Rbit7 + NR), $Rmod, $Rmod, $(Rbit7 + NR)
{ andbi         $(Rtimes2 + 2*NR), $(Rtimes2 + 2*NR), 0xFE
{       shufb   $(Rbit7 + 2*NR), $Rmod, $Rmod, $(Rbit7 + 2*NR)
{ andbi         $(Rtimes2 + 3*NR), $(Rtimes2 + 3*NR), 0xFE
{       shufb   $(Rbit7 + 3*NR), $Rmod, $Rmod, $(Rbit7 + 3*NR)
{ xor           $Rrows, $Rrows, $Rtimes2                # 2*(0+1) + (1+2+3) + RK
{       shufb   $Rbit7, $Rmod, $Rzero, $Rbit7           # 1B -> 00, 00 -> 1B
{ xor           $(Rrows + NR), $(Rrows + NR), $(Rtimes2 + NR)
{       shufb   $(Rbit7 + NR), $Rmod, $Rmod, $(Rbit7 + NR)
{ xor           $(Rrows + 2*NR), $(Rrows + 2*NR), $(Rtimes2 + 2*NR)
{       shufb   $(Rbit7 + 2*NR), $Rmod, $Rmod, $(Rbit7 + 2*NR)
{ xor           $(Rrows + 3*NR), $(Rrows + 3*NR), $(Rtimes2 + 3*NR)
{       shufb   $(Rbit7 + 3*NR), $Rmod, $Rmod, $(Rbit7 + 3*NR)
  xor           $Rstate, $Rrows, $Rbit7                 # mod GF poly
  xor           $(Rstate + NR), $(Rrows + NR), $(Rbit7 + NR)
  xor           $(Rstate + 2*NR), $(Rrows + 2*NR), $(Rbit7 + 2*NR)
        .align  3
{ xor           $(Rstate + 3*NR), $(Rrows + 3*NR), $(Rbit7 + 3*NR)
{Lroundloop_end:
{       brnz    $Rround, Lroundloop                     # branch if not last round
 # LAST ROUND
 # SIMD version of S-box
        .align  3
{ andbi         $Ridx, $Rstate, 0x1F                    # lower 5 bits (0-4) for lookup
{       hbrr    Lblockloop_end, Lblockloop              # hint for block loop
{ andbi         $(Ridx + NR), $(Rstate + NR), 0x1F
{       lqd     $Rroundkey, 0($Rroundkeys)              # get round key
  andbi         $(Ridx + 2*NR), $(Rstate + 2*NR), 0x1F
  andbi         $(Ridx + 3*NR), $(Rstate + 3*NR), 0x1F
{ andbi         $Rbit5, $Rstate, 0x20                   # get next bit (#5)
{       shufb   $Rsbox01, $Rsbox0, $Rsbox1, $Ridx       # partial lookup if 3 msb = 000
```

```
{ andbi         $(Rbit5 + NR), $(Rstate + NR), 0x20
  shufb         $(Rsbox01 + NR), $Rsbox0, $Rsbox1, $(Ridx + NR)
{ andbi         $(Rbit5 + 2*NR), $(Rstate + 2*NR), 0x20
  shufb         $(Rsbox01 + 2*NR), $Rsbox0, $Rsbox1, $(Ridx + 2*NR)
{ andbi         $(Rbit5 + 3*NR), $(Rstate + 3*NR), 0x20
  shufb         $(Rsbox01 + 3*NR), $Rsbox0, $Rsbox1, $(Ridx + 3*NR)
{ ceqbi         $Rbit5, $Rbit5, 0x20                    # form bytewise selector
  shufb         $Rsbox23, $Rsbox2, $Rsbox3, $Ridx       # partial lookup if 3 msb = 001
{ ceqbi         $(Rbit5 + NR), $(Rbit5 + NR), 0x20
  shufb         $(Rsbox23 + NR), $Rsbox2, $Rsbox3, $(Ridx + NR)
{ ceqbi         $(Rbit5 + 2*NR), $(Rbit5 + 2*NR), 0x20
  shufb         $(Rsbox23 + 2*NR), $Rsbox2, $Rsbox3, $(Ridx + 2*NR)
{ ceqbi         $(Rbit5 + 3*NR), $(Rbit5 + 3*NR), 0x20
  shufb         $(Rsbox23 + 3*NR), $Rsbox2, $Rsbox3, $(Ridx + 3*NR)
{ andbi         $Rbit6, $Rstate, 0x40                   # get next bit (#6)
  shufb         $Rsbox45, $Rsbox4, $Rsbox5, $Ridx       # partial lookup if 3 msb = 010
{ andbi         $(Rbit6 + NR), $(Rstate + NR), 0x40
  shufb         $(Rsbox45 + NR), $Rsbox4, $Rsbox5, $(Ridx + NR)
{ andbi         $(Rbit6 + 2*NR), $(Rstate + 2*NR), 0x40
  shufb         $(Rsbox45 + 2*NR), $Rsbox4, $Rsbox5, $(Ridx + 2*NR)
{ andbi         $(Rbit6 + 3*NR), $(Rstate + 3*NR), 0x40
  shufb         $(Rsbox45 + 3*NR), $Rsbox4, $Rsbox5, $(Ridx + 3*NR)
{ ceqbi         $Rbit6, $Rbit6, 0x40                    # form bytewise selector
  shufb         $Rsbox67, $Rsbox6, $Rsbox7, $Ridx       # partial lookup if 3 msb = 011
{ ceqbi         $(Rbit6 + NR), $(Rbit6 + NR), 0x40
  shufb         $(Rsbox67 + NR), $Rsbox6, $Rsbox7, $(Ridx + NR)
{ ceqbi         $(Rbit6 + 2*NR), $(Rbit6 + 2*NR), 0x40
  shufb         $(Rsbox67 + 2*NR), $Rsbox6, $Rsbox7, $(Ridx + 2*NR)
{ ceqbi         $(Rbit6 + 3*NR), $(Rbit6 + 3*NR), 0x40
  shufb         $(Rsbox67 + 3*NR), $Rsbox6, $Rsbox7, $(Ridx + 3*NR)
{ clgtbi        $Rbit7, $Rstate, 0x7F                   # form selector based on msb (#7)
  shufb         $Rsbox89, $Rsbox8, $Rsbox9, $Ridx       # partial lookup if 3 msb = 100
{ clgtbi        $(Rbit7 + NR), $(Rstate + NR), 0x7F
  shufb         $(Rsbox89 + NR), $Rsbox8, $Rsbox9, $(Ridx + NR)
{ clgtbi        $(Rbit7 + 2*NR), $(Rstate + 2*NR), 0x7F
  shufb         $(Rsbox89 + 2*NR), $Rsbox8, $Rsbox9, $(Ridx + 2*NR)
{ clgtbi        $(Rbit7 + 3*NR), $(Rstate + 3*NR), 0x7F
  shufb         $(Rsbox89 + 3*NR), $Rsbox8, $Rsbox9, $(Ridx + 3*NR)
{ selb          $Rsbox03, $Rsbox01, $Rsbox23, $Rbit5    # partial lookup if 2 msb = 00
  shufb         $RsboxAB, $RsboxA, $RsboxB, $Ridx       # partial lookup if 3 msb = 101
{ selb          $(Rsbox03 + NR), $(Rsbox01 + NR), $(Rsbox23 + NR), $(Rbit5 + NR)
  shufb         $(RsboxAB + NR), $RsboxA, $RsboxB, $(Ridx + NR)
{ selb          $(Rsbox03 + 2*NR), $(Rsbox01 + 2*NR), $(Rsbox23 + 2*NR), $(Rbit5 + 2*NR)
  shufb         $(RsboxAB + 2*NR), $RsboxA, $RsboxB, $(Ridx + 2*NR)
{ selb          $(Rsbox03 + 3*NR), $(Rsbox01 + 3*NR), $(Rsbox23 + 3*NR), $(Rbit5 + 3*NR)
  shufb         $(RsboxAB + 3*NR), $RsboxA, $RsboxB, $(Ridx + 3*NR)
{ selb          $Rsbox47, $Rsbox45, $Rsbox67, $Rbit5    # partial lookup if 2 msb = 01
  shufb         $RsboxCD, $RsboxC, $RsboxD, $Ridx       # partial lookup if 3 msb = 110
{ selb          $(Rsbox47 + NR), $(Rsbox45 + NR), $(Rsbox67 + NR), $(Rbit5 + NR)
  shufb         $(RsboxCD + NR), $RsboxC, $RsboxD, $(Ridx + NR)
{ selb          $(Rsbox47 + 2*NR), $(Rsbox45 + 2*NR), $(Rsbox67 + 2*NR), $(Rbit5 + 2*NR)
  shufb         $(RsboxCD + 2*NR), $RsboxC, $RsboxD, $(Ridx + 2*NR)
{ selb          $(Rsbox47 + 3*NR), $(Rsbox45 + 3*NR), $(Rsbox67 + 3*NR), $(Rbit5 + 3*NR)
  shufb         $(RsboxCD + 3*NR), $RsboxC, $RsboxD, $(Ridx + 3*NR)
```

```
{ selb          $Rsbox8B, $Rsbox89, $RsboxAB, $Rbit5    # partial lookup if 2 msb = 10
\       shufb   $RsboxEF, $RsboxE, $RsboxF, $Ridx       # partial lookup if 3 msb = 111
{ selb          $(Rsbox8B + NR), $(Rsbox89 + NR), $(RsboxAB + NR), $(Rbit5 + NR)
\       shufb   $(RsboxEF + NR), $RsboxE, $RsboxF, $(Ridx + NR)
{ selb          $(Rsbox8B + 2*NR), $(Rsbox89 + 2*NR), $(RsboxAB + 2*NR), $(Rbit5 + 2*NR)
\       shufb   $(RsboxEF + 2*NR), $RsboxE, $RsboxF, $(Ridx + 2*NR)
{ selb          $(Rsbox8B + 3*NR), $(Rsbox89 + 3*NR), $(RsboxAB + 3*NR), $(Rbit5 + 3*NR)
\       shufb   $(RsboxEF + 3*NR), $RsboxE, $RsboxF, $(Ridx + 3*NR)
{ selb          $RsboxCF, $RsboxCD, $RsboxEF, $Rbit5    # partial lookup if 2 msb = 11
\       lqr     $Rdat, SaveReg+0x00                     # get data ptr
{ selb          $(RsboxCF + NR), $(RsboxCD + NR), $(RsboxEF + NR), $(Rbit5 + NR)
\       lqr     $(Rdat + NR), SaveReg+0x10              # get data ptr
{ selb          $(RsboxCF + 2*NR), $(RsboxCD + 2*NR), $(RsboxEF + 2*NR), $(Rbit5 + 2*NR)
\       lqr     $(Rdat + 2*NR), SaveReg+0x20            # get data ptr
{ selb          $(RsboxCF + 3*NR), $(RsboxCD + 3*NR), $(RsboxEF + 3*NR), $(Rbit5 + 3*NR)
\       lqr     $(Rdat + 3*NR), SaveReg+0x30            # get data ptr
  selb          $Rsbox07, $Rsbox03, $Rsbox47, $Rbit6    # partial lookup if 1 msb = 0
  selb          $(Rsbox07 + NR), $(Rsbox03 + NR), $(Rsbox47 + NR), $(Rbit6 + NR)
  selb          $(Rsbox07 + 2*NR), $(Rsbox03 + 2*NR), $(Rsbox47 + 2*NR), $(Rbit6 + 2*NR)
  selb          $(Rsbox07 + 3*NR), $(Rsbox03 + 3*NR), $(Rsbox47 + 3*NR), $(Rbit6 + 3*NR)
{ selb          $Rsbox8F, $Rsbox8B, $RsboxCF, $Rbit6    # partial lookup if 1 msb = 1
\       lqx     $Rdatablk, $Rdat, $Rblock               # get next block of data
{ selb          $(Rsbox8F + NR), $(Rsbox8B + NR), $(RsboxCF + NR), $(Rbit6 + NR)
\       lqx     $(Rdatablk + NR), $(Rdat + NR), $Rblock
{ selb          $(Rsbox8F + 2*NR), $(Rsbox8B + 2*NR), $(RsboxCF + 2*NR), $(Rbit6 + 2*NR)
\       lqx     $(Rdatablk + 2*NR), $(Rdat + 2*NR), $Rblock
{ selb          $(Rsbox8F + 3*NR), $(Rsbox8B + 3*NR), $(RsboxCF + 3*NR), $(Rbit6 + 3*NR)
\       lqx     $(Rdatablk + 3*NR), $(Rdat + 3*NR), $Rblock
  selb          $Rstate, $Rsbox07, $Rsbox8F, $Rbit7     # finish table lookup
  selb          $(Rstate + NR), $(Rsbox07 + NR), $(Rsbox8F + NR), $(Rbit7 + NR)
  selb          $(Rstate + 2*NR), $(Rsbox07 + 2*NR), $(Rsbox8F + 2*NR), $(Rbit7 + 2*NR)
        .align  3
{ selb          $(Rstate + 3*NR), $(Rsbox07 + 3*NR), $(Rsbox8F + 3*NR), $(Rbit7 + 3*NR)
\       shlqbyi $Rround, $Rnrounds, 0                   # initialize round counter
# SIMD version of shift rows
{ xor           $Rdatablk, $Rdatablk, $Rroundkey        # add RK to data
\       shufb   $Rstate, $Rstate, $Rstate, $Rshiftrows  # move bytes around
{ xor           $(Rdatablk + NR), $(Rdatablk + NR), $Rroundkey
\       shufb   $(Rstate + NR), $(Rstate + NR), $(Rstate + NR), $Rshiftrows
{ xor           $(Rdatablk + 2*NR), $(Rdatablk + 2*NR), $Rroundkey
\       shufb   $(Rstate + 2*NR), $(Rstate + 2*NR), $(Rstate + 2*NR), $Rshiftrows
{ xor           $(Rdatablk + 3*NR), $(Rdatablk + 3*NR), $Rroundkey
\       shufb   $(Rstate + 3*NR), $(Rstate + 3*NR), $(Rstate + 3*NR), $Rshiftrows
# SIMD version of Add Round Key
{ xor           $Rdatablk, $Rdatablk, $Rstate           # now encrypted data
\       shlqbyi $Rblockout, $Rblock, 0                  # copy block counter
  xor           $(Rdatablk + NR), $(Rdatablk + NR), $(Rstate + NR)
  xor           $(Rdatablk + 2*NR), $(Rdatablk + 2*NR), $(Rstate + 2*NR)
  xor           $(Rdatablk + 3*NR), $(Rdatablk + 3*NR), $(Rstate + 3*NR)
# use similar count-up with block counter
        .align  3
{ ai            $Rblock, $Rblock, 0x40                  # next block
\       stqx    $Rdatablk, $Rdat, $Rblockout            # overwrite block of data
```

```
{ xor           $(Rstate + NR), $(Rctr + NR), $Rroundkey0
{       stqx    $(Rdatablk + NR), $(Rdat + NR), $Rblockout
{ xor           $(Rstate + 2*NR), $(Rctr + 2*NR), $Rroundkey0
{       stqx    $(Rdatablk + 2*NR), $(Rdat + 2*NR), $Rblockout
{ xor           $(Rstate + 3*NR), $(Rctr + 3*NR), $Rroundkey0
{       stqx    $(Rdatablk + 3*NR), $(Rdat + 3*NR), $Rblockout
{ xor           $Rstate, $Rctr, $Rroundkey0             # add RK0 to CTR for next block
{Lblockloop_end:
{       brnz    $Rblock, Lblockloop                     # branch if not last block
 # be sure to return correct counter for block after last
        rotqmbyi  $Rblkpad, $Rblkpad, -12               # move to rightmost word
  sf            $(Rctr + NR), $Rincr, $Rctr             # back up loop
  ceqi          $2, $Rblkpad, 0
  selb          $Rctr, $(Rctr + NR), $Rctr, $2          # # to pad
{ a             $Rout_ctr, $Rctr, $Rblkpad              # now +1 for last block
{       bi      $lr             # return
        .ident  "DRC"
```

D AES CBC Assembly Code

Here is our optimized version of the CBC code (called CBC2). Since the feedback of this cryptographic mode dictates only one block can be done at a time, the resulting code is somewhat readable. In particular, this code shows our one-block optimized *MixColumns* (with *ShiftRows* and *AddRoundKey*).

There are still some unavoidable data dependency stalls in this code, where an instruction waits to use the output of a previous one. (Of the instructions used: all pipeline 0 instructions last 2 cycles except rotate and shift instructions are 4 cycles; all pipeline 1 instructions last 4 cycles except load and store instructions are 6 cycles and branches take 1 if correctly hinted or not taken.)

The format is as in the examples above: named registers begin `$R` and statement labels begin `L`; pipeline 0 instructions are flush left while pipeline 1 instructions are indented; dual-issued instruction pairs are indicated by braces.

Note: the no-operation instructions (`nop` and `lnop`) are only to keep the instruction address parity aligned with the pipeline, to allow later dual issues; of course, they themselves are dual-issued and do not affect the timing; they could have been replaced by `.align` directives.

```
## AES function, CBC mode, 2008 Dec 14 Sun 16:32:44
##   with NEW improved version of Mix Columns
##      (moved polynomial add to State)
## 5 input parameters:        (NO error checking)
##     pointer to data buffer
##     pointer to Round Key buffer
##     number of data blocks (must be compatible with length of data buffer)
##     number of rounds (must be compatible with length of Round Key buffer)
##     initial value for first data block
## NO output parameters

        .file   "aes_cbc.s"
        .section        mydata,"a",@progbits
        .align  4
Sbox:
        .octa   0x637C777BF26B6FC53001672BFED7AB76
        .octa   0xCA82C97DFA5947F0ADD4A2AF9CA472C0
        .octa   0xB7FD9326363FF7CC34A5E5F171D83115
        .octa   0x04C723C31896059A071280E2EB27B275
        .octa   0x09832C1A1B6E5AA0523BD6B329E32F84
        .octa   0x53D100ED20FCB15B6ACBBE394A4C58CF
        .octa   0xD0EFAAFB434D338545F9027F503C9FA8
        .octa   0x51A3408F929D38F5BCB6DA2110FFF3D2
        .octa   0xCD0C13EC5F974417C4A77E3D645D1973
        .octa   0x60814FDC222A908846EEB814DE5E0BDB
        .octa   0xE0323A0A4906245CC2D3AC629195E479
        .octa   0xE7C8376D8DD54EA96C56F4EA657AAE08
        .octa   0xBA78252E1CA6B4C6E8DD741F4BBD8B8A
        .octa   0x703EB5664803F60E613557B986C11D9E
        .octa   0xE1F8981169D98E949B1E87E9CE5528DF
        .octa   0x8CA1890DBFE6426841992D0FB054BB16
ShiftRows:
        .octa   0x00050A0F04090E03080D02070C01060B      # standard (row 0)
        .octa   0x050A0F00090E03040D02070801060B0C      # row 1 on top
        .octa   0x0A0F00050E0304090207080D060B0C01      # row 2 on top
        .octa   0x0F00050A0304090E07080D020B0C0106      # row 3 on top
BranchHints:                                            # for dynamic br. hints
        .fill   16*4, 4, 0
.text
```

```
.global aes_cbc
.type   aes_cbc, @function

##REGISTER DEFINITIONS##
.set    Rin_dat,         3      # 1st param = ptr to block
.set    Rin_key,         4      # 2nd param = ptr to keys
.set    Rin_nb,          5      # 3rd param = number of blocks
.set    Rin_nr,          6      # 4th param = number of rounds
.set    Rin_iv,          7      # 5th param = counter initial value
.set    Rround,         10      # Round counter
.set    Rroundkey,      11      #
.set    Riv,            12      # IV = Initial Value
.set    Rstate,         13      # block State
.set    Ridx,           14      #
.set    Rblock,         15      # block counter
.set    Rbit5,          16      #
.set    Rbit6,          17      #
.set    Rbit7,          18      #
.set    Rsbox01,        19      #
.set    Rsbox23,        20      #
.set    Rsbox45,        21      #
.set    Rsbox67,        22      #
.set    Rsbox89,        23      #
.set    RsboxAB,        24      #
.set    RsboxCD,        25      #
.set    RsboxEF,        26      #
.set    Rsbox03,        27      #
.set    Rsbox47,        28      #
.set    Rsbox8B,        29      #
.set    RsboxCF,        30      #
.set    Rsbox07,        31      #
.set    Rsbox8F,        32      #
.set    Rshiftrows,     33      #
.set    Rshiftrow1,     34      #
.set    Rshiftrow2,     35      #
.set    Rshiftrow3,     36      #
.set    Rrow0,          37      #
.set    Rrow1,          38      #
.set    Rrow2,          39      #
.set    Rrow3,          40      #
.set    Rrow01,         41      #
.set    Rtimes2,        42      #
.set    Rtimes2m,       43      #
.set    Rblockout,      44      # block counter copy
.set    Rnextdat,       45      # block counter copy
.set    Rhint,          46      # branch hint
.set    Rhints,         47      # branch hint table
.set    Rroundkey0,     57      #
.set    Rdatablk,       58      #
.set    Rnrounds,       59      # # of Rounds
.set    Rdat,           61      # 1st param = ptr to block
.set    Rroundkeys,     62      # Keys Ptr (const)
.set    Rsbox0,         64      # S-box Table (const)
.set    Rsbox1,         65      # S-box Table (const)
```

```
        .set    Rsbox2,         66      # S-box Table (const)
        .set    Rsbox3,         67      # S-box Table (const)
        .set    Rsbox4,         68      # S-box Table (const)
        .set    Rsbox5,         69      # S-box Table (const)
        .set    Rsbox6,         70      # S-box Table (const)
        .set    Rsbox7,         71      # S-box Table (const)
        .set    Rsbox8,         72      # S-box Table (const)
        .set    Rsbox9,         73      # S-box Table (const)
        .set    RsboxA,         74      # S-box Table (const)
        .set    RsboxB,         75      # S-box Table (const)
        .set    RsboxC,         76      # S-box Table (const)
        .set    RsboxD,         77      # S-box Table (const)
        .set    RsboxE,         78      # S-box Table (const)
        .set    RsboxF,         79      # S-box Table (const)

        .align  3
aes_cbc:
# setup so round reg counts up to zero from neg.
# then adjust pointer to roundkeys so sum points to round key
# use similar count-up with block counter
# load tables into registers and do Round #0 for first block
{ shli          $Rnrounds, $Rin_nr, 4   # #rounds*16
  lqr     $Rsbox7, Sbox+0x70
{ shli          $Rblock, $Rin_nb, 4     # #blocks*16
  lqr     $Rsbox0, Sbox+0x00
{ ori           $Rdat, $Rin_dat, 0                      # move data pointer
  lqr     $Rsbox1, Sbox+0x10
{ ori           $Rstate, $Rin_iv, 0                     # move IV to State
  lqr     $Rsbox2, Sbox+0x20
{ sfi           $Rnrounds, $Rnrounds, 0x10              # neg. of (#rounds-1)*16 to addr QW
  lqr     $Rsbox3, Sbox+0x30
{ sfi           $Rblock, $Rblock, 0                     # neg. of (#blocks)*16 to addr QW
  lqr     $Rsbox4, Sbox+0x40
{ sf            $Rroundkeys, $Rnrounds, $Rin_key        # offset: roundkeys+round -> round key
  lqr     $Rsbox5, Sbox+0x50
{ sf            $Rdat, $Rblock, $Rdat                   # offset: dataptr+block -> data
  lqr     $Rsbox6, Sbox+0x60
{ ori           $Rround, $Rnrounds, 0                   # initialize round counter
  lqx     $Rroundkey0, $Rroundkeys, $Rnrounds     # get round key #0
{ ai            $Rnextdat, $Rdat, 0x10                  # data ptr for next round (*16)
  lqx     $Rdatablk, $Rdat, $Rblock               # get first block of data
{ ila           $Rhints, BranchHints
  lqr     $Rsbox8, Sbox+0x80
{ ila           $Ridx, Lroundloop_end + 4               # address not to loop
  lqr     $Rsbox9, Sbox+0x90
{ sf            $Rhints, $Rnrounds, $Rhints             # offset: hints+round -> round hint
  lqr     $RsboxA, Sbox+0xA0
{ ila           $Rhint, Lroundloop
  lqr     $RsboxB, Sbox+0xB0
        lqr     $RsboxC, Sbox+0xC0
        lqr     $Rshiftrow1, ShiftRows+0x10
{ xor           $Rstate, $Rstate, $Rdatablk             # add data to current state
  lqr     $RsboxD, Sbox+0xD0
        lqr     $RsboxE, Sbox+0xE0
```

```
        lqr     $RsboxF, Sbox+0xF0
{ xor           $Rstate, $Rstate, $Rroundkey0           # add round key 0 to state
\       lqr     $Rshiftrows, ShiftRows
        lqr     $Rshiftrow2, ShiftRows+0x20
        lqr     $Rshiftrow3, ShiftRows+0x30
Lhinttabloop:
        stqx    $Rhint, $Rhints, $Rround                # put hint for each round
  ai            $Rround, $Rround, 0x10                  # next round (*16)
        brnz    $Rround, Lhinttabloop                   # branch if not last round
        stqx    $Rhint, $Rhints, $Rround                # put hint for next round loop
        stqd    $Ridx, -16($Rhints)                     # store hint not to loop
  ori           $Rround, $Rnrounds, 0                   # initialize round counter
        .align  3
Lroundloop:                                             # also top of Block Loop
# SIMD version of S-box
{ andbi         $Ridx, $Rstate, 0x1F                    # lower 5 bits for partial lookup
\       lnop
{ ai            $Rround, $Rround, 0x10                  # next round (*16)
\       hbr     Lroundloop_end, $Rhint                  # hint for round loop
{ andbi         $Rbit5, $Rstate, 0x20                   # get next bit (#5)
\       shufb   $Rsbox01, $Rsbox0, $Rsbox1, $Ridx       # partial lookup if 3 msb = 000
{ andbi         $Rbit6, $Rstate, 0x40                   # get next bit (#6)
\       shufb   $Rsbox23, $Rsbox2, $Rsbox3, $Ridx       # partial lookup if 3 msb = 001
{ ceqbi         $Rbit5, $Rbit5, 0x20                    # form bytewise selector
\       shufb   $Rsbox45, $Rsbox4, $Rsbox5, $Ridx       # partial lookup if 3 msb = 010
{ ceqbi         $Rbit6, $Rbit6, 0x40                    # form bytewise selector
\       shufb   $Rsbox67, $Rsbox6, $Rsbox7, $Ridx       # partial lookup if 3 msb = 011
{ clgtbi        $Rbit7, $Rstate, 0x7F                   # form selector based on msb (#7)
\       shufb   $Rsbox89, $Rsbox8, $Rsbox9, $Ridx       # partial lookup if 3 msb = 100
{ selb          $Rsbox03, $Rsbox01, $Rsbox23, $Rbit5    # partial lookup if 2 msb = 00
\       shufb   $RsboxAB, $RsboxA, $RsboxB, $Ridx       # partial lookup if 3 msb = 101
{ nop
\       shufb   $RsboxCD, $RsboxC, $RsboxD, $Ridx       # partial lookup if 3 msb = 110
{ selb          $Rsbox47, $Rsbox45, $Rsbox67, $Rbit5    # partial lookup if 2 msb = 01
\       shufb   $RsboxEF, $RsboxE, $RsboxF, $Ridx       # partial lookup if 3 msb = 111
{ selb          $Rsbox8B, $Rsbox89, $RsboxAB, $Rbit5    # partial lookup if 2 msb = 10
\       lqx     $Rroundkey, $Rroundkeys, $Rround        # get round key
{ selb          $RsboxCF, $RsboxCD, $RsboxEF, $Rbit5    # partial lookup if 2 msb = 11
\       lqx     $Rhint, $Rhints, $Rround                # get hint for next round
  selb          $Rsbox07, $Rsbox03, $Rsbox47, $Rbit6    # partial lookup if 1 msb = 0
  selb          $Rsbox8F, $Rsbox8B, $RsboxCF, $Rbit6    # partial lookup if 1 msb = 1
  selb          $Rstate, $Rsbox07, $Rsbox8F, $Rbit7     # finish table lookup
# SIMD version of shift rows
        shufb   $Rrow1, $Rstate, $Rstate, $Rshiftrow1   # move bytes: row 1
        shufb   $Rrow0, $Rstate, $Rstate, $Rshiftrows   # move bytes around: row 0
        shufb   $Rrow2, $Rstate, $Rstate, $Rshiftrow2   # move bytes: row 2
        shufb   $Rrow3, $Rstate, $Rstate, $Rshiftrow3   # move bytes: row 3
# SIMD version of Mix Columns and Add Round Key
  xor           $Rstate, $Rrow1, $Rroundkey             # 1 + RK
  xor           $Rrow01, $Rrow0, $Rrow1                 # 0+1
  xor           $Rstate, $Rstate, $Rrow2                # 1+2 + RK
        .align  3
{ clgtbi        $Rbit7, $Rrow01, 0x7F                   # if msb = 1
\       shlqbii $Rtimes2, $Rrow01, 1                    # shift block 1 bit
```

```
  xor           $Rstate, $Rstate, $Rrow3                # 1+2+3 + RK
  xorbi         $Rtimes2m, $Rstate, 0x1B                # mod field polynomial
  andbi         $Rtimes2, $Rtimes2, 0xFE                # clear lsb
  selb          $Rstate, $Rstate, $Rtimes2m, $Rbit7     # now 1+2+3+RK mod poly
        .align  3               # not really nec. here
  xor           $Rstate, $Rstate, $Rtimes2              # 2*(0+1) + 1+2+3 + RK, done
Lroundloop_end:
        brnz    $Rround, Lroundloop                     # branch if not last round
# LAST ROUND
# SIMD version of S-box
        .align  3
  andbi         $Ridx, $Rstate, 0x1F                    # lower 5 bits for partial lookup
        hbrr    Lblockloop_end, Lroundloop              # hint for block loop
  ori           $Rblockout, $Rblock, 0                  # copy block # for output
        lqx     $Rdatablk, $Rnextdat, $Rblock           # get next block of data
  andbi         $Rbit5, $Rstate, 0x20                   # get next bit (#5)
        shufb   $Rsbox01, $Rsbox0, $Rsbox1, $Ridx       # partial lookup if 3 msb = 000
  andbi         $Rbit6, $Rstate, 0x40                   # get next bit (#6)
        shufb   $Rsbox23, $Rsbox2, $Rsbox3, $Ridx       # partial lookup if 3 msb = 001
  ceqbi         $Rbit5, $Rbit5, 0x20                    # form bytewise selector
        shufb   $Rsbox45, $Rsbox4, $Rsbox5, $Ridx       # partial lookup if 3 msb = 010
  ceqbi         $Rbit6, $Rbit6, 0x40                    # form bytewise selector
        shufb   $Rsbox67, $Rsbox6, $Rsbox7, $Ridx       # partial lookup if 3 msb = 011
  clgtbi        $Rbit7, $Rstate, 0x7F                   # form selector based on msb (#7)
        shufb   $Rsbox89, $Rsbox8, $Rsbox9, $Ridx       # partial lookup if 3 msb = 100
  selb          $Rsbox03, $Rsbox01, $Rsbox23, $Rbit5    # partial lookup if 2 msb = 00
        shufb   $RsboxAB, $RsboxA, $RsboxB, $Ridx       # partial lookup if 3 msb = 101
  ai            $Rblock, $Rblock, 0x10                  # next block
        shufb   $RsboxCD, $RsboxC, $RsboxD, $Ridx       # partial lookup if 3 msb = 110
  selb          $Rsbox47, $Rsbox45, $Rsbox67, $Rbit5    # partial lookup if 2 msb = 01
        shufb   $RsboxEF, $RsboxE, $RsboxF, $Ridx       # partial lookup if 3 msb = 111
  selb          $Rsbox8B, $Rsbox89, $RsboxAB, $Rbit5    # partial lookup if 2 msb = 10
        lqd     $Rroundkey, 0x10($Rroundkeys)           # get round key
  selb          $RsboxCF, $RsboxCD, $RsboxEF, $Rbit5    # partial lookup if 2 msb = 11
        shlqbyi $Rround, $Rnrounds, 0                   # initialize round counter
  selb          $Rsbox07, $Rsbox03, $Rsbox47, $Rbit6    # partial lookup if 1 msb = 0
  selb          $Rsbox8F, $Rsbox8B, $RsboxCF, $Rbit6    # partial lookup if 1 msb = 1
  selb          $Rstate, $Rsbox07, $Rsbox8F, $Rbit7     # finish table lookup
        .align  3
# SIMD version of shift rows
  xor           $Rdatablk, $Rdatablk, $Rroundkey0       # add round key 0 to next data
        shufb   $Rstate, $Rstate, $Rstate, $Rshiftrows  # move bytes around
# SIMD version of Add Round Key
  xor           $Rstate, $Rstate, $Rroundkey            # add round key to state
# use similar count-up with block counter
        stqx    $Rstate, $Rdat, $Rblockout              # overwrite block of data
  xor           $Rstate, $Rstate, $Rdatablk             # add data+RK0 to current state
Lblockloop_end:
        brnz    $Rblock, Lroundloop                     # branch if not last block
        bi      $lr             # return
        .ident  "DRC"
```

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. George Dinolt
   Naval Postgraduate School
   Monterey, California

4. David Canright, Code MA/Ca (5)
   Naval Postgraduate School
   Monterey, California

5. Simson Garfinkel
   Naval Postgraduate School
   Monterey, California

6. Bruce Allen
   Naval Postgraduate School
   Monterey, California