AES Finalists on PA-RISC and IA-64: Implementations & Performance

John Worley, Bill Worley, Tom Christian, Christopher Worley[1]
Hewlett Packard Labs
Fort Collins, CO

Overview

The Advanced Encryption Standard selection process has, for the first time, included software execution speed as a relevant criterion for the choice of the next standard. The initial submissions included keying, encryption, and decryption execution times, in clock cycles, for Intel Pentium, Pentium II, and Pentium Pro microprocessors. While Pentium execution speeds are important, by no means do they completely characterize software performance, particularly that of existing RISC microprocessors and the new IA-64 microprocessor family.

In order to enable a more complete characterization of software performance, our group, working from HP Labs, decided in January 1999, to study and publish the performance of likely AES finalists for PA-RISC and IA-64 microprocessors. We initially selected RC6, Rijndael, Serpent, and Twofish. Our preliminary results were informally presented at the 1999 Rome Conference. Following the selection of the five finalists, we included work on MARS. This paper discusses the issues, implementations, and results of our work for each of the five AES finalists.

Details of specific engineering tradeoffs for Itanium and McKinley chips remain proprietary. We therefore are not at liberty to disclose complete source codes and performance details from which such information can be deduced. What we have chosen to present are actual simulation cycle counts for a snapshot of the evolving McKinley design. These are not cycle counts for an actual product. We offer them as well-substantiated, conservative indicators of the performance of the future family of IA-64 processors. Itanium will be somewhat slower; future implementations will be faster. We believe these results do provide a reasonable basis for software performance judgments about the AES finalists. A summary table appears at the end of the paper.

In addition to processor cycle count, we also present PA-RISC and IA-64 code sizes, register usage, and instruction-level parallelism. Finally, we describe the programming approaches we employed for effective use of both architectures. We would be happy to share full details with the finalists' authors under non-disclosure terms.

Methodology

We focused on hand-optimized assembly language implementations of the algorithms for 128-bit keys and 128-bit blocks, using compiled codes as sanity checkers. We agree with Bruce Schneier that AES codes will be implemented in this manner in actual systems; this also leads to the clearest comparisons between instruction set architectures. Codes for this study were optimized for performance, not code size or table size.

For PA-RISC we measured execution speeds on a PA-8500. We timed executions using the PA-RISC 64-bit interval timer, which counts actual clock cycles. To eliminate cache and system effects, we ran tens of millions of executions, varying keys and data blocks on a lightly loaded system, and profiled those runs with minimum cycle counts. We observed that runs often would differ by only a few cycles, and that the cycle counts formed Gaussian distributions. It was further observed that the input value (input key for keying, data block for encryption/decryption) noticeably affected performance for algorithms that used table look-ups. Thus, while the PA-RISC times are best observed times, we also show the distribution's average and maximum values.

Lacking IA-64 hardware, we employed three different types of simulators. Initial debugging used a fairly fast and purely functional instruction set simulator. The second type was considerably slower, but simulated parallel execution, latencies, and memory hierarchy behavior. This was used for additional code validation and preliminary execution cycle counts.

These simulators, while useful, did not guarantee absolute fidelity to the chip designs. Therefore, final timings used fully simulated RTL designs of the Merced (now Itanium) and McKinley chips. This approach was extremely slow, and our results often varied from day to day, as engineers improved their designs. We constructed special tools that automatically prepared test inputs and displayed the cycle-by-cycle behavior of the microprocessor pipeline. The memory hierarchy was initialized for each run, and the timing could be computed by subtracting cycle numbers from the pipeline output.

Notation

`A <<< n`	Left rotation by `n` bits
`A >>> n`	Right rotation by `n` bits
`A ⊕ B`	Bit-wise Exclusive-OR
`A +.× B`	Matrix multiplication
`[b0, b1, ..., bn]`	Column vector, LSB first

PA-RISC Facts

PA-RISC first shipped in 1986 and is the processor for Hewlett-Packard's RISC workstation and server products. Architecture features include 64-bit virtual addressing, 32 general-purpose registers, and 32 floating point registers. Current processors implement the 64-bit Version 2.0 of the PA-RISC architecture.

[1] John S. Worley **jworley@fc.hp.com** William S. Worley, Jr. **worley@hpl.hp.com**
Tom W. Christian **twc@fc.hp.com** Christopher S. Worley **cworley@fc.hp.com**

This study utilized the PA-8200 and PA-8500 microprocessor chips. Both of these chips are out-of-order superscalar designs, capable of executing two memory operations and two integer or floating point instructions per cycle. Only one store instruction can complete per cycle. Careful software scheduling is required to realize the full parallelism.

IA-64 Overview

This section provides a *very* brief overview of the IA-64, highlighting features in the discussions that follow. Readers familiar with the architecture can skip this section.

Parallelism and Functional Units

The majority of processor architectures specify sequential instruction execution. Microarchitectures then employ superscalar logic to issue multiple instructions in parallel whenever possible. In contrast, the IA-64 architecture puts all the parallelism cards on the table. There are four types of functional units: **M** (memory), **I** (integer), **F** (floating point), and **B** (branch); each IA-64 implementation has two or more of each of these units. IA-64 hardware detects when program parallelism exceeds the capabilities of the implementation, but responsibility for organizing instructions to execute in parallel is wholly with the programmer or compiler.

Instructions, Bundles, and Issue Groups

There is a corresponding instruction class for each functional unit type, although a specific instruction may not be able to execute on all units of that type in a given implementation. In addition, there is an **A** (ALU) instruction class that can execute on both **I** and **M** units. **A** instructions include most integer arithmetic and logical operations, so that otherwise idle memory units can be used for parallel computation.

Three instructions are grouped into a *bundle*, where all instructions in the bundle may be eligible to be issued in parallel to functional units specified by the bundle type. Sequential bundles that can issue in parallel form an *issue group*. One characteristic of an IA-64 implementation is the maximum number of bundles that can issue together. For example, a processor that can issue at most two bundles in one cycle is referred to as a "two-banger."[2]

Registers and the Register Stack

IA-64 provides 128 64-bit integer registers. The low 32 registers (`r0` - `r31`) are common for all code. For function arguments and local values, each procedure can allocate up to 96 additional registers in a *register stack frame*. Saving and restoring registers in the register stack is handled by an independent hardware thread, so that no registers need to be saved and restored explicitly.

In addition to the integer registers, IA-64 provides 128 extended precision (64-bit mantissa, 17-bit exponent) floating point registers, 64 1-bit predicate registers (see below), and eight branch registers for indirect branches.

Predication

A powerful feature of IA-64 is *instruction predication*. Every instruction, except for certain branch and control instructions, is predicated, i.e., its execution is enabled or disabled by one of the 64 predicate bits. One predicate, `p0`, is hardwired to '1' for instructions that execute unconditionally or cannot be predicated. Predicates are set or cleared by compare instructions and certain floating-point instructions. Also, the 64 predicates can be read or set in parallel using special instructions. Predication allows, for example, one of two instructions to execute based on a comparison condition, or for instructions to be enabled during the first pass of a loop and disabled for all subsequent iterations.

Counted Loops

IA-64 provides hardware support for counted loops. The special registers `ar.lc` (loop counter) and `ar.ec` (epilogue counter) control when the branch instructions `br.ctop` and `br.cexit` are taken. For example, if `ar.lc` is set to 9 and `ar.ec` is set to 0, a counted loop will execute 10 times if the loop ends with `br.ctop`, 9 times if the loop begins with `br.cexit`. The hardware is designed to predict perfectly when a branch will be taken or fall through, so that counted loops can execute with no branch penalties.

Rotating Registers

When a subroutine allocates a register stack frame, some or all of the local registers, starting from `r32`, can be set to *rotate*. Each time a counted loop branch is taken, the rotating registers are circularly renamed such that the next iteration of the loop can operate on different data without changing the register name. For example, if there are eight registers designated as rotating, the renaming is as follows:

`r32` → `r33` → `r34` → `r35` → `r36` → `r37` → `r38` → `r39` → `r32`

Fixed portions of the floating point and predicate registers also rotate. The high 96 floating point registers (`f32` through `f127`) rotate. The high 48 predicate registers (`p16` to `p63`) also rotate, but with a slight difference. While the loop counter `ar.lc` is non-zero, a '1' value is shifted into `p16`; if `ar.lc` is zero and the epilogue counter `ar.ec` > 1, a '0' value is shifted in instead.

Programming Issues

There are three operations commonly used in cryptographic algorithms that are not fully realized in the integer hardware on PA-RISC and IA-64: fixed 32-bit rotations, variable 32-bit rotations, and 32x32→32 unsigned integer multiplies.

[2] This term comes from the slang term for a two-cylinder engine. While three-banger or more implementations are foreseeable, it seems unlikely that IA-64 will ever give rise to, say, a V12.

PA-RISC

On PA-RISC, fixed rotations can be executed in one cycle using the shift right pair word (`shrpw`) instruction. This instruction concatenates the low 31 and 32 bits from left and right source registers, respectively, shifts right the specified distance, and leaves the high 32 bits undefined. If the two source registers are the same, the low bits are concatenated with the high bits, exactly as would occur in a rotation. Thus, fixed rotations on PA-RISC can be defined as follows:

```
ROTR   .macro       src, dst, count
       shrpw        src, src, count, dst
       .endm
ROTL   .macro       src, dst, count
       shrpw        src, src, 32 - count, dst
       .endm
```

Variable rotations use the same strategy, except that an extra cycle is required to move the shift distance into the SAR (shift amount register). For a right rotation, the actual shift distance is used. For a left rotation, the 5-bit complement of the distance is used and the value is pair-shifted right one before the variable shift. The left shift also executes in two cycles since the `mtsarcm` (move to SAR complement) and the first `shrpw` can issue in the same cycle on the PA-8000 family.

Integer multiplication on PA-RISC requires using the unsigned integer multiply in the floating point unit. Since the only path for moving data between the integer and floating point units is memory, the multiplicands must be stored, loaded into the FPU, multiplied, stored again, and reloaded into the integer unit. This adds latencies on both sides of the multiply, in addition to the multiply time itself.

IA-64

Although the IA-64 architecture has a shift right register pair instruction, it only operates on full 64-bit registers. This can still be used to implement 32-bit fixed rotations in two cycles as follows:

```
dep.z  TMP = src, 32, 32
shrp   dst = src, TMP, count + 32
```

for right rotations, and

```
dep.z  TMP = src, 32, 32
shrp   dst = src, TMP, 64 - count
```

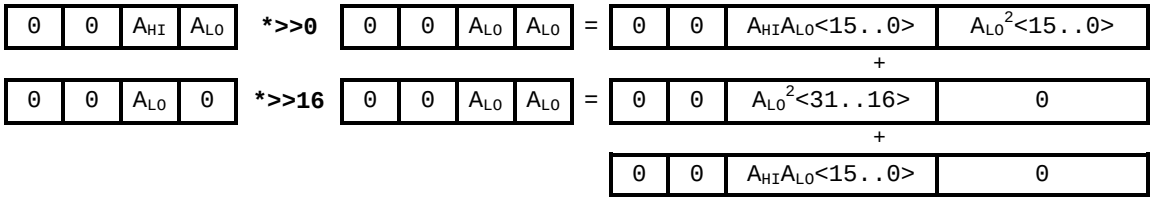
for left rotations.

The `dep.z` instruction puts the low 32 bits of the source register in the high half of a temporary register, clearing the low half. The pair-shift concatenates the low bits with the high bits and shifts far enough to put the proper set of bits in the low half of the destination. Like the PA-RISC instruction, the destination's high half is not cleared. None of the AES finalists require these bits to be cleared; however, the `zxt4` instruction can be used if necessary.

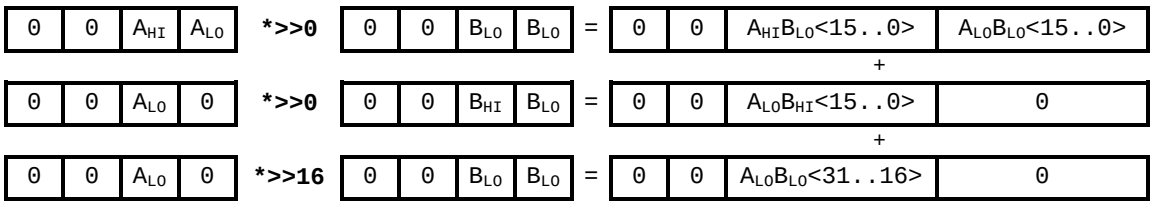
On IA-64, variable rotates are implemented much as in the C language: shift left `j`, shift right `(32 - j)`, `OR` or `ADD` the results together. This involves four operations and a minimum of three cycles. The variable shifts are executed on the multimedia units (MMUs).

Like PA-RISC, the IA-64 primary integer multiply is implemented on the floating point unit and involves latency cycles to move back and forth. However, 16x16 MMU multiplies and parallel adds can be used to compute and sum the partial products instead. This is effective when only the low 32 bits of the result are of interest. In particular, the parallel 16-bit unsigned multiply and shift instruction (`pmpyshr.u`) can be used to complete a 32x32→32 multiply.

If we consider multiplicands derived from `A` as four 16-bit elements, A^2 can be computed with two multiplies and two adds as follows:



One of the operands is just the argument, `A`. The other two arguments are generated by the 16-bit `mux` MMU instructions; the additional addend is derived from the first product using the 16-bit `mix` instruction. The general 32x32→32 requires three multiplies and two additions. If we consider multiplicands derived from `A` and `B` as four 16-bit elements, the operations are:



Two of the operands are just the arguments, `A` and `B`. The other two arguments are generated by the 16-bit `mix` and `mux` MMU instructions.

It has been noted that with better hardware support for 32-bit rotations and 32x32→32 multiplication, all the AES finalists will outperform Pentium on IA-64. In the performance analysis for each algorithm, we have estimated performance for a hypothetical IA-64 implementation, called IA-64++, with the following enhancements:

- A single-cycle shift right pair word instruction, as in PA-RISC
- Single-cycle, 32-bit, left and right variable rotate instructions
- A two-cycle 32x32→32 unsigned multiply

Mars

The Mars encryption scheme (IBM team) uses a mix of approaches: substitution boxes, Feistel networks, multiplication, and fixed and variable rotates. The single substitution box, `S[]`, is fixed, and is employed both as a 512 word array (9-bit index), and as low (`S0[]`) and high (`S1[]`) 256 word arrays (8-bit index). The principal challenges for PA-RISC and IA-64 implementations are the 32x32→32 multiply and variable rotates.

Keying[3]

Mars keying initializes the first `N` elements of a fifteen-element array, `T[]`, to the input key `k[]`, where N is the size of the key in 32-bit words. The key is then padded to 15 words by setting `T[N]`←`N` and zeroing the remainder of the array. Instead of generating the entire expanded key directly, Mars generates ¼ of the array, or 10 words, each time, repeating the process four times to develop the entire key array, `K[]`. There are three steps in each iteration: linear transform, stirring, and storing. The linear transform applies the formula:

```
T[i] = T[i] ⊕ ((T[i-7 mod 15] ⊕ T[i-2 mod 15]) <<< 3) ⊕ (4i + R)
```

to each element of the array, where `R` is the iteration count (0..3). Stirring uses the following formula:

```
T[i] = (T[i] + S[T[i-1 mod 15] & 0x1ff]]) <<< 9
```

applied to each word, repeated four times. Finally, 10 words from the intermediate array are stored in the expanded key array as follows:

```
K[10×R + i] = T[4i mod 15]
```

which effectively stores words 0, 4, 8, 12, 1, 5, 9, 13, 2, and 6, in that order, from the temporary array. After all the expanded key words are generated, those used in multiplication (`K[5]`, `K[7]`, …, `K[35]`) are modified if they are weak, i.e., contain long runs of 1's or 0's. The algorithm for identifying weak key words comes from the Mars implementation by Brian Gladman.

PA-RISC

The PA-RISC implementation keeps `T[]` in registers. The linear transform, the inner stirring loop, and key stores are straight-lined. In the fix-up phase, the two-ALU PA-RISC has sufficient execution bandwidth to compute the fix-up mask in parallel with looking for long runs of 1's or 0's. If there are no such runs, the remainder of the fix-up is skipped. Using the authors' estimates that statistically 1 out of 41 keys are weak, the extra computation is skipped 97.6% of the time, a performance win even with a branch penalty.

IA-64

The IA-64 Mars keying implementation uses software pipelining to increase keying speed. The routine allocates a 16-register stack frame, all of which are rotating. The register usage is as follows (indices are modulo 15):

r32	r33	r34	r35	r36	r37	r38	r39	r40	r41	r42	r43	r44	r45	r45	r47
T_{i-1}	T_{i-2}	T_{i-3}	T_{i-4}	T_{i-5}	T_{i-6}	T_{i-7}	T_{i-8}	T_{i-9}	T_{i-10}	T_{i-11}	T_{i-12}	T_{i-13}	T_{i-14}	T_i	T_x

By assigning $T_x \leftarrow T_i$ at the end of the loop, this organization implements a 15-register rotation. The linear transform XORs T_{i-2} (`r33`) and T_{i-7} (`r38`), rotates the result, the XORs with T_i and the iteration constant (`4i + R`). This would normally require four cycles; however, the transform can be reorganized into a two-stage, two-cycle pipeline. The first stage computes $T_{i-2} \oplus T_{i-7}$ and extracts the high three bits of the result; the second phase computes $T_i \oplus$ (`4i + R`) and completes the rotation, then XORs the final result. The loop uses rotating predicates to disable the second phase on the first iteration, while the last execution of the second phase is handled after the loop so that the values return to their initial positions when the loop is complete.

Pipelining the inner stirring loop is limited by the use of `T[i-1 mod 15]` in computing `T[i]`; however, the high-order nine bits extracted for rotation can be used to start the S-Box look-up for the next iteration. This allows a two-stage, four-cycle pipeline, which executes 33% faster than the 6-cycle, non-pipelined equivalent.

Like PA-RISC, the fix-up mask can be computed in parallel with looking for runs of 1's and 0's. Unlike PA-RISC, branches include 'hints', so that the branch penalty is only incurred for weak keys, or 2.4% of the time.

Encryption

Mars encryption consists of four phases, each repeated eight times: forward mix, forward keyed transform, backward keyed transform, and backward mix. The forward and backward mixing uses table look-ups, fixed rotation, XORs, and addition and subtraction in a rotating pattern, e.g., `fmix(A, B, C, D)`, `fmix(B, C, D, A)`, etc. There are asymmetric additions in steps 1, 2, 4, and 5 of the forward mix, with corresponding subtractions in steps 2, 3, 5 and 6 of the backward mix.

[3] This is the 'tweaked' version of the Mars keying. The implementation of the initialization, mixing, and stirring phases of the original scheme is discussed in Appendix B. The key fix-up is identical for both schemes.

The core of the keyed transforms is the *E* function, which takes one data word and uses table look-up, multiplication, variable rotation, additions and XORs to generate three data words (`L`, `M` and `R`) to add or XOR with the other three data words as follows:

Forward Mode	Backward Mode
`D[1] += L`	`D[1] ^= R`
`D[2] += M`	`D[2] += M`
`D[3] ^= R`	`D[3] += L`

On PA-RISC, the mixing phases are coded as straight-line operations. Even with the four table look-ups per step, there is enough memory bandwidth to load the 16 multiplicative keys into the floating-point unit at the same time. The real bottleneck is the integer multiply in the *E* function: the data word must be rotated, stored, loaded into the floating point unit, multiplied, stored again and reloaded into an integer register. Although an addition and table look-up can be evaluated in parallel, these do not fully amortize the performance cost of the multiply.

On IA-64, both forward and backward mixing can be coded as a single loop: the asymmetric operations are controlled by loading a specific bit pattern in the rotating predicates, enabling the appropriate operation at the proper step. Because of perfect branch prediction with counted loops, this approach executes in the same cycle count as straight-line code.

On IA-64, MMU multiplies are used to compute the *E* function multiplication. Once the multiplication is complete, the remainder of the *E* function can be evaluated. Like the mixing phases, a predetermined bit pattern loaded in the rotating predicates controls whether the forward or backward mode operations are enabled at each step.

Performance

Cycles	Pentium	PA-RISC			IA-64	IA-64++
		Min	Average	Max		
Keying	2128	1797	1804.65	1879	1408	1408
Keying (Original)	3894	1969	1975.89	2060	1903	1313
Encryption	320	540	563.01	584	511	255
Decryption	374	538	552.37	566	527	271

On PA-RISC, Mars keying executes in 1797 cycles, compared to the best-reported Pentium results of 2128, a 15.6% performance advantage. Encryption and decryption, however, run slower due to the multiplication overhead: 68.8% slower for encryption (540 vs. 320) and 43.9% slower for decryption (538 vs. 374).

On IA-64, keying completes in 1408 cycles, a 33.8% performance gain. Encryption and decryption, with the extra cycles required for multiplication and variable rotation, are slower than Pentium: 59.7% slower for encryption (511 vs. 320) and 40.9% slower for decryption (527 vs. 374). Keying on IA-64++ is the same 1408 because the software pipelines hide the extra cycles needed for rotation. Encryption improves to 255 cycles (20.3% faster than Pentium), and decryption also improves to 271 cycles (27.5% faster).

RC6

The principal programming challenge when implementing RC6 (Rivest, Robshaw, Sidney, Yin) on PA-RISC and IA-64 is the lack of the fast 32x32→32 multiply and variable rotate primitive the algorithm requires for performance. On the positive side, IA-64's rotating integer registers and instruction predication simplify data management and allow for a very compact code size.

Keying

RC6 keying starts with the input key, `L[]`.The key array, `S[]`, is initialized using the two magic numbers P_{32} = 0xB7E15163 and Q_{32} = 0x9E3779B9, as follows:

```
S[0] = P32
S[1] = P32 + Q32
S[2] = P32 + 2 * Q32
S[3] = P32 + 3 * Q32
       . . .
```

The keying algorithm then performs three mixing passes over the two arrays:

```
A = S[i] = (S[i] + A + B) <<< 3
B = L[j] = (L[j] + A + B) <<< (A + B)
```

where A and B are initially zero, and i and j count circularly through the key and input key arrays, respectively. If the first pass through the key array is handled separately, it is possible to combine the key array initialization with the first mixing phase. The first mix can also be partially hard coded, since A = B = 0, and S[0] = P_{32}. Since, after the first loop pass, B is just the previous, modified input key word, the variable B is replaced with LPREV(k), the user input key L[(k-1) mod 4]. The first pass is coded as follows:

```
keyVal = P32;
A = T = ROTL(P32, 3);
for (k = 1; k < NKEYS; ++k) {
       LPREV(k) = ROTL(LPREV(k) + T, T);
       keyVal += Q32;
       S[k - 1] = A;
       A = ROTL(keyVal + A + LPREV(k), 3);
       T = LPREV(k) + A;
}
S[NKEYS - 1] = A;
```

This organization saves one full load and store of the key array and does not require computing the modulus 2*r + 4, where r is the number of encryption rounds. The last two passes are identical, with a similar structure to the first pass, but do not, of course, re-initialize the key array.

For PA-RISC, each instance of the loop can be unrolled four ways, with the input key words reordered circularly each time - this eliminates loading and storing the keys, and the modulus computation on the input key index.

The IA-64 architecture suggests a different strategy for implementation. The large register file allows *the entire key array* to be kept in registers; the rotating integer registers naturally mimic the way data flows through the computation, such that no indexing or modulo operations are required. The keying routine allocates a 56-register stack frame, all of which are rotating. The rotating registers are allocated as follows:

r32-r33	r34	r35	r36	r37	r38	r39	r40	r41	r42-r82	r83	r84-r87
Unused	L_X	L_n	L_{n+1}	L_{n+2}	L_{n+3}	S_X	S_{Active}	S_{Prev}	Key Array	S_{Next}	Unused

where <L_n … L_{n+3}> are initialized from the user input key. In order to circulate the keys and key array separately, $L_X \leftarrow L_{n+3}$ and $S_X \leftarrow S_{Next}$ before the registers are rotated. Each time through the loop, the code operates on L_n, S_{Active}, and S_{Prev}. Rewriting the mixing loop in these terms:

```
for (k = 1; k < NKEYS; ++k) {
       Ln = ROTL(Ln + T, T);
       A = ROTL(SActive + SPrev + Ln, 3);
       T = Ln + A;
       SActive = A;
       LX = Ln+3;
       SX = SNext;
}
```

Predicated instructions enable key array initialization during the first mixing pass and storing the final key words during the final pass, all within the same code loop and *without* branching. There are enough unused instruction slots to compute the two qualifying predicates with no additional cycles. The keying routine is thus coded in a single loop:

```
for (k = 1; k < 3 * NKEYS; ++k) {
       Ln = ROTL(Ln + T, T);       if (firstMix) SActive = SPrev + Q32;
       firstMix = k < NKEYS-1;     lastMix = k >= 2 * NKEYS;
       SPrev = A;                  if (lastMix) *S++ = A;
       A = ROTL(SActive + A + Ln, 3);
       T = Ln + A;
       LX = Ln+3;
       SX = SNext;
}
*S = A;
```

This coding is extremely compact: the entire routine consists of 39 instructions in 16 IA-64 bundles; the core loop is 20 instructions.

Encryption

The RC6 definition is compact and elegant, but the algorithm relies on a fast 32x32→32 multiply and variable rotate for performance. To multiply on PA-RISC, the two data words must be stored, loaded into the floating point unit, multiplied, stored again and reloaded into integer registers. The inner loop is unrolled to rotate the data words.

On IA-64, MMU multiplies are used to compute A^2. Once the full multiplication is complete, the `shladd` instruction computes the final product $2A^2 + A \equiv A*(2A + 1)$. Using rotating registers for the data words, RC6 encryption can be coded in a single loop.

Performance

Cycles	Pentium	PA-RISC			IA-64	IA-64++
		Min	Average	Max		
Keying	1632	1077	1077	1077	1581	1057
Encryption	243	580	590.76	597	490	150
Decryption	226	493	496.37	499	490	130

On PA-RISC, RC6 keying executes in 1077 cycles, compared to the best-reported Pentium results of 1632, a 34% performance advantage. Encryption and decryption, however, run slower due to the multiplication overhead: 138% slower for encryption (580 vs. 243) and 118% slower for decryption (493 vs. 226).

On IA-64, keying completes in 1581 cycles, a 3.1% performance gain. Encryption and decryption, with the extra cycles required for multiplication and variable rotation, are slower than Pentium: 101.7% slower for encryption (490 vs. 243) and 116.8% slower for decryption (490 vs. 226). For IA-64++, keying is estimated to run in 1057 cycles, 54% faster than Pentium, encryption in 150 cycles (38.3% faster), and decryption in 130 cycles (42.5% faster)

Rijndael

The principles for a fast Rijndael (Daemen, Rijmen) implementation are largely explained in the algorithm specification. A short comment in section 5.2.2 summarizes the general approach:

> *"In the table-lookup implementation, all table lookups can in principle be done in parallel. The EXORs can be done in parallel for the most part also."*

This turns out to be an understatement. In other AES candidates, parallelism must be squeezed from the specification, while Rijndael's parallelism cup runneth over. Even the keying phase has considerable parallelism, as will be shown.

Realizing this parallelism requires five 4K tables, as discussed below, although only two tables are used for any one operation. Each 4K table is made up of 4 256x4 byte tables, where each 1K table is rotated one byte position from the previous. The tables and the operations they're used in are:

S-Box	**Keying, Encryption** Implements byte substitution only
I-Box	**Decryption** Implements inverse byte substitution only
Column Mix	**Encryption** Main substitution box - combines the byte substitution and column mix operations
Inverse Mix	**Decryption** Inverse substitution box - combines the byte substitution and inverse column mix operations
Key Mix	**Keying** Column mix box for computing the inverse key table

These tables are all derived from the basic GF(2^8) mathematics outlined in the specification. A simple C program is used to generate all tables and print them as C array declarations to compile and link with the algorithm codes. While 20K bytes of tables may be not optimal for some target implementations, large memory, large cache machines like PA-RISC and IA-64 gain substantial performance with what is negligible extra data. Rijndael outperforms all other AES submissions in keying, encryption, and decryption. In particular, Rijndael keying is a full order of magnitude faster than most other algorithms.

Keying

Rijndael key expansion looks largely serial. There are four look-ups every fourth key word, but little else to suggest parallelism. The discussion in section 5.3.3, however, shows that decryption can be more efficiently implemented if an "inverse" key table is used. If the basic key generation loop is unrolled four times, we can combine the inverse key computation with the key generation:

```
A = SubByte(RotByte(D)) ^ Rcon[i];
B = B ^ A;
C = C ^ B;
D = D ^ C;
IA = InvMixColumn(A);
IB = InvMixColumn(B);
IC = InvMixColumn(C);
ID = InvMixColumn(D);
```

Clearly, the `InvMixColumn` operation, which is four byte-indexed lookups into four 256-entry tables and three XORs, can begin as soon as the key word is ready. Thus, both the forward and inverse key tables can be computed in the same time as computing the inverse table. As a minor space optimization, the last forward key and first inverse key, which are identical, are stored only once in a combined key table.

Both the PA-RISC and IA-64 implementations are straightforward: as soon as the forward key is available, start the look-ups for the inverse key. Two look-ups are performed on the key word `A`, but only one set of byte extractions is needed, saving four operations per

round. PA-RISC has 28 registers available to a subroutine: all of these are needed to hold the intermediate results. The large register file on IA-64 provides enough temporary registers to perform the computation with maximum concurrency. Rijndael keying improves greatly when everything can be kept in registers.

On PA-RISC, a load address can be the sum of a base register and a scaled offset register; thus, table look-up requires two instructions. IA-64, however, only takes a load address from a register without offset. Therefore, a table look-up must explicitly scale the index and add it to the desired table address: this is accomplished with the `shladd` instruction. The sequence of extract, scale and add, load is pipelined, so that the entire look-up sequence only requires one extra cycle over the equivalent PA-RISC sequence. The greater parallelism in IA-64 allows the forward key computation and XOR trees to overlap the look-ups, giving it an overall performance advantage.

Encryption

Rijndael encryption, while defined as several, separate steps, can be collapsed into a single set of table look-ups by (1) computing the look-up tables to combine the byte substitution and column mix operations, and (2) selecting the index bytes from the data block to reflect the row rotation in each round. Decryption is identical except for the look-up table and the order of byte selection. It is not surprising, then, that encryption and decryption are very similar to keying, except that only 16 look-ups are done per round instead of the 20 performed for each keying round.

Performance

Cycles	Pentium	PA-RISC			IA-64	IA-64++
		Min	Average	Max		
Keying	1338	239	249.25	261	148	148
Forward Keying	217	85	92.18	101	104	104
Encryption	284	168	175.5	193	124	124
Decryption	283	168	175.88	192	125	125

On PA-RISC, Rijndael full keying executes in 239 cycles, compared to the best-reported Pentium results of 1338, a 5.6:1 performance advantage. Encryption and decryption are faster: 40.9% faster for encryption (168 vs. 284) and 40.6% faster for decryption (168 vs. 283). On IA-64, keying completes in 148 cycles, a 9:1 performance improvement over Pentium. Encryption and decryption are also faster: 56.3% faster for encryption (124 vs. 284) and 55.8% faster for decryption (125 vs. 283).

The parallelism of Rijndael saturates a two-banger IA-64. To explore the limits of Rijndael's parallelism, a code schedule was developed for a hypothetical, four-banger implementation. With this 12-way parallel IA-64, the inner loop of Rijndael encryption can be executed in 7 cycles, which suggests a total encryption time of 74 cycles per 128-bit data block. This is only one cycle short of the theoretical limit of 6 cycles per round for an arbitrarily wide IA-64 implementation, which would perform 16 extracts, 20 address computations, 20 loads, then three levels of XORs.

Serpent

The heart of the Serpent algorithm (Anderson, Biham, Knudsen) is the set of Boolean equations implementing the "bit-slice" substitution boxes. One set of equations was submitted with the AES proposal; Brian Gladman and Sam Simpson used a recursive expression search program to develop an alternative set of equations that improved performance on the Pentium-II platform. Dr. Gladman, however, cautions on his Serpent web page[4]:

> *"On any particular machine it will be desirable to experiment with the order of terms (where there is quite a lot of flexibility) and with the reuse of the temporary variables used during function evaluation."*

Taking this advice to heart, the two sets of equations, along with an earlier version of Gladman's equations, and a set of equations optimized for Pentium submitted to the authors by Dag Arne Osvik[5], were analyzed according to the following metrics:

Ops Count of Boolean operations required to compute the substitution or reverse substitution function. The equation parser looks for occurrences of `A & ~B` to take advantage of the `and-complement` instruction in both the PA-RISC and IA-64 instruction sets.

Cycles Number of steps required to complete the computation on a highly parallel machine, such as IA-64, and a two-ALU operation superscalar machine, such as PA-RISC.

Width For IA-64, the largest number of operations executed concurrently.

Temps Number of temporary values. In order to reduce the number of temporaries, a simple register analysis was performed that first re-used the output terms as intermediate results, then assigned temporaries as needed by the computation.

The results of this analysis for IA-64, summarized in Table 1 below, are interesting: even though the Gladman equations consistently have fewer operations than the others, only 4 of the 16 sets compute faster. When the equations are analyzed for two-ALU

[4] The expression search program, Boolean equations and reference implementations are available at **http://www.btinternet/~brian.gladman/cryptography_technology/Serpent**

[5] Dag Arne Osvik **osvik@ii.uib.no**

operation on PA-RISC, the results (Table 2) favor Gladman's equations, but four of Osvik's equations compute faster. A follow-up submission from Mr. Osvik for S-Box 3 resulted in a spectacular, 4-cycle, solution for IA-64, even though it has the highest operation count of any equation.

The conclusion here is that there is no optimal set of bit-slice equations for all Serpent implementations: the capability and constraints of the target machine must be carefully considered. The authors invite others to submit their own equations for analysis, and offer the analysis tools used here to the Serpent team for their own use.

Keying

Serpent keying starts with the input key, padded to 256 bits, and generates 132 4-byte values with the recurrence:

$$W_i = (W_{i-8} \oplus W_{i-5} \oplus W_{i-3} \oplus W_{i-1} \oplus \Phi \oplus i) <<< 11$$

where W_{-8} = input key word 0, W_{-7} = input key word 1, etc., and Φ is `0x9e3779b9`, derived from the Golden ratio. The resulting values, $[W_0 \ldots W_{131}]$, are then processed in groups of four, $<W_n, W_{n+1}, W_{n+2}, W_{n+3}>$, applying the Serpent forward substitution boxes in the order $S_3, S_2, S_1, S_0, S_7, \ldots, S_4, S_3$. This generates the 33 128-bit keys required for encryption.

Inspecting the recurrence, there is an active state of eight words and that W_i replaces W_{i-8} at each step. If we label the initial key words W_{-8} = A, W_{-7} = B, … W_{-1} = H, we can rewrite the recurrence as the following pattern:

```
A′ = (A ⊕ D  ⊕ F  ⊕ H  ⊕ Φ ⊕ 0) <<< 11
B′ = (B ⊕ E  ⊕ G  ⊕ A′ ⊕ Φ ⊕ 1) <<< 11
C′ = (C ⊕ F  ⊕ H  ⊕ B′ ⊕ Φ ⊕ 2) <<< 11
D′ = (D ⊕ G  ⊕ A′ ⊕ C′ ⊕ Φ ⊕ 3) <<< 11
E′ = (E ⊕ H  ⊕ B′ ⊕ D′ ⊕ Φ ⊕ 5) <<< 11
F′ = (F ⊕ A′ ⊕ C′ ⊕ E′ ⊕ Φ ⊕ 6) <<< 11
G′ = (G ⊕ B′ ⊕ D′ ⊕ F′ ⊕ Φ ⊕ 7) <<< 11
H′ = (H ⊕ C′ ⊕ E′ ⊕ G′ ⊕ Φ ⊕ 8) <<< 11
                  . . .
A′ = (A ⊕ D  ⊕ F  ⊕ H  ⊕ Φ ⊕ 128) <<< 11
B′ = (B ⊕ E  ⊕ G  ⊕ A′ ⊕ Φ ⊕ 129) <<< 11
C′ = (C ⊕ F  ⊕ H  ⊕ B′ ⊕ Φ ⊕ 130) <<< 11
D′ = (D ⊕ G  ⊕ A′ ⊕ C′ ⊕ Φ ⊕ 131) <<< 11
```

This formulation has some limited parallelism in the `XOR` trees. Eventually, the equations will serialize on the 11-bit rotation, but the overall sequence can be organized on a parallel machine to minimize the performance effect. Intermediate loads and stores can be eliminated by overlapping the S-box lookup for $<W_n, W_{n+1}, W_{n+2}, W_{n+3}>$ with the computation of $<W_{n+4}, W_{n+5}, W_{n+6}, W_{n+7}>$. Because different S-boxes are used at each step, the highest performance for Serpent keying is realized by a straight-line implementation.

On PA-RISC, limited to two-way integer instruction parallelism, each set of four recurrence computations saturates the processor for 11 cycles (22 operations). The 11-bit rotation is implemented with a single instruction (`shrpw`); common subexpressions (e.g., `F ⊕ H`) remove two of the 24 operations (five XORs and one rotate per step, times four steps). Since PA-RISC does not have an immediate XOR operation, the `(Φ ⊕ i)` term is computed by adding the low 11 bits of the value (constant for each step) to the high 21 bits (constant for all steps); thus, the computation still occurs in one cycle. To avoid errors, the 11-bit values are generated by a simple program.

IA-64 rotation requires two instructions (deposit and shift register pair). This increases the cycle count for computing four steps from 11 on PA-RISC to 14. However, the machine's greater parallelism can be employed to overlap S-Box and recurrence logic as follows:

```
Recurrence(W0, W1, W2, W3)
Recurrence(W4, W5, W6, W7)          Sbox3(W0, W1, W2, W3)
Recurrence(W8, W9, W10, W11)        Sbox2(W4, W5, W6, W7)
Recurrence(W12, W13, W14, W15)      Sbox1(W8, W9, W10, W11)
                  . . .                       . . .
Recurrence(W124, W125, W126, W127)  Sbox5(W120, W121, W122, W123)
Recurrence(W128, W129, W130, W131)  Sbox4(W124, W125, W126, W127)
                                    Sbox3(W128, W129, W130, W131)
```

Each step in this parallel evaluation, including storing the key words, executes in the 14 cycles needed for the recurrence alone, yielding a substantial speed-up for Serpent keying.

Encryption

Serpent encryption and decryption use 32 rounds of key exclusive OR's, substitution box logic and linear transforms. The S-box issues are almost identical to those for keying, as discussed above. The linear transform, which accelerates the avalanche effect, limits the potential for overlap with the S-box computations. Depending on the S-box equations used, at most one or two cycles can be removed per S-box; the current implementation overlaps one cycle for six of the eight S-box equations.

The forward linear transform, diagrammed in Figure 1, consists of 16 operations (six fixed rotations, two rotations, eight exclusive-OR's). Ideally, this sequence can be executed in seven cycles on a parallel machine:

```
X0 = X0 <<< 13      X2 = X2 <<< 3
X1 = X1 ⊕ X0        X3 = X3 ⊕ X2        T1 = X0 << 3
X1 = X1 ⊕ X2        X3 = X3 ⊕ T1
X1 = X1 <<< 1       X3 = X3 <<< 7
X0 = X0 ⊕ X3        X2 = X2 ⊕ X3        T2 = X1 << 7
X0 = X0 ⊕ X1        X2 = X2 ⊕ T2
X0 = X0 <<< 5       X2 = X2 <<< 22
```

The inverse linear transform, diagrammed in Figure 2, also has 16 operations; however, it can be computed in five cycles:

```
X0 = X0 >>> 5       X2 = X2 >>> 22      T1 = X1 ⊕ X3        T2 = X3 >>> 7
X0 = X0 ⊕ T1        X2 = X2 ⊕ X3        T3 = X1 << 7        X1 = X1 >>> 1
X1 = X1 ⊕ X0        X2 = X2 ⊕ T3        T4 = X0 << 3
X1 = X1 ⊕ X2        X3 = X3 ⊕ X2        X0 = X0 >>> 13
X3 = X3 ⊕ T4        X2 = X2 >>> 3
```

On PA-RISC, the single-cycle fixed rotation allows both transforms to execute in eight cycles, optimal for the two-way superscalar machine. The two-cycle rotation on IA-64 increases the operation count to 22, and the dependencies are such that the best implementation for the transforms requires 12 cycles. Loading and XORing the key material in parallel with the transforms can reclaim some performance; however, the linear transformation accounts for over 50% of the encryption and decryption cycles.

As with keying, the best performance is achieved with straight-line code. The program source for both PA-RISC and IA-64 make heavy use of macros and bear strong resemblance to the algorithm specification. An extension of the software tools used to analyze Serpent equations actually produces the raw instruction stream for each equation, in either machine language format, which is then easily integrated into the source program through the macro definitions.

Performance

Cycles	Pentium	PA-RISC			IA-64	IA-64++
		Min	Average	Max		
Keying	1292	668	668.79	669	475	380
Encryption	900	580	580	580	565	468
Decryption	885	585	586.62	587	631	407

On PA-RISC, Serpent keying executes in 668 cycles, compared to the best-reported Pentium results of 1292, almost a 2:1 performance advantage. Encryption and decryption also run substantially faster: a 35.6% advantage for encryption (580 vs. 900) and a 33.9% advantage for decryption (585 vs. 885).

On IA-64, the extra parallelism pays off handsomely in keying, where the routine completes in 475 cycles, a 2.7:1 performance gain over Pentium. Encryption and decryption, with the extra cycles required to complete the linear transform, are better than Pentium, although not as overwhelmingly: 37.2% for encryption (565 vs. 900), 28.7% for decryption (631 vs. 885). For IA-64++, keying is estimated to run in 380 cycles, 3.4 times faster than Pentium, encryption in 468 cycles (48.0% faster), and decryption in 407 cycles (54% faster).

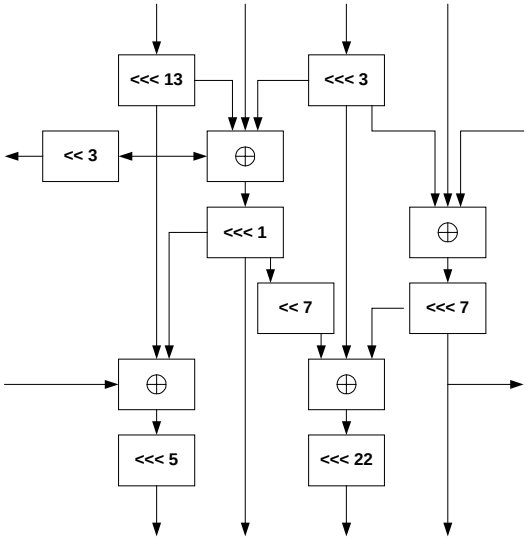


Figure 1 – Serpent Linear Transform

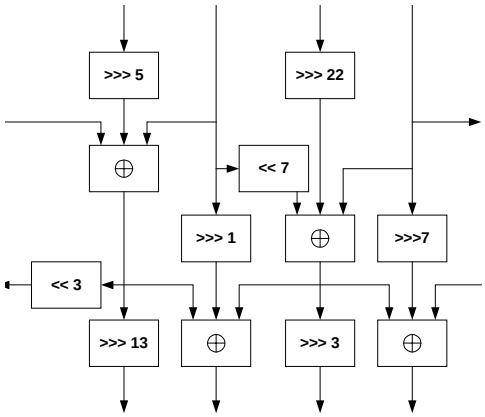


Figure 2 – Serpent Inverse Transform

	AES Submission				Gladman (Best)				Osvik			
	Ops	**Cycles**	**Width**	**Tmps**	**Ops**	**Cycles**	**Width**	**Tmps**	**Ops**	**Cycles**	**Width**	**Tmps**
S Box 0	18	9	6	4	15	6	4	3	17	6	4	5
S Box 1	18	9	5	3	14	8	3	2	17	7	3	3
S Box 2	16	9	3	4	16	8	3	3	14	7	3	5
S Box 3	18	7	5	4	16	8	5	3	21	4	6	6
S Box 4	19	7	4	5	15	8	3	3	19	9	3	3
S Box 5	17	8	3	4	16	7	4	3	18	7	3	3
S Box 6	19	6	7	4	17	6	5	4	17	9	3	3
S Box 7	19	8	4	3	17	11	3	3	19	8	4	5
I Box 0	19	8	5	4	15	10	2	2	18	8	3	4
I Box 1	18	9	3	3	17	7	5	2	18	11	3	3
I Box 2	18	7	5	4	16	8	4	3	18	7	3	3
I Box 3	17	7	4	3	17	9	4	4	17	8	3	3
I Box 4	17	7	4	4	17	6	5	5	19	11	3	3
I Box 5	17	7	5	4	16	7	4	3	18	10	2	3
I Box 6	19	6	4	4	17	8	4	2	16	8	3	3
I Box 7	18	9	4	2	17	9	3	2	18	8	4	4

Table 1 - Serpent IA-64 Metrics

	AES Submission			Gladman (Best)			Osvik		
	Ops	**Cycles**	**Tmps**	**Ops**	**Cycles**	**Tmps**	**Ops**	**Cycles**	**Tmps**
S Box 0	18	11	3	15	8	2	17	9	2
S Box 1	18	11	3	14	8	2	17	9	3
S Box 2	16	11	4	16	9	3	14	8	3
S Box 3	18	9	4	16	9	3	17	9	3
S Box 4	19	10	6	15	8	3	19	10	1
S Box 5	17	9	4	16	9	3	18	9	1
S Box 6	19	10	4	15	9	3	17	10	2
S Box 7	19	10	3	17	12	5	19	10	2
I Box 0	19	10	4	15	10	2	18	11	2
I Box 1	18	10	3	17	9	3	18	11	2
I Box 2	18	10	3	16	9	2	18	10	2
I Box 3	17	9	3	17	9	4	17	9	1
I Box 4	17	9	5	17	9	4	19	11	2
I Box 5	17	9	4	16	8	4	18	10	2
I Box 6	19	10	5	17	9	3	16	8	2
I Box 7	18	9	3	17	10	2	18	9	2

Table 2 - Serpent PA-RISC Metrics

Twofish

The Twofish block cipher employs a *"Feistel-like structure with additional whitening of the input and output."*[6] The 128-bit plaintext block is split into four 32-bit words. In the input whitening step each 32-bit word is XORed with a different 32-bit input-whitening key. This is followed by 16 rounds in which the left two words are transformed by the F-function. The leftmost word produced by the F-function is XORed with the third word, and the result is rotated to the right by one bit. The rightmost word produced by the F-function is XORed with the fourth word, which previously had been rotated to the left by one bit. For all but the 16th round, the left and right pairs of words then are swapped for the next round. Each of the final four words is XORed with a different 32-bit output-whitening key.

Within the F-function, the first input word is transformed by the g-function. The second input word first is rotated to the left by eight bits, and then transformed by the g-function. The two g-function outputs then are mixed into two new words by a Pseudo-Hadamard Transform (PHT). After mixing, a different round key is added to each of the two new words, producing the two output words of the F-function.

The g-function may be implemented in a variety of ways, depending upon one's choice of keying strategy. Twofish defines five different keying strategies: Compiled, Full, Partial, Minimum, and Zero. These choices enable a wide range of time/memory trade-offs for a Twofish implementation.

For RISC and EPIC microprocessors, the choice of Full keying is the most natural. Full keying requires 4096+128+32 = 4256 bytes of table for the four key-dependent S-boxes, 32 round keys, and eight whitening keys. This table size poses no problem for a modern computer platform. Compiled keying is able to reduce the Twofish Pentium-Pro encryption time from 315 cycles to 258 cycles, but it necessitates a separate copy of the encryption and decryption codes for each different key. For superscalar RISC and EPIC microprocessors, Compiled keying is unlikely to result in a performance gain. Given sufficiently many general registers, key loading always can be overlapped and executed in parallel.

The heart of the Twofish g-function is defined as:

1. Partition the 32-bit input word into four 8-bit bytes.
2. Use the value of each of the four bytes to index and fetch a new byte value from a corresponding, 256 byte, key-dependent S-box.
3. Matrix multiply the MDS matrix, a predefined, maximal distance separation byte matrix by the vector of the four bytes fetched from the S-boxes. Scalar multiply of bytes in $GF(2^8)$ is represented as GF(2)[x] modulo v(x), where v(x) is the primitive polynomial $x^8+x^6+x^5+x^3+1$. Scalar addition of bytes in $GF(2^8)$ is XOR.

For Full keying, each of the four S-boxes contains 256 32-bit words, rather than 256 8-bit bytes. Each 32-bit word of $S\text{-box}_{32}[i]$ is the four-byte vector computed by matrix multiplication of the MDS matrix by the four-byte vector whose sole non-zero component is the byte $S\text{-box}_8[i]$. If we denote matrix multiplication by +.×, and the bytes of a column vector, least significant byte first, as [B0:B3] or [B0, B1, B2, B3] the 32-bit S-boxes are:

$$S\text{-box}0_{32}[i] = MDS +.\times [S\text{-box}0_8[i], 0, 0, 0]$$
$$S\text{-box}1_{32}[i] = MDS +.\times [0, S\text{-box}1_8[i], 0, 0]$$
$$S\text{-box}2_{32}[i] = MDS +.\times [0, 0, S\text{-box}2_8[i], 0]$$
$$S\text{-box}3_{32}[i] = MDS +.\times [0, 0, 0, S\text{-box}3_8[i]]$$

In this manner, all $GF(2^8)$ byte multiplications of the g-function MDS matrix multiply are pre-computed, and saved in the 32-bit S-boxes. With these S-boxes, all that is required for a g-function MDS matrix multiplication is to fetch a 32-bit word from each of the four S-boxes and XOR the words together. Therefore, the Full keying computation of the g-function consists of extracting four 8-bit bytes from the input word, using each extracted byte to index and fetch a 32-bit word from a corresponding S-box, and XORing the four fetched words. The rotation by eight bits of the right input word to the F-function actually requires no explicit computation. It is accomplished simply by the order in which 8-bit bytes are extracted from the input word. Similarly, no computation is required for word swapping between rounds.

Keying

Full keying for a Twofish 128-bit user-supplied key proceeds in three phases. In each phase the approach taken utilizes modestly sized tables to accelerate the performance. The user-supplied key is taken as four 32-bit words, in little-endian byte order. These words are called M_0, M_1, M_2, and M_3. Their byte contents, respectively are: $[m_0:m_3]$, $[m_4:m_7]$, $[m_8:m_{11}]$, and $[m_{12}:m_{15}]$, where m_i is the i'th byte of the user-supplied key.

In the first phase of keying, two four-byte vectors denoted S_0 and S_1 are derived from the user-supplied key. These vectors are utilized in the computation of the S-boxes. S_0 and S_1 each are computed by a matrix multiplication of the RS matrix by an eight-byte vector of user-supplied key bytes. The 4×8 RS matrix is derived from a Reed-Solomon code, and is specified by the Twofish definition. Specifically:

$$S_0 = [RS] +.\times [m_0:m_7] \qquad S_1 = [RS] +.\times [m_8:m_{15}]$$

For the RS matrix multiplication, scalar multiply of bytes in $GF(2^8)$ is represented as GF(2)[x] modulo w(x), where w(x) is the primitive polynomial $x^8+x^6+x^3+x^2+1$. Scalar addition of bytes in $GF(2^8)$ is XOR. The actual computation of these two matrix multiplications is accomplished by simulating the LFSRs for the RS code. Doug Whiting programmed this in the following manner in the original Twofish submission.

---

[6] Schneier, Kelsey, Whiting, Wagner, Hall, Ferguson, *The Twofish Encryption Algorithm*, John Wiley & Sons, 1999.

```
#define      RS_GF_FDBK   0x14D        /* field generator */
#define      RS_rem(x)                                                  \
{ BYTE  b  = x >> 24;                                                   \
  DWORD g2 = ((b << 1) ^ ((b & 0x80) ? RS_GF_FDBK : 0 )) & 0xFF;        \
  DWORD g3 = ((b >> 1) & 0x7F) ^ ((b & 1) ? RS_GF_FDBK>>1 : 0) ^ g2;    \
        x = (x << 8) ^ (g3 << 24) ^ (g2 << 16) ^ (g3 << 8) ^ b;
}
```

S_0 and S_1 then can be calculated by the following triply-nested loop, where `M[i]` denotes M_i and `S[i]` denotes S_i:

```
for( i = 0; i < 2; ++i ) {
    for( j = 0, r=0; j < 2; ++j ) {
        r ^= (j) ? M[i*2] : M[i*2+1];
        for( k = 0; k < 4; ++k ) {
            RS_rem( r );
        }
        S[i] = r;
    }
}
```

The calculation of S_0 and S_1 can be accelerated by using a pre-computing a table of 32-bit words, `RStbl[256]`, where `RStbl[i]` = `RS_prem(i)`. `RS_prem(x)` is identical to `RS_rem(x)` but without the `(x << 8)` term in the final assignment statement. Each cycle of the LFSRs then may be simulated simply by:

```
unsigned int x;
#define   RS_rem(x)  x = (x << 8) ^ RStbl[x >> 24];
```

The triply-nested loop to compute S_0 and S_1 is completely unrolled. Housekeeping instructions may be executed in parallel with this computation.

The second phase of keying is to compute the four key-dependent S-boxes. Four pre-computed, 256 entry, 32-bit word auxiliary tables are utilized to accelerate this computation. These tables, denoted `MD0`, `MD1`, `MD2`, and `MD3`, are similar to the Full key S-boxes. Two additional 256 entry, 8-bit tables are required for the S-box computation. These are the tables containing the basic q0 and q1 byte permutations defined in the Twofish specification. These tables are denoted `q0` and `q1`. Each auxiliary table entry combines the final q0 or q1 byte permutation of the S-box computation, and the MDS matrix multiplication. Specifically :

```
MD0[i]  =  MDS +.× [q1[i], 0, 0, 0]
MD1[i]  =  MDS +.× [0, q0[i], 0, 0]
MD2[i]  =  MDS +.× [0, 0, q1[i], 0]
MD3[i]  =  MDS +.× [0, 0, 0, q0[i]]
```

This is the same matrix multiplication used in the g-function. Each Full key S-box contains exactly the same 32-bit words as the corresponding auxiliary table, but permuted according to the user-supplied key. If we designate the bytes of the words S_0 and S_1 as S0(3:0) and S1(3:0), byte zero being least significant, the Full key S-box computation loop is:

```
for( i = 0; i < 256; ++i ) {
   S-box0₃₂[i] = MD0[ q0[ q0[i]^S0(0) ] ^ S1(0) ];
   S-box1₃₂[1] = MD1[ q0[ q1[i]^S0(1) ] ^ S1(1) ];
   S-box2₃₂[i] = MD2[ q1[ q0[i]^S0(2) ] ^ S1(2) ];
   S-box3₃₂[i] = MD3[ q1[ q1[i]^S0(3) ] ^ S1(3) ];
}
```

This computation further can be accelerated by yet another, 256-entry, auxiliary 32-bit word table. This table is called `q0q1q0q1`. The i'th entry of this table consists of `[ q0[i], q1[i], q0[i], q1[i] ]`. The word `q0q1q0q1[i]` can be fetched by a single instruction, and can be XORed with S_0. This computes the inner XOR of all four assignment statements in parallel. Each byte of this intermediate result then is used to fetch a byte from q0 or q1. Following one more XOR with the corresponding byte of S_1, the $S\text{-box}_{32}$ entry is obtained by indexing and fetching the 32-bit word from the proper MD table. This word is stored into the proper 32-bit S-box.

The code to perform this computation is organized as a 256-pass loop for both PA-RISC and IA-64. The S_0 and S_1 words already reside in general registers. For each loop iteration, the required operations are one indexed load for the q0q1q0q1 table entry, a word XOR with S_0, four byte extracts[7], four indexed byte loads from the q0 and q1 tables, four XORs with S_1 bytes, four indexed word loads from the MD tables, four indexed word stores to the S-boxes, and a loop closing instruction. For IA-64, eight additional instructions are required for computing table addresses. IA-64 post address modification is used for indexing the q0q1q0q1 table and the S-boxes.

The total number of 256-entry tables used to accelerate the computation of S_0, S_1, and the key-dependent S-boxes is eight, occupying 6656 bytes. These table sizes are quite acceptable for a modern RISC or EPIC platform. No IA-64 bank optimization was done for these tables[8]. No additional tables are required for the third phase of keying.

1. q0 256 bytes
2. q1 256 bytes
3. q0q1q0q1 1024 bytes
4. MD0, …, MD3 1024 bytes each, 4096 bytes total
5. RStbl 1024 bytes

[7] The four S_1 byte extracts are done outside the loop.

[8] Described in the next section.

The third and final phase of keying is the computation of the 40 whitening and round keys. This code is similar to the computation of the S-boxes. It is organized as a 20-iteration loop, in which two keys are computed per iteration. Unlike the S-box computation, each key requires a full MDS matrix multiply. Further, a final PHT transform is applied to each pair of keys. The Twofish definition systematically uses the same MDS matrix multiply and PHT operations in the keying algorithms and in the encryption and decryption algorithms.

The same table techniques used above are used to accelerate computation of the whitening and round keys. The initial eight of the 40 keys are taken as the input and output whitening keys. The final 32 keys are taken as the round keys. Using the previously defined notations, and K to denote the newly computed keys, the computation for the 40 whitening and round keys is:

```
for( i = 0; i < 40; i += 2 ) {
   T0  = MD0[ q0[ q0[i]^M2(0) ] ^ M0(0) ];
   T0 ^= MD1[ q0[ q1[i]^M2(1) ] ^ M0(1) ];
   T0 ^= MD2[ q1[ q0[i]^M2(2) ] ^ M0(2) ];
   T0 ^= MD3[ q1[ q1[i]^M2(3) ] ^ M0(3) ];
   T1  = MD0[ q0[ q0[i+1]^M3(0) ] ^ M1(0) ];
   T1 ^= MD1[ q0[ q1[i+1]^M3(1) ] ^ M1(1) ];
   T1 ^= MD2[ q1[ q0[i+1]^M3(2) ] ^ M1(2) ];
   T1 ^= MD3[ q1[ q1[i+1]^M3(3) ] ^ M1(3) ];
   T1  = (T1 <<< 8);
   T0 += T1;
   T1 += T0;
   T1  = (T1 <<< 9);
   K[i] = T0;
   K[i+1] = T1;
}
```

The code to perform this computation is organized as a 20-pass loop for both PA-RISC and IA-64. Note that the M_i words are used in even-subscript and odd-subscript pairs. Also note that the M_i words are used in an order reversed from the order of the S_i words in the S-box computation. The M_0, M_1, M_2, and M_3 words already reside in general registers. For each loop iteration, the required operations are two indexed loads for the q0q1q0q1 table entries, two word XORs with M_2 and M_3, eight byte extracts[9], eight indexed byte loads from the q0 and q1 tables, eight XORs with M_0 and M_2 bytes, eight indexed word loads from the MD tables, six XORs to complete the MDS matrix multiplies, two rotates, one add and one shift-and-add for the PHT, two indexed word stores to the key array, and a loop closing instruction. For IA-64, sixteen additional instructions are required for computing table addresses. IA-64 post address modification is used for indexing the q0q1q0q1 table and the key array.

Encryption

For PA-RISC the encryption and decryption functions are organized as straight-line code. Each is provided two pointer arguments, the first to the 16-byte cleartext block or ciphertext block, the second to the concatenation of the round keys, whitening keys, and four Full key S-boxes. Input blocks are whitened 64-bits at a time. Housekeeping instructions are overlapped with the first and last rounds.

Each PA-RISC round, including the one-bit circular shifts, executes in about a dozen cycles. PA-RISC includes an instruction that can extract any contiguous 8-bit field from a word in one cycle. The extracted byte can be used directly as an index for a 32-bit word load instruction. Further, the PA-RISC shift-and-add instruction permits the PHT to be done in two instructions during the same cycle. Thus, each round needs 32 instructions: eight extract instructions (`extrw,u`), eight instructions to load from S-boxes (`ldw,s`), two instructions to load round keys (`ldw`), two one-bit circular shift instructions (`shrpw`), eight XOR instructions (`xor`), three add instructions (`add,l`), and one shift-and-add instruction (`shladd,l`). The instruction schedule is nearly optimal, but the final right rotate by one bit adds one cycle to the round.

For IA-64 the encryption and decryption functions are organized in exactly the same way. In each round, an additional instruction is required to compute an S-box address from each extracted byte. Although this requires eight additional instructions, there also is an added benefit. Microprocessor caches often are organized as independent 8-byte banks. An optimal memory strategy, therefore, shuffles the four S-boxes, so that each S-box is entirely contained in a single cache bank. This results in a 16-byte stride between successive S-box words. The IA-64 shift-and-add instructions, used to compute S-box addresses, therefore, use a shift value of four. This assures the absence of cache bank conflicts when executing two S-box loads during the same cycle.

A second technique employed for IA-64 is computational height reduction, a practice common for parallel instruction issue machines. Additional instructions are executed, but the entire computation completes in fewer cycles.

In Twofish, the rightmost bit of the first F-function output becomes the high order bit of a byte to be extracted in the next round. For PA-RISC, the fact that the extract instruction demands a contiguous bit field requires that the one-bit right rotate be done after computation of the first F-function output and prior to the extract for the next round. For IA-64, parallelism and predicates offer a better solution.

The first F-function output is computed as three XORs, two adds, and a final XOR. Although these operations do not commute or freely associate, they in fact do so for the rightmost bit, which actually is the result of six XORs. By computing the rightmost bit of the last XOR sooner (round-key XOR third-block-word), one redundantly can compute the rightmost bit of the first F-function output one cycle earlier. This permits the rightmost bit also to be tested without adding a cycle to the round. The result of the test is written to a predicate. This predicate then is used to set a temporary S-box pointer either to the beginning, or to the halfway point, of the corresponding S-box at the start of the next round. Only the seven leftmost bits of the unrotated first F-function output are extracted in

[9] The eight M_0 and M_1 byte extracts are done outside the loop.

the next round. They then are used as an index relative to the temporary pointer. The full first F-function output word can be rotated later.

It also turns out that, with proper table alignment, height reduction can be used to compute two S-box addresses one cycle earlier in the next round. The enabling fact here is that offsets into S-boxes consist of 12 bits, of which the right four are zero. For a 4096-byte aligned and shuffled table, an XOR can be used for the address calculation. The terms for two such XORs redundantly can be computed in the previous round. This can be seen from the following equations for one pair of encryption terms (note: [7:0] denotes the rightmost 8 bits of a word):

```
Let:   PHT1 be the second PHT output for the current Round.
RK1    be the second Key word for the current Round.
BW3    be the fourth Block word for the current Round.
Fin1   be the second input word to the next Round.
PSB1   be the pointer to S-box 1.
pSBE   be the pointer to the S-box 1 entry for Fin1[7:0].[10]

Fin1   = (PHT1+RK1) ⊕ BW3

pSBE   = pSB1  +  16*( Fin1 )[7:0]
       = pSB1  +  16*( (PHT1+RK1) ⊕ BW3 )[7:0]
       = pSB1  +  ( 16*(PHT1+RK1)[7:0] ⊕ 16*BW3[7:0] )
       = pSB1  ⊕  ( 16*(PHT1+RK1)[7:0] ⊕ 16*BW3[7:0] ) [11]
       = (pSB1 ⊕ 16*BW3[7:0]) ⊕ (16*(PHT1+RK1)[7:0])
```

Performance

Cycles	Pentium	PA-RISC			IA-64	IA-64++
		Min	Average	Max		
Keying	8414	2846	2901.79	2964	2445	2445
Encryption	315	205	217.45	233	182	182
Decryption	311	200	210.29	224	182	182

On PA-RISC, Twofish keying executes in 2846 cycles, compared to the best-reported Pentium results of 8414, a 2.96:1 performance advantage. Encryption and decryption also run faster: a 36% advantage for encryption (205 vs. 315) and a 35.7% advantage for decryption (200 vs. 311).

On IA-64, Twofish executes even faster. Twofish keying executes in 2445 cycles, compared to the best-reported Pentium results of 8414, a 3.44:1 performance advantage. Encryption and decryption also run faster: a 42.2% advantage for encryption (182 vs. 315) and a 41.5% advantage for decryption (182 vs. 311).

[10] S-box 1 is used for the rightmost bits because of the logical (Fin_1 <<< 8).

[11] Addition is equivalent to exclusive-or because of the S-box table alignment.

Conclusions

All the algorithms have reasonable implementations on PA-RISC and IA-64; all make good use of the architectures. It is clear that the underlying computer architecture has a direct and significant effect on the optimal implementation for each candidate. The large register files in PA-RISC and IA-64 enable complete state to be kept without using memory, influencing the structure of Rijndael, Twofish, and keying codes. The choice of equations for Serpent is a direct result of the available execution width and ALU operations. Sometimes, effects are expressible only at the assembly level, such as the software pipelines in the Mars keying or the MMU multiplication in RC6 encryption. In other cases, algorithm structures to exploit the underlying architecture are best expressed in high level source, such as the restructuring of the RC6 keying algorithm.

Our second conclusion is that *algorithm performance cannot be measured by a single number*. A complete performance characterization must filter out large system effects such as caching, memory latencies, interrupts, paging, process swaps, and I/O activity, but should draw attention to fine-grain system effects such as cache interference and execution latencies. When timing keying for random input key values, the results will exhibit a performance distribution rather than a single number.

Another consideration is parallelism. Future CPU's will be increasingly, and we believe explicitly, parallel; algorithms that can exploit parallelism will see continuing performance improvement over the life of the new AES algorithm. It should be observed that as better Serpent equations are developed, Serpent will further improve both its performance and parallelism. A final factor in evaluating software is memory usage; none of the finalists use tables uncomfortably large for modern server and desktop systems.

Using these criteria, and assuming that the IA-64++ additions will/will-not be made, the results of this study rank the AES finalists as follows:

Performance	Memory	Parallelism
Rijndael	RC6	Rijndael
RC6/Twofish	Serpent	Twofish
Twofish/RC6	Mars	Serpent
Mars	Twofish	Mars
Serpent	Rijndael	RC6

Acknowledgments

We wish to express our thanks to Doug Whiting for his unerring guidance, especially his prescient counsel in the selection of first-round candidates to investigate. We also wish to thank and to acknowledge the contributions of Dr. Brian Gladman, whose work is well known and appreciated by the AES community. Brian's codes were used to generate test values and, in many instances, improved our understanding of the algorithms. Brian also kept us up to date on his Pentium performance improvements. We appreciate Rohit Bhatia's suggestions for 32×32 multiplies. Dag Arne Osvik contributed his Serpent equations, which forced our equation analysis tools to improve and sped up both the PA-RISC and the IA-64 Serpent implementations.

We are indebted to John Crawford and members of the Intel Itanium team, who provided access and support for the Itanium simulator. Finally, our thanks go to the Hewlett Packard Ft. Collins McKinley team, whose assistance with the development and simulation tools, and patience with our endless questions, was the *sine qua non* of the IA-64 work.

Appendix A:
Summary of Best Performance

Candidate	Encryption					Decryption					Keying				
	Clocks	Ops	IPC	Regs	Bytes	Clocks	Ops	IPC	Regs	Bytes	Clocks	Ops	IPC	Regs	Bytes
Mars															
Pentium	320					374					3894				
New Keying											2128				
PA-RISC	540	631	1.17	12(18)	2588	538	632	1.17	12(18)	2592	1969	2908	1.48	20	2584
New Keying	538	631	1.17	12(18)	2588	537	632	1.17	12(18)	2592	1797	1805	1.00	20	1984
IA-64	511	1013	1.98	18//8	784	527	1013	1.92	18//8	784	1903	3332	1.75	14//48	1344
New Keying											1408	3132	2.22	12//16	976
IA-64++	255					271					1313				
New Keying											1408				
Table Sizes					2048					2208					2208
Alg Parallelism			2.0					2.0					3.0		
RC6															
Pentium	243					226					1632				
PA-RISC	580	577	0.99	12(4)	2308	493	558	1.13	12(4)	2232	1077	1519	1.41	12	760
IA-64	490	826	1.69	4/27/8	480	490	826	1.69	4/27/8	528	1581	2629	1.66	8//56	256
IA-64++	150					130					1057				
Table Sizes					0					176					176
Alg Parallelism			2.0					2.0					2.0		
Rijndael															
Pentium	284					283					1338				
PA-RISC	168	537	3.20	24	2160	168	539	3.21	24	2160	239	686	2.87	28	2800
Fwd Keying											85	228	2.68	19	1504
IA-64	125	704	5.63	20/12	3808	126	706	5.60	20/12	3824	148	822	5.55	24/21	4480
Fwd Keying											104	282	2.71	19	1504
IA-64++	same					same					same				
Table Sizes					8192					8368					8368
Alg Parallelism			10.0					10.0					10.0		
Serpent															
Pentium	900					885					1301				
PA-RISC	580	1273	2.19	17	5100	585	1309	2.24	17	5240	668	1409	2.11	19	5640
IA-64	565	1517	2.61	24	8480	631	1546	2.45	24	8848	475	1527	3.21	22/4	8368
IA-64++	468					407					380				
Table Sizes					0					528					528
Alg Parallelism			3.0					3.0					4.0		
Twofish															
Pentium	315					311					8414				
PA-RISC	205	548	2.67	20	2192	200	548	2.74	20	2192	2846	8904	3.13	30	1324
IA-64	182	927	5.09	23	5184	182	915	5.03	23	4960	2445	9561	3.91	26/21	1600
IA-64++	same					same					same				
Table Sizes					6656					4256					4256
Alg Parallelism			6.0					6.0					4.0		

Notes:

-- IA64++ is a *hypothetical* IA-64 implementation – refer to the text for details. It does not represent any current or planned IA-64 implementation.

-- Twofish times for Full keying are from: *The Twofish Encryption Algorithm*, John Wiley & Sons, 1999.

-- Pentium, Alpha clocks are lowest reported clocks from the NIST Round 1 Report, August 1999.

-- Regs = GRs, or statics/stacked, or statics//rotating, or statics/stacked/rotating, or GRs(FRs) registers.

-- Bytes are object code sizes. Table Sizes are total tables for *keying*, key table plus look-up tables for *encryption* and *decryption.*

-- Alg Parallelism is an estimated integral upper bound for software parallelism.

Appendix B: Mars Keying
Original Implementation

The original Mars keying initializes the first seven elements of an array, `T[-7..39]`, to the first seven entries of the Mars S-box, then sets the rest of the array as follows:

```
T[i] = ((T[i-7] ⊕ T[i-2]) <<< 3) ⊕ k[i mod N] ⊕ i    i = 1 ... 38
T[39] = N
```

where *k* is the input key and `N` is the size, in words, of the input key. This recurrence has an active state of seven words, `A`, `B`, …, `G`, such that the expansion can be rewritten:

```
T[0]  = A = ((A ⊕ F) <<< 3) ⊕ k[0] ⊕ 0
T[1]  = B = ((B ⊕ G) <<< 3) ⊕ k[1] ⊕ 1
T[2]  = C = ((C ⊕ A) <<< 3) ⊕ k[2] ⊕ 2
T[3]  = D = ((D ⊕ B) <<< 3) ⊕ k[3] ⊕ 3
T[4]  = E = ((E ⊕ C) <<< 3) ⊕ k[0] ⊕ 4
                  . . .
T[38] = D = ((D ⊕ B) <<< 3) ⊕ k[2] ⊕ 38
T[39] = N
```

where `A` is initialized to `S[0]`, `B` to `S[1]`, and so forth. When key expansion is complete, the data words are then "stirred" seven times as follows:

```
T[i] = (T[i] + S[T[i-1] & 0x1ff]]) <<< 9       i = 1 ... 39
T[0] = (T[0] + S[T[39] & 0x1ff]]) <<< 9
```

It is possible to overlap the first stirring with the key expansion: after the first eight expansion steps, `T[1]` is no longer involved in the expansion recurrence and can therefore be stirred. This requires adding one extra word to the expansion state so that both `T[i]` and `T[i-1]` are available for stirring. After the stirring, the keys are reordered, mapping `T[i]` → `K[7i mod 40]`

PA-RISC

The PA-RISC implementation uses straight-line coding for the expansion/stir phase, rotating the key words each step. The remaining six stirring passes are executed in a loop as per the specification. The reordering, however, is again straight-line code. If reordering is considered as replacement rather than a permuted copy, the replacements form chains, that is:

`T[1]` → `T[7]` → `T[9]` → `T[23]` → `T[1]`

There are 8 chains of four, 3 chains of two, and two chains of one (`T[0]`→`T[0]` and `T[20]`→`T[20]`). Since PA-RISC can issue two memory operations per cycle but can retire only one store per cycle, the optimal ordering loads from one chain, then interleaves the stores with the loads from the next chain. The expected performance is 40 cycles, which is the number of times a multiple cycle loop would have to run to perform the same task. It also eliminates the need for a temporary key array: the target key array can be used for all intermediate values.

IA-64

The IA-64 Mars keying implementation takes advantage of the large register files, rotating registers, and rotating predicates. The routine allocates a 48-register stack frame, all of which are rotating. The initial register usage is as follows:

r32-r34	r35	r36	r37	r38	r39	r40	r41	r42	r43	r44	r45	r46	r47	r48	r49	r50-r79
Unused	k_X	k_3	k_2	k_1	k_0	T_i	T_{i-1}	T_{i-2}	T_{i-3}	T_{i-4}	T_{i-5}	T_{i-6}	T_{i-7}	A	B	T[10..39]

The first nine computations simply initialize `T`$_i$ and rotate registers to the right. After that, the registers `A` and `B` contain the first two values for stirring. Unlike PA-RISC, this phase of the computation is enabled by the rotating predicates, where a '1' is shifted in each time through the main body of the loop. To circulate the key words, $k_0 \rightarrow k_X$ at the end of the loop. When the initialization phase of the loop is finished, the loop switches to the epilogue phase, which now shifts a '0' into the rotating predicates, which disables the initialization instructions. Thus, the entire expansion/mix phase executes in one loop that runs 48 times, 6 cycles per loop.

When the first phase is finished, the intermediate key values are in the rotating registers, with `r39` = `T[0]`, `r38` = `T[1]`, …, `r32` = `T[7]`, `r79` = `T[8]`, …, `r48` = `T[39]`. This allows the stirring phases to compute on the rotating register file. Since the registers rotate 39 places during the stirring loop, the registers used in each phase are:

Pass	T[i]	T[i-1]	T[0](Final)
2	r39	r38	r78
3	r78	r77	r69
4	r69	r68	r60
5	r60	r59	r51
6	r51	r50	r42
7	r42	r41	r33

The reorder is efficiently handled in a two cycle loop. In the first cycle, the key word is stored, the data pointer incremented seven words, and a look-ahead target index counter is tested for overflow and incremented. In the second cycle, the index and data pointers are adjusted if the index had overflowed in the previous cycle.