A More Compact AES, and More

Dag Arne Osvik
SnT Interdisciplinary Centre for Security Reliability and Trust,
University of Luxembourg, Luxembourg[*]

David Canright
Naval Postgraduate School, Monterey CA 93943, USA,
dcanright@nps.edu

June 19, 2024

Abstract

We reduce the number of bit operations required to implement AES to a new minimum, and also compute improvements to elements of some other ciphers. Exploring the algebra of AES allows choices of basis and streamlining of the nonlinear parts. We also compute a more efficient implementation of the linear part of each round. Similar computational optimizations apply to other cryptographic matrices and S-boxes. This work may be incorporated into a hardware AES implementation using minimal resources, or potentially in a bit-sliced software implementation to increase speed.

keywords: AES, compact, tower field, composite field

1 Introduction

The Advanced Encryption Standard was designed to be efficiently implemented in either software or hardware. Some early approaches for resource-limited hardware, e.g., [4, 5, 15, 16, 21], sought to minimize circuitry, often through a different representation of the finite byte field $GF(2^8)$ built on subfields.

In 2009 we sought to minimize the number of (two-input) bit operations required to compute the full AES128. The result[3] was, in this sense, the smallest AES for a decade, to the best of our knowledge.

One of the more recent advances is the innovative approach of Ueno et al.[17, 18], using multiple different representations of subfield elements for certain efficiencies in the 8-bit inverter of the S-box, resulting in the best area-time product for that step. Another recent advance is improved computational approaches to minimize circuitry, both linear and nonlinear, such as in the work of Maximov and Ekdahl [9, 10]; they find the “fastest and

[*]APSIA

smallest" S-boxes, and smallest MixColumns. The "smashing" S-box of [14] set new records in terms of area in gate equivalents and in speed (but not in bit operations per byte).

In the present work we seek to further reduce the size of AES, in terms of the number of bit operations. As in our previous work [2, 3] we use the "tower-field" representation of Paar[13] and Satoh[16], maintaining this representation through full rounds. Analyzing the S-box showed many efficient basis choices; we considered 1008 of them. Computationlly optimizing the linear parts showed that three basis choices were smaller than the rest; we optimized one of those three for minimal bit ops. (The logic gates used are AND, OR, XOR, occasionally complemented.)

These improvements result in one full AES round needing only 134 bit operations per byte, and the full 10-round AES128 needing only 1338 bit operations per byte. For comparison, the corresponding numbers from our earlier work[3] are 139.5 and 1391.5 respectively; the new result is 4% better. (Note: we cannot directly compare with others' results for S-box or MixColumns separately, since our approach divides a round into the nonlinear inverse and the linear part following.)

Below, first we explore the AES algorithm and describe our approach in Section 2. Next, Section 3 discusses our optimizations of the 4-bit inverter and the 32-bit linear parts of AES, and some related results. Lastly, Section 4 summarizes our results and conclusions.

2 Algebra of AES

The Advanced Encryption Standard is a symmetric block cipher with 128-bit blocks and three key sizes: 128, 192, or 256 bits[11]. The block is split into 4 32-bit columns, and each column is split into 4 bytes; the block is thus considered as a 4×4 array of bytes. The block is processed through 10, 12, or 14 rounds (according to key size), and the key schedule generates a 128-bit round key for each round from the given key.

Each round is defined in four steps:

1. *SubBytes*, also called the S-box, replaces each byte value with a different one. Software can use a look-up table, but this table is algebraically described in two sub-steps:

   (a) The byte $\mathbf{x}$, considered as an element of the field $GF(2^8)$, is replaced by its multiplicative inverse: $\mathbf{y} = \mathbf{x}^{-1}$ (a zero byte is unchanged, so strictly this is $\mathbf{y} = \mathbf{x}^{254}$).

   (b) The result, as a vector of bits, is subject to an affine transformation: $\mathbf{z} = A\,\mathbf{y} + \mathbf{b}$, where A is a fixed bit matrix and $\mathbf{b}$ a fixed bit vector (with bit arithmetic in $GF(2)$, where multiplication is AND and addition is XOR).

2. *ShiftRows* rotates each byte row of the 4×4 array by 0–3 bytes, according to row position.

3. *MixColumns* multiplies each byte column $\mathbf{c}$ by a fixed byte matrix $M = \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix}$, with byte arithmetic in the field $GF(2^8)$: $\mathbf{d} = M\,\mathbf{c}$.

4. *AddRoundKey* bitwise adds (XOR) the round key to the 128-bit state.

The first round is preceded by an *AddRoundKey* step (we call this round 0), and the last round skips the *MixColumns* step. Note that ShiftRows just moves bytes around, so can come before SubBytes. (We will not consider decryption here.)

By definition, AES treats a byte in the field $GF(2^8)$ as the coefficients of a polynomial of degree less than 8: bits $[b_7\, b_6\, b_5\, b_4\, b_3\, b_2\, b_1\, b_0]$ represent polynomial $b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x + b_0$, where x is a root of the polynomial $q(x) = x^8 + x^4 + x^3 + x + 1 = 0$. This type of representation is called a polynomial basis. There are very many other possible choices of basis, where the bits are coefficients of some other selection of field elements. In any basis, field addition is bitwise XOR. For the standard polynomial basis, field multiplication is polynomial multiplication modulo $q(x)$ with coefficient arithmetic mod 2.

The "tower field" approach builds $GF(((2^2)^2)^2)$ as successive extensions of degree 2 of subfields; this has been shown[16, 2] to give an efficient way to calculate the "inverse" part of the S-box. In this work, as in [3], we use the tower field throughout complete rounds, not just for the S-box.

The 2-bit field $GF(2^2)$ includes 0, 1 and two more elements; if we call one of them Ω, then it is a root of $\Omega^2 + \Omega + 1 = 0$ and the other root is $\Omega^2 = \Omega + 1$. These two roots are called *conjugates* of each other, so we may notate them as $\Omega, \bar{\Omega}$ (the overbar means conjugate); their Trace is their sum $\Omega + \bar{\Omega} = 1$; their Norm is their product $\Omega * \bar{\Omega} = 1$. We can represent field elements by 2 bits $[b_0, b_1]$ as coefficients with a polynomial basis $[1, \Omega]$ or $[1, \bar{\Omega}]$, or use both conjugates in a *normal* basis $[\Omega, \bar{\Omega}]$.

(Note: in this section we adopt little-endian ordering; bit variables are lower-case letters, 2-bit variables are capital Greek, 4-bits are lower-case Greek, 8-bit variables are upper-case letters.)

Analogous representations apply in the 4-bit (nibble) field $GF(2^4)$. For $\eta \in GF(2^4)$ a defining element (not in the subfield), then η and its conjugate $\bar{\eta}$ are the roots of $\eta^2 + T\eta + N = 0$, with Trace $\eta + \bar{\eta} = T \in GF(2^2)$ and Norm $\eta * \bar{\eta} = N \in GF(2^2)$ (those are Greek Tau, Nu). And a pair $[\Theta_0, \Theta_1]$ of elements of $GF(2^2)$ can represent an element of $GF(2^4)$ over a polynomial basis $[1, \eta]$ or $[1, \bar{\eta}]$, or a normal basis $[\eta, \bar{\eta}]$.

Similarly in the 8-bit (byte) field $GF(2^8)$. For $W \in GF(2^8)$ a defining element, W and $\bar{W}$ are the roots of $W^2 + \tau W + \nu = 0$, with Trace $W + \bar{W} = \tau \in GF(2^4)$ and Norm $W * \bar{W} = \nu \in GF(2^4)$. And a pair $[\theta_0, \theta_1]$ of elements of $GF(2^4)$ can represent an element of $GF(2^8)$ over a polynomial basis $[1, W]$ or $[1, \bar{W}]$, or a normal basis $[W, \bar{W}]$.

In a tower field, using a normal basis for $GF(2^8)$ saves 5 XORs in the "inverse" step, so we only considered those. Now to find the inverse of $X = \alpha\, W + \beta\, \bar{W}$ with coefficients $\alpha, \beta \in GF(2^4)$, then its conjugate is $\bar{X} = \alpha\, \bar{W} + \beta\, W$ (i.e., swap coefficients, the subfield is unchanged). Then

$$\mathrm{Norm}(X) = X\bar{X} = (\alpha\beta)(W^2 + \bar{W}^2) + (\alpha^2 + \beta^2)(W\bar{W}) = \tau^2(\alpha\beta) + \nu(\alpha + \beta)^2 = \gamma$$

with $\gamma \in GF(2^4)$. Next we need its inverse γ^{-1} in $GF(2^4)$. Finally

$$X^{-1} = (X\bar{X})^{-1}\bar{X} = \gamma^{-1}(\beta\, W + \alpha\, \bar{W}) = (\gamma^{-1}\beta)\, W \ + \ (\gamma^{-1}\alpha)\, \bar{W}$$

Thus the 8-bit inverter consists of several 4-bit operations: the Norm function, involving a multiply with scaling by a constant, and a sum-square-scale by another constant, added together; the 4-bit inverter; and two more 4-bit multiplies.

A 4-bit multiplier includes three 2-bit multipliers with scaling by constants (T, N), and four or more sums. For a normal basis $\eta, \bar{\eta}$, the product of $A\,\eta + B\,\bar{\eta}$ and $\Gamma\,\eta + \Delta\,\bar{\eta}$ uses an intermediate step:

$$E = (N/T)((A+B)(\Gamma+\Delta))$$

$$(T(A\Gamma) + E)\,\eta \ + \ (T(B\Delta) + E)\,\bar{\eta}$$

For a polynomial basis $1, \eta$, the generic case is more complicated, but for each specific basis choice, the multiplier can always be expressed with three 2-bit multipliers with scaling and at least 4 sums. But for choices other than η a root of $\eta^4 + \eta + 1 = 0$ there are more than 4 sums, so those other two families of 4 isomorphic conjugates were not competitive. With that chosen η, the trace $T = 1$, and the product of $A + B\,\eta$ and $\Gamma + \Delta\,\eta$ is:

$$E = (A\Gamma)$$

$$(N(B\Delta) + E) \ + \ ((A+B)(\Gamma+\Delta) + E)\,\eta$$

A breakthrough realization in this work was that, for the 2-bit multipliers, scaling is free! For either polynomial or normal basis, for any nonzero constant $\Gamma \in GF(2^2)$, the scaled product ΓAB can always be computed with three multiplies (ANDs) and four adds (XORs). This allowed consideration of tower bases we had previously thought uncompetitive.

In the 4-bit Norm function (of the 8-bit input), the $\tau^2(\alpha\beta)$ multiply-scale involves up to six 2-bit scaled multiplies, but some cases only need four, two are zeros; we only considered the latter. For a normal 4-bit basis, this happens when the trace τ of the 8-bit basis is in the 2-bit field; of the 30 possible families in $GF(2^8)$ of 8 isomorphic conjugates (or minimal polynomials), 6 have this property, giving $6 \times 4 \ \times \ 3 \times 2 \ \times \ 3 = 432$ tower basis choices. Those same 6 families also give two zeros, in different terms, for the polynomial 4-bit basis, giving another $6 \times 4 \ \times \ 4 \ \times \ 3 = 288$ basis choices. And for each W in the other 24 families, there is one 4-bit polynomial basis where $\eta = \tau^2$ that gives two zeros, for another $24 \times 4 \ \times \ 1 \ \times \ 3 = 288$ more basis choices. Altogether, there are 1008 cases we looked at (including the 16 cases considered in [3]).

The other part of the Norm function, $\nu(\alpha + \beta)^2$, is linear. Expanding to the bit level, all of the bit sums in that part could be eliminated by changing some ANDs to ORs in the nonlinear part. (Sometimes both AND and OR were needed for one product in different terms, but that still replaces two XORs with one OR.)

The 4-bit inverter was computationally optimized (see Section 3).

After the inversion substep of the S-box, the rest of the round is linear. The affine transformation $A\,\mathbf{y} + \mathbf{b}$, combined with the scaling (by 1, 2, 3 in the standard basis) of MixColumns, can be computed with two 16×8 bit matrices, as in [3]. (The additive constant $\mathbf{b}$ is handled by changing some XORs to XNORs.) For special rounds other 8×8 bit matrices are needed: conversion from standard to tower basis in round 0; and affine combined with conversion back to standard basis in the last round.

Of course, in this approach all of the round keys also need to be in the tower basis, except the last is in the standard basis. (Appendix A includes a key schedule in the tower basis.)

We applied the "Shortest Linear Program" algorithm of Boyar and Peralta[1], with cosmetic improvements by [7], to the 16×8 matrices for initial ranking of the 1008 candidates, then applied more thorough optimization to find the best candidates. The top three tied, significantly ahead of the rest. We chose this basis (expressed in standard AES hex form):

$$[W = \mathrm{0x1B}, W^{16} = \mathrm{0xFA}] \times [1 = \mathrm{0x01}, \eta = \mathrm{0x5C}] \times [\Omega = \mathrm{0xBD}, \Omega^2 = \mathrm{0xBC}]$$

giving traces and norms $\tau = \mathrm{0xE1}, \nu = \mathrm{0xBD} = \Omega, T = \mathrm{0x01} = 1, N = \mathrm{0xBC} = \Omega^2$, and by design $\tau^2 = \eta$. In the end, rather than use those 16×8 matrices, we optimized the whole 32×32 matrix of affine transformation and MixColumns for a column.

When we started this new work on making AES smaller, we did not immediately assume the same approach as before[3]. Rather, we were looking for some new approach. One idea we explored was a redundant 5-bit representation of elements of $GF(2^4)$, as in [17]. There, two different forms were used: the Polynomial Ring Representation (PRR) and the Redundant Representation Basis (RRB). Both use 5 bits as coefficients to the 5 roots of $x^5 - 1 = (x-1)(x^4 + x^3 + x^2 + x + 1) = 0$; PRR is a linear code uniquely representing each element of $GF(2^4)$ while RRB may use either the PRR representation *or* its bitwise complement. In either form, squaring is free, and so is multiplying by a constant that is a basis element. The PRR form is efficient for the inverse in $GF(2^4)$, while the RRB form gives an efficient multiplier. This novel approach significantly reduces the depth (hence delay) of the $GF(2^8)$ inverter, compared to a $GF(((2^2)^2)^2)$ tower-field implementation. However, when we tried to minimize size with redundant representations, including transforming them back to the tower basis, we could not get it small enough (in terms of bit ops) to be competitive[1].

Another new approach we considered was to use a $GF(2^8)$ normal basis over $GF(2)$, i.e., the 8 roots of an irreducible polynomial of degree 8. (There are 16 choices for the polynomial.) This gives rotational symmetry to the algebra. So if the first bit of the inverse $\mathbf{y}$ of $\mathbf{x}$ is given by $y_0 = f(x_0, x_1, x_2, x_3, x_4, x_5, x_6, x_7)$ then the next bit is $y_1 = f(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_0)$ and so on; each uses the same Boolean function f. The hope was to use this symmetry to find an efficient inverter, but we did not. However, we have found a compact implementation of f (see Appendix E). This could form the core of a very compact inverter computing one output bit per clock cycle, using shift registers for the input and output bits. If the required MUXs and flip-flops are small enough, this could lead to a new size record while allowing a high clock speed. This bit-serial inverse might be useful in the byte-serial "Sub-Atomic AES"' implementation of Wamser and Sigl. Their 255-clock-long inverse loop could be replaced by the 8-clock loop of our rotational approach, for increased speed at a modest increase in size.

3 Computational Optimizations

We (Osvik) developed powerful tools for optimizing linear and nonlinear circuits. The linear optimizer, for bit matrix transformations, is a depth-first traversal where width is a parameter, with heuristics for efficiency.

This method gave efficient implementations for the linear transformations in our small AES as well as those used in our key schedule (see code in Appendix A). The most important was optimizing the "AfMix32" 32×32 matrix that applies the affine transformation of the

[1]For example, the $GF(2^8)$ inverter in [17] (which does not convert back from the RRB form) uses 109 bit ops; they show comparison with the inverter of [2], which uses 96 ops.

Table 1: **Linear Transformations**. Each line shows the name of the 8×8 transformation matrix, what it does ("tower" means change basis to/from the tower field), which round or Key Schedule uses it, and the improvement from the original matrix to the optimized result.

			original		optimized	
name	transform	use	XORs	depth	XORs	depth
Tb1	to tower	R0	24	3	13	6
TeX	affine & from tower	R10	24	3	12	6
TeK	affine in tower	KS	16	3	10	3
Tb2	from tower	KS	22	3	13	3
Tx2	$\times$ 02 but in tower	KS	11	3	10	2

S-box to each byte as well as the MixColumns transformation to the whole column in each regular round of encryption. The original matrix with no optimization (ignoring redundancy) would take 392 XORs with depth 5, but the optimized algorithm needs only 140 XORs with depth 10. Table 1 shows corresponding results for the various byte transformations needed for special rounds and the Key Schedule.

Applying the same method to the standard AES MixColumns matrix has generated some results that are slightly larger but less deep (faster) than the 92-XOR depth 6 result of Maximov[9]; our two new results are 94-XOR depth 5 (see Appendix F) and 97-XOR depth 4 (see Appendix G).

The nonlinear optimization algorithm, suitable for 4-input functions like inverters and S-boxes, is based on the earlier method of Osvik[12] (where it was applied to Serpent), with various refinements. This algorithm performs common subexpression elimination by gradually building an implementation of a given target S-box. It is guided by the limitations of the target (software or hardware) architecture and an evaluation function. The evaluation function calculates an upper limit to the computational depth or area remaining for each of the target outputs if they were to be computed separately.

A collection of resulting S-box implementations are summarized in Table 2, with many of the implementations provided in Appendix C. Those are the results from a series of optimization runs for S-box functions across a set of common CPU characteristics. The total search effort for the results in the table was about one core-month on a Ryzen 9 3950X.

In Table 3 we summarize similar results for S-box functions optimized for hardware implementation using the UMC180 standard cell library[20]. These include both 4-input and 5-input S-boxes. Note that by moving the inversions into their dependent gates, the result for Keccak changes to 19 GE with a depth of only two 2-input gates.

4 Conclusions

We designed an implementation of AES encryption that uses fewer bit operations than all previous implementations. (This only considers two-input logic gates; having more inputs can of course give fewer operations.) Much of the improvement is due to well optimized linear transformations, particularly the 32×32 matrix for the linear part of each round. This

Table 2: **Minimization of S-Box Latency on CPUs**. All the S-boxes have 4 input bits and 4 output bits, and we limit the number of available registers for the S-box computation to 5. The operations are limited to AND/OR/XOR/NOT.
"2AC" and "3AC" refer to instruction sets with 2- or 3-address code. With 2AC we have operations like "a += b", which always overwrite the old value of one input, while with 3AC we have "a = b + c". Common examples are x86 processors (2AC) and various RISC type processors (3AC).
The next line is for the number of boolean instructions that can be executed in parallel in the same clock cycle (execution pipes). Finally, "insn" is the instruction count, while "clk" is the number of clock cycles the S-box takes from start to finish. For single-pipe processors, these are the same, so only one value is listed. For example implementations, see Appendix C.

S-box	2AC							3AC		
	1	2		3		4		1	2	
	insn	insn	clk	insn	clk	insn	clk	insn	insn	clk
GOST pi1	18	19	10	20	7	21	6	15	17	9
GOST pi2	19	20	10	20	8	24	7	16	17	10
GOST pi3	19	19	10	21	7	23	6	17	18	10
GOST pi4	17	17	9	18	7	20	6	15	17	9
GOST pi5	19	20	11	21	8	22	7	16	19	10
GOST pi6	18	19	10	21	7	21	6	17	18	9
GOST pi7	18	18	10	20	7	24	6	15	16	8
GOST pi8	18	18	10	19	7	22	6	16	17	9
LAC/Lblock S0	13	14	7	14	6	15	5	11	12	6
Luffa	15	15	8	15	5	17	5	13	13	7
Minalpher	18	19	10	21	7	21	7	15	17	9
Noekeon	15	15	8	15	6	17	5	12	12	7
Piccolo/Joltik	13	13	7	15	5	15	5	10	10	6
PRESENT	15	16	9	17	7	18	6	14	15	8
Prst	11	11	7	12	5	12	5	8	9	5
QARMA	19	20	10	22	8	22	7	16	18	10
QARMAv2	18	18	10	20	7	21	6	17	18	9
RECTANGLE	14	14	7	14	6	15	5	12	12	7
Serpent S0	16	16	9	17	6	20	6	14	14	8
Serpent S1	16	17	9	17	6	19	6	14	14	8
Serpent S2	15	15	8	15	6	17	5	13	14	7
Serpent S3	17	17	9	17	7	19	6	15	16	8
Serpent S4	17	17	9	17	7	20	6	15	16	8
Serpent S5	17	17	9	17	7	21	6	15	15	8
Serpent S6	17	17	9	17	6	18	6	14	15	8
Serpent S7	18	18	10	20	7	21	6	16	17	9
Skinny S4	13	13	7	14	5	14	5	11	11	6
Twine	18	19	10	20	8	22	7	15	17	9

Table 3: **Minimization of S-Box Area in Hardware**. These S-boxes have either 4 or 5 inputs and outputs. Implementations of all of them are given in Appendix D. Optimization was made for the same subset of gates from the UMC180 standard cell library [20] as was used by [6] and [8].

S-box	GE	Depth	
Twine	21.7	8	[6]
	19.7	5	New
Present	21.3	12	[6]
	20.0	5	New
LAC/LBlock S0	16.3	10	[8]
	16.0	4	New
Keccak	17.7	3	[8]
	17.3	3	New
Ascon	28.7	6	[8]
	27.0	4	New

implementation could be useful for a bit-sliced AES application, where fewer ops translates to greater speed.

Detailed comparisons with some previous results are given in Table 4, in terms of bit operations per byte of a state block. Numbers of different types of ops are shown as well. (AddKey is considered to act on the whole block; Mix acts on a whole column.) This shows that our previous record[3] of 139.5 ops per byte for a whole round of AES was recently broken by using the results of Maximov and Ekdahl [9, 10], at 139 ops per byte. Our new result of 134 ops per byte is a new record; our previous round[3] is 4.1% bigger, while that of [9, 10] is 3.7% bigger.

The last column in the table gives the depth (number of gates in the critical path). This suggests that, for hardware implementations with adequate room and where speed is important, then our new compact approach is not the best choice. Rather, our approach in [3], with a full round only 23 gates deep (or 22, using our new Inv4n), is clearly preferable to the others shown for fast hardware.

As Table 4 also shows, our 4-bit inverters are more compact than in [10]. Moreover, we found that we can reduce the depth of our inverter by adding more ops; our 16-op version reduces the depth from 10 to 4. See Appendix B for the details.

Appendix A gives a demonstration program in C that encrypts one block using our new algorithm. The code works on arrays of bits (each bit stored in a byte) and shows each bit operation in the various functions involved; the result of each round is printed. The user may specify the plaintext and key blocks; if not, the Intermediate Values test case of [11] is used.

Table 4: **AES Results**. The number of bit operations per byte is given for various functions up to full rounds. (RoundX is the last round.) Our integrated approach has no separate S-Box per se. Gate types are grouped with their negations (e.g., XOR and XNOR) for counting purposes; in practice often some gates may be replaced by their negations without changing the function.

function	X(N)OR	(N)AND	(N)OR	NOT	MUX	ops/byte	depth
Canright and Osvik [3], 2009							
AddKey	128					8	1
Inv4	8	5	2			15	5
Inv8	55	27	10			92	15
AffTrans	19.5					17.5	4
Mix	88					22	3
Round0	23					23	5
Round	139.5	27	10			139.5	23
RoundX	113	27	10			113	19
Ueno et al. [17, 18], 2015, 2019							
Inv8	51	38	16	4		109	10
Sbox	94	45	10	10		159	15
Maximov and Ekdahl [9, 10], 2019							
Inv4M[a]	1	1	1		6	9	3
Inv4	6	7	2			15	3
Sbox	69	33	6			108	24
Mix	92					23	6
Round	100	33	6			139	31
current work							
Inv4M[b]	2	1			5	8	4
Inv4n[c]	7	7				14	4
Inv4	8	5	1			14	10
Inv8	55	28	8			91	20
AfMix32	140					35	10
Round0	21					21	7
Round	98	28	8			134	31
RoundX	75	28	8			111	27

[a]This 4-bit inverter is the smallest in [10] but uses 6 MUXs. In the nonlinear functions following it we use the alternate inverter (their section B.2) they provided for comparison using only 2-input gates.

[b]with MUXs, directly compares to [10]

[c]no MUXs, directly compares to [10, B.2]; both this and our Inv4M use the same 4-bit normal tower basis as in [3, 10], but our new AES uses a different basis, as in Inv4.

References

[1] Joan Boyar and René Peralta. C++ implementation of slp algorithm. `http://www.imada.sdu.dk/~joan/xor/Improved2.cc`, 2018.

[2] D. Canright. A very compact S-box for AES. In Josyula R. Rao and Berk Sunar, editors, *CHES 2005*, volume 3659 of *LNCS*, pages 441–455. Springer-Verlag, 2005.

[3] David Canright and Dag Arne Osvik. A more compact AES. In *Selected Areas in Cryptography (SAC2009)*, volume 5867 of *LNCS*, pages 157–169. Springer-Verlag, 2009.

[4] Pawel Chodowiec and Kris Gaj. Very compact FPGA implementation of the AES algorithm. In C. Walter, Ç.K. Koç, and C. Paar, editors, *CHES 2003*, number 2779 in LNCS, pages 319–333. Springer-Verlag, 2003.

[5] Martin Feldhofer, Johannes Wolkerstorfer, and Vincent Rijmen. AES implementation on a grain of sand. In *IEE Proceedings on Information Security*, volume 152, pages 13–20. IEE, 2005.

[6] Jérémy Jean, Thomas Peyrin, and Siang Meng Sim. Optimizing implementations of lightweight building blocks. Cryptology ePrint Archive, Report 2017/101, 2017. `http://eprint.iacr.org/`.

[7] Thorsten Kranz, Gregor Leander, Ko Stoffelen, and Friedrich Wiemer. Shorter linear straight-line programs for MDS matrices. *IACR Transactions on Symmetric Cryptology*, pages 188–211, 2017.

[8] Zhenyu Lu, Weijia Wang, Kai Hu, Yanhong Fan, Lixuan Wu, and Meiqin Wang. Pushing the limits: Searching for implementations with the smallest area for lightweight s-boxes. Cryptology ePrint Archive, Report 2021/1644, 2021. `http://eprint.iacr.org/`.

[9] Alexander Maximov. AES MixColumn with 92 XOR gates. *IACR Cryptol. ePrint Arch.*, 2019:833, 2019.

[10] Alexander Maximov and Patrik Ekdahl. New circuit minimization techniques for smaller and faster AES SBoxes. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 91–125, 2019.

[11] NIST. Specification for the Advanced Encryption Standard (AES), November 2001. FIPS PUB 197.

[12] Dag Arne Osvik. Speeding up Serpent. In *AES Candidate Conference*, pages 317–329, 2000.

[13] C. Paar. *Efficient VLSI Architectures for Bit-Parallel Computation in Galois Fields.* PhD thesis, Institute for Experimental Mathematics, University of Essen, Germany, 1994.

[14] Arash Reyhani-Masoleh, Mostafa Taha, and Doaa Ashmawy. Smashing the implementation records of aes s-box. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 298–336, 2018.

[15] Atri Rudra, Pradeep K. Dubey, Charanjit S. Jutla, Vijay Kumar, Josyula R. Rao, and Pankaj Rohatgi. Efficient Rijndael encryption implementation with composite field arithmetic. In Ç.K. Koç, D. Naccache, and C. Paar, editors, *CHES 2001*, number 2162 in LNCS, pages 171–184. Springer-Verlag, 2001.

[16] A. Satoh, S. Morioka, K. Takano, and Seiji Munetoh. A compact Rijndael hardware architecture with S-box optimization. In C. Boyd, editor, *ASIACRYPT 2001*, number 2248 in LNCS, pages 239–254. Springer-Verlag, 2001.

[17] R Ueno, N Homma, Y Sugawara, Y Nogami, and T Aoki. Highly efficient $GF(2^8)$ inversion circuit based on redundant GF arithmetic and its application to AES design. In *CHES 2015*, volume 9293 of *LNCS*, pages 63–80. Springer-Verlag, 2015.

[18] Rei Ueno, Naofumi Homma, Yasuyuki Nogami, and Takafumi Aoki. Highly efficient $GF(2^8)$ inversion circuit based on hybrid GF representations. *Journal of Cryptographic Engineering*, 9(2):101–113, 2019.

[19] TSMC 0.18 μm process 1.8-volt SAGE-X standard cell library databook, October 2001.

[20] 0.18 μm VIP Standard Cell Library Tape Out Ready, Part Number: UMCL18G212T3, October 2004.

[21] J. Wolkerstorfer, E. Oswald, and M. Lamberger. An ASIC implementation of the AES S-boxes. In B. Preneel, editor, *CT-RSA*, number 2271 in LNCS, pages 67–78. Springer-Verlag, 2002.

A Tower Field AES in C

```
/*
    smallAES.c
    David Canright and Dag Arne Osvik, 2022 January 18
    demonstration of AES algorithm, optimized for size: minimum bit ops per byte
Function   XOR     AND      OR    ops/byte  depth
Mul4        15       9       0      24       5
Norm8       12       5       7      24       3
Inv4         8       5       1      14      10
Inv8        55      28       8      91      20
AfMix32     35       0       0      35      10
Round0      21       0       0      21       7
Round       98      28       8     134      31
RoundX      75      28       8     111      27
AES128     978     280      80    1338     313
    This shows all those bit ops explicitly, and verifies that this works.
    uses tower field basis:  normal 8-bit, polynomial 4-bit, normal 2-bit
    [ 0x1B, 0xFA ] x [ 0x01, 0x5C ] x [ 0xBD, 0xBC ] (in standard AES representation)

    Usage:
        smallAES [ plaintext(hex) [ key(hex) ] ]
        with no arguments, uses PT and Key of FIPS 197, example C.1 (below)
        given hex plaintext (or partial), that overwrites the example PT
        given hex key also (or partial), that overwrites the example Key
        i.e.: smallAES 1feedface deadbeef2
    prints state after each round, in standard basis

Federal Information Processing Standards Publication 197
Appendix C  Example Vectors, C.1 AES-128
PLAINTEXT:      00112233445566778899aabbccddeeff
KEY:            000102030405060708090a0b0c0d0e0f
round[ 0].input 00112233445566778899aabbccddeeff
round[ 1].start 00102030405060708090a0b0c0d0e0f0
round[ 2].start 89d810e8855ace682d1843d8cb128fe4
round[ 3].start 4915598f55e5d7a0daca94fa1f0a63f7
round[ 4].start fa636a2825b339c940668a3157244d17
round[ 5].start 247240236966b3fa6ed2753288425b6c
round[ 6].start c81677bc9b7ac93b25027992b0261996
round[ 7].start c62fe109f75eedc3cc79395d84f9cf5d
round[ 8].start d1876c0f79c4300ab45594add66ff41f
round[ 9].start fde3bad205e5d0d73547964ef1fe37f1
round[10].start bd6e7c3df2b5779e0b61216e8b10b689
round[10].output 69c4e0d86a7b0430d8cdb78070b4c55a
*/

#include <stdio.h>
#include <string.h>
```

```
#define rc(r,c) (((r)%4+((c)%4)*4)*8)
typedef unsigned char byte;
byte State[12][128], Key[12][128], b[128*2];

void KeySchedule(void) ;
void Round0(int n) ;
void Round(int n) ;
void RoundX(int n) ;
void PrintState(byte State[128], int round, char *tag, byte T(byte) ) ;
void i2b(int i, byte b[], int n) ;
byte X(byte x) ;
byte I(byte x) ;

int main(int argc, char *argv[]) {
    int i, j, n;
    byte t[128];

//  set up PT, Key for intermediate value tests
    for (i = 0; i < 16; i++) {
        i2b( i*0x11, State[0] + i*8, 8);
        i2b( i, Key[0] + i*8, 8);
    }
    if (argc > 1) {     // modify PT if given
    if ((n = (strlen(argv[1]) + 1) / 2) > 16) n = 16;
    for (i = 0; i < n; i++) {
        if (sscanf(argv[1] + 2 * i, "%2x", &j) < 1) break;
        i2b( j, State[0] + i*8, 8);
    }}
    if (argc > 2) {     // modify Key if given
    if ((n = (strlen(argv[1]) + 1) / 2) > 16) n = 16;
    for (i = 0; i < n; i++) {
        if (sscanf(argv[2] + 2 * i, "%2x", &j) < 1) break;
        i2b( j, Key[0] + i*8, 8);
    }}

    KeySchedule();
//  show round results (in standard basis)
    PrintState(State[0], 0, "PT", I);
    PrintState(Key[0], 0, "Key", X);
    Round0(0);
    PrintState(State[0], 0, "St", X);
    for (j=1;j<10;j++) {
        Round(j);
        PrintState(State[j], j, "St", X);
        }
    RoundX(j);
    PrintState(State[j], j, "CT", I);
    return (0);
}

// i2b(i,b,n) puts n bits of i into array b (utility)
void i2b(int i, byte b[], int n){
    int j;
    for (j=0;j<n;j++) b[j] = (i >> j) & 1;
}

// b2i(b,n) returns an integer from n bits of array b (utility)
unsigned int b2i(byte b[], int n){
    unsigned int i=0,j;
    for (j=0;j<n;j++)  i |= (b[j] & 1) << j;
    return i;
}

void PrintState(byte State[128], int round, char *tag, byte T(byte) ) {
    int i;

    printf("%-3s%3d = ", tag, round);
    for (i = 0; i < 16; i++) printf("%02x ", T(b2i( State + i*8, 8)) );
    printf("\n");
}

void Tb1(byte x[], byte y[]) ;
void Tb2(byte x[], byte y[]) ;

// X converts a byte from tower basis to standard (utility)
byte X(byte x) {
    byte t[8], y[8];

    i2b(x,t,8);
    Tb2(t, y);
    return b2i(y,8);
}
// Xi converts a byte to tower basis from standard (utility)
byte Xi(byte x) {
    byte t[8], y[8];

    i2b(x,t,8);
    Tb1(t, y);
    return b2i(y,8);
}
// I does nothing to a byte (utility)
byte I(byte x) {
    return x;
```

```
}

// Add adds 2 bitstrings of length n: n ^ ; d = 1
void Add(byte x[], byte y[], byte z[], int n) {
    int i;
    for (i=0;i<n;i++) z[i] = x[i] ^ y[i];
}
// AddByte adds 2 bytes: 8 ^ ; d = 1
void AddByte(byte x[], byte y[], byte z[]) {
    Add(x, y, z, 8);
}
// AddBlock adds 2 blocks of 16 bytes: 128 ^ ; d = 1
void AddBlock(byte x[], byte y[], byte z[]) {
    Add(x, y, z, 128);
}

// MulSum4 finds the bit sums for Mul4 : 5 ^ ; d = 2
void MulSum4(byte x[], byte s[]) {
    byte i;

    for (i=0;i<4;i++) s[i] = x[i];
    s[4] = x[0] ^ x[1];     // (a+b)
    s[5] = x[0] ^ x[2];     // (a+c)
    s[6] = x[1] ^ x[3];     // (b+d)
    s[7] = x[2] ^ x[3];     // (c+d)

    s[8] = s[4] ^ s[7];     // (a+b+c+d)
}

// Mul4 multiplies 2 4-bit values with bit sums in tower field : 10 ^  9 & ; d = 3
void Mul4(byte x[], byte y[], byte z[]) {
    byte t[16];

    t[0] = x[0] & y[0];     // a*e
    t[1] = x[1] & y[1];     // b*f
    t[2] = x[2] & y[2];     // c*g
    t[3] = x[3] & y[3];     // d*h
    t[4] = x[4] & y[4];     // (a+b)*(e+f)
    t[5] = x[5] & y[5];     // (a+c)*(e+g)
    t[6] = x[6] & y[6];     // (b+d)*(f+h)
    t[7] = x[7] & y[7];     // (c+d)*(g+h)
    t[8] = x[8] & y[8];     // (a+b+c+d)*(e+f+g+h)

    t[ 9] = t[0] ^ t[4];    // a*e + (a+b)*(e+f)
    t[10] = t[1] ^ t[4];    // b*f + (a+b)*(e+f)
    t[11] = t[2] ^ t[3];    // c*g + d*h
    t[12] = t[2] ^ t[7];    // c*g + (c+d)*(g+h)
    t[13] = t[5] ^ t[8];    // (a+c)*(e+g) + (a+b+c+d)*(e+f+g+h)
    t[14] = t[6] ^ t[8];    // (b+d)*(f+h) + (a+b+c+d)*(e+f+g+h)

    z[0] = t[ 9] ^ t[11];   // a*e + c*g + d*h + (a+b)*(e+f)
    z[1] = t[10] ^ t[12];   // b*f + c*g + (a+b)*(e+f) + (c+d)*(g+h)
    z[2] = t[ 9] ^ t[13];   // a*e + (a+b)*(e+f) + (a+c)*(e+g) + (a+b+c+d)*(e+f+g+h)
    z[3] = t[10] ^ t[14];   // b*f + (a+b)*(e+f) + (b+d)*(f+h) + (a+b+c+d)*(e+f+g+h)
}

// Norm8 finds the norm of a byte with bit sums in tower field : 12 ^ 5 & 7 | ; d = 3
void Norm8(byte a[], byte b[], byte y[]) {
    byte t[32];

    t[0] = a[0] | b[0];     // a|e
    t[1] = a[1] | b[1];     // b|f
    t[2] = a[2] & b[2];     // c&g
    t[3] = a[2] | b[2];     // c|g
    t[4] = a[3] & b[3];     // d&h
    t[5] = a[5] & b[5];     // (a^c)&(e^g)
    t[6] = a[5] | b[5];     // (a^c)|(e^g)
    t[7] = a[4] | b[4];     // (a^b)|(e^f)
    t[8] = a[6] & b[6];     // (b^d)&(f^h)
    t[9] = a[6] | b[6];     // (b^d)|(f^h)
    t[10] = a[7] | b[7];    // (c^d)|(g^h)
    t[11] = a[8] & b[8];    // (a^b^c^d)&(e^f^g^h)

    t[12] = t[0] ^ t[1];    // a|e ^ b|f
    t[13] = t[5] ^ t[9];    // (a^c)&(e^g) ^ (b^d)|(f^h)
    t[14] = t[7] ^ t[6];    // (a^b)|(e^f) ^ (a^c)|(e^g)
    t[15] = t[0] ^ t[11];   // a|e ^ (a^b^c^d)&(e^f^g^h)
    t[16] = t[4] ^ t[5];    // d&h ^ (a^c)&(e^g)
    t[17] = t[3] ^ t[11];   // c|g ^ (a^b^c^d)&(e^f^g^h)
    t[18] = t[8] ^ t[10];   // (b^d)&(f^h) ^ (c^d)|(g^h)
    t[19] = t[2] ^ t[11];   // c&g ^ (a^b^c^d)&(e^f^g^h)

    y[0] = t[12] ^ t[13];   // a|e ^ b|f ^ (a^c)&(e^g) ^ (b^d)|(f^h)
    y[1] = t[14] ^ t[15];   // a|e ^ (a^b)|(e^f) ^ (a^c)|(e^g) ^ (a^b^c^d)&(e^f^g^h)
    y[2] = t[16] ^ t[17];   // c|g ^ d&h ^ (a^c)&(e^g) ^ (a^b^c^d)&(e^f^g^h)
    y[3] = t[18] ^ t[19];   // c&g ^ (b^d)&(f^h) ^ (c^d)|(g^h) ^ (a^b^c^d)&(e^f^g^h)
}

// Inv4 finds the inverse of a 4-bit value in the tower field : 8 ^ 5 & 1 | ; d = 10
// optimized by Dag Arne
void Inv4(byte x[], byte y[]) {
    byte r[8];

    r[4] = x[1] & x[3];
    r[5] = x[2] ^ r[4];
    r[2] = x[1] & r[5];   r[1] = x[1] ^ r[5];
    r[0] = x[0] ^ r[2];
    r[4] = r[0] & x[3];
    y[2] = r[4] ^ r[5];
    r[4] = r[0] & y[2];   r[6] = r[1] & y[2];
    r[2] = r[2] ^ r[4];   y[0] = r[1] ^ r[4];   y[3] = r[6] ^ x[3];
    r[2] = y[3] | r[2];
    y[1] = r[0] ^ r[2];
}

// Inv8 finds the inverse of a 8-bit value in the tower field : 55 ^ 28 & 8 | ; d = 20
void Inv8(byte x[], byte y[]) {
    byte n[4], a[10], b[10], c[10];

    MulSum4(x, a);
    MulSum4(x+4, b);
    Norm8(a, b, n);

    Inv4(n, c);

    MulSum4(c, c);
    Mul4(b, c, y);
    Mul4(a, c, y+4);
}

/* Linear Transformations, optimized by Dag Arne
// little-endian row list, nibbles (&bits), so 25 = 01001010
D1 22 AA B5 F1 E8 87 55:    b1   13/6, 14/4, 15/3
C0 FC BE A6 01 94 77 8E:    A.b2 12/6, 14/4
D9 68 40 EC 25 6A 44 08:    e1   10/3
A4 11 73 EA 2D 31 F2 8A:    b2   13/3
02 03 08 0C 2C 34 85 CA:    x2   10/2
*/

// Tb1 does initial byte transformation to tower basis : 13 ^ ; d = 6
void Tb1(byte x[], byte y[]) {
    byte t[16];

    t[4] = x[6] ^ x[4];     // 05 = 04 ^ 01
    y[1] = x[1] ^ x[5];     // 22 = 20 ^ 02
    t[6] = x[3] ^ x[1];     // a0 = 80 ^ 20

    t[7] = t[6]  ^ x[7];    // a8 = a0 ^ 08
    t[8] = t[6]  ^ t[4] ;   // a5 = a0 ^ 05

    y[2] = t[7]  ^ x[5];    // aa = a8 ^ 02
    y[5] = t[7]  ^ x[2];    // e8 = a8 ^ 40
    y[3] = t[8]  ^ x[0];    // b5 = a5 ^ 10
    y[6] = t[8]  ^ y[1] ;   // 87 = a5 ^ 22

    t[13] = y[3]  ^ x[2];   // f5 = b5 ^ 40

    y[4] = t[13] ^ x[6];    // f1 = f5 ^ 04
    y[7] = t[13] ^ t[6] ;   // 55 = f5 ^ a0

    y[0] = y[4]  ^ x[1];    // d1 = f1 ^ 20
}

// TeX does final byte transformation affine & from tower basis : 12 ^ ; d = 6
void TeX(byte x[], byte y[]) {
    byte t[16];

    y[4] = x[4];            // 01 = x[4]

    t[4] = x[2] ^ x[5];     // 42 = 40 ^ 02
    t[5] = x[3] ^ x[6];     // 84 = 80 ^ 04
    y[0] = x[3] ^ x[2] ^ 1; // c0 = 80 ^ 40 // XNOR

    t[7] = t[5] ^ x[5];     // 86 = 84 ^ 02
    y[5] = t[5] ^ x[0] ^ 1; // 94 = 84 ^ 10 // XNOR

    y[7] = t[7] ^ x[7];     // 8e = 86 ^ 08
    y[3] = t[7] ^ x[1];     // a6 = 86 ^ 20

    t[11] = y[3] ^ x[0];    // b6 = a6 ^ 10

    t[12] = t[11] ^ x[4];   // b7 = b6 ^ 01
    y[2] = t[11] ^ x[7];    // be = b6 ^ 08

    y[6] = t[12] ^ y[0];    // 77 = b7 ^ c0
    y[1] = y[2] ^ t[4] ^ 1; // fc = be ^ 42 // XNOR
}

// TeK does affine trans w/ const 0x29 in tower basis for Key Sch. : 10 ^ ; d = 3
void TeK(byte x[], byte y[]) {
    byte t[16];

    y[2] = x[2];            // 40 = x[2]
    y[7] = x[7];            // 08 = x[7]

    t[4] = x[0] ^ x[4];     // 11 = 10 ^ 01
    t[5] = x[1] ^ x[6];     // 24 = 20 ^ 04
    y[6] = x[2] ^ x[6];     // 44 = 40 ^ 04
    t[7] = x[2] ^ x[7];     // 48 = 40 ^ 08

    y[4] = t[5]  ^ x[4];    // 25 = 24 ^ 01
    y[1] = t[7]  ^ x[1];    // 68 = 48 ^ 20
    t[10] = t[7]  ^ x[3];   // c8 = 48 ^ 80

    y[5] = y[1]  ^ x[5] ^ 1;// 6a = 68 ^ 02     XNOR
    y[0] = t[10] ^ t[4] ^ 1;// d9 = c8 ^ 11     XNOR
    y[3] = t[10] ^ t[5] ^ 1;// ec = c8 ^ 24     XNOR
}

// Tb2 does reverse byte transformation from tower basis : 13 ^ ; d = 3
void Tb2(byte x[], byte y[]) {
    byte t[16];

    y[1] = x[0] ^ x[4];     // 11 = 10 ^ 01
    t[5] = x[2] ^ x[5];     // 42 = 40 ^ 02
    t[6] = x[3] ^ x[7];     // 88 = 80 ^ 08
    t[7] = x[3] ^ x[1];     // a0 = 80 ^ 20

    y[5] = y[1]  ^ x[1];    // 31 = 11 ^ 20
    t[9] = t[6]  ^ x[4];    // 89 = 88 ^ 01
    y[7] = t[6]  ^ x[5];    // 8a = 88 ^ 02
    y[0] = t[7]  ^ x[6];    // a4 = a0 ^ 04
    t[12] = t[7]  ^ t[5] ;  // e2 = a0 ^ 42

    y[2] = y[5]  ^ t[5] ;   // 73 = 31 ^ 42
    y[4] = y[0]  ^ t[9] ;   // 2d = a4 ^ 89
    y[3] = t[12] ^ x[7];    // ea = e2 ^ 08
    y[6] = t[12] ^ x[0];    // f2 = e2 ^ 10
}

// Tx2 multiplies by 0x02 (std basis) in KeySched in tower basis : 10 ^ ; d = 2
void Tx2(byte x[], byte y[]) {
    byte t[16];

    y[0] = x[5];            // 02 = x[5]
    y[2] = x[7];            // 08 = x[7]

    y[1] = x[5] ^ x[4];     // 03 = 02 ^ 01
    t[5] = x[6] ^ x[4];     // 05 = 04 ^ 01
    t[6] = x[7] ^ x[5];     // 0a = 08 ^ 02
    y[3] = x[7] ^ x[6];     // 0c = 08 ^ 04
    t[8] = x[0] ^ x[6];     // 14 = 10 ^ 04
    t[9] = x[3] ^ x[2];     // c0 = 80 ^ 40

    y[6] = t[5]  ^ x[3];    // 85 = 05 ^ 80
    y[4] = y[3]  ^ x[1];    // 2c = 0c ^ 20
    y[5] = t[8]  ^ x[1];    // 34 = 14 ^ 20
    y[7] = t[9]  ^ t[6] ;   // ca = c0 ^ 0a
}

// AfMix32 does affine trans & Mix for single column : 140 ^ ; d = 10
/* Dag Arne's optimized 32x32 matrix for affine & scaling & Mix
// little-endian row list, nibbles (&bits), so 25 = 01001010
0x6ab3d9d9 0x4f276868 0x08484040 0x4ca0ecec 0x24012525 0xf59f6a6a 0x8dc94444 0xcec60808
0xd96ab3d9 0x684f2768 0x40084840 0xec4ca0ec 0x25240125 0x6af59f6a 0x448dc944 0x08cec608
0xd9d96ab3 0x68684f27 0x40400848 0xecec4ca0 0x25252401 0x6a6af59f 0x44448dc9 0x0808cec6
0xb3d9d96a 0x2768684f 0x48404008 0xa0ecec4c 0x01252524 0x9f6a6af5 0xc944448d 0xc60808ce
*/
void AfMix32(byte x[], byte y[]) {
    byte r[135+1];

    r[  0] = x[ 23] ^ x[ 26];       // 0x00000840 = 0x00000800 ^ 0x00000040
    r[  1] = x[ 19] ^ x[ 20];       // 0x00008100 = 0x00008000 ^ 0x00000100
    r[  2] = x[ 13] ^ x[ 21];       // 0x00020200 = 0x00020000 ^ 0x00000200
    r[  3] = x[ 14] ^ x[ 22];       // 0x00040400 = 0x00040000 ^ 0x00000400
    r[  4] = x[ 14] ^ x[ 13];       // 0x00060000 = 0x00040000 ^ 0x00020000
    r[  5] = x[ 15] ^ x[ 26];       // 0x00080040 = 0x00080000 ^ 0x00000040
    r[  6] = x[  9] ^ x[ 17];       // 0x00202000 = 0x00200000 ^ 0x00002000
    r[  7] = x[ 11] ^ x[ 19];       // 0x00808000 = 0x00800000 ^ 0x00008000
    r[  8] = x[ 11] ^ x[ 12];       // 0x00810000 = 0x00800000 ^ 0x00010000
    r[  9] = x[  4] ^ x[ 12];       // 0x01010000 = 0x01000000 ^ 0x00010000
    r[ 10] = x[  6] ^ x[ 30];       // 0x04000004 = 0x04000000 ^ 0x00000004
    r[ 11] = x[  7] ^ x[ 10];       // 0x08400000 = 0x08000000 ^ 0x00400000
    r[ 12] = x[  1] ^ x[ 25];       // 0x20000020 = 0x20000000 ^ 0x00000020
    r[ 13] = x[  2] ^ x[ 31];       // 0x40000008 = 0x40000000 ^ 0x00000008
    r[ 14] = x[  3] ^ x[ 29];       // 0x80000002 = 0x80000000 ^ 0x00000002
    r[ 15] = x[  3] ^ x[  4];       // 0x81000000 = 0x80000000 ^ 0x01000000

    r[ 16] = r[  0] ^ x[ 18];       // 0x00004840 = 0x00000840 ^ 0x00004000
    r[ 17] = r[  0] ^ x[  7];       // 0x08000840 = 0x00000840 ^ 0x08000000
    r[ 18] = r[  1] ^ x[ 16];       // 0x00009100 = 0x00008100 ^ 0x00001000
    r[ 19] = r[  3] ^ x[  6];       // 0x04040400 = 0x00040400 ^ 0x04000000
    r[ 20] = r[  3] ^ r[  2];       // 0x00060600 = 0x00040400 ^ 0x00020200
    r[ 21] = r[  4] ^ x[ 30];       // 0x00060004 = 0x00060000 ^ 0x00000004
    r[ 22] = r[  6] ^ x[  1];       // 0x20202000 = 0x00202000 ^ 0x20000000
    r[ 23] = r[  6] ^ r[  5];       // 0x00282040 = 0x00202000 ^ 0x00080040
    r[ 24] = r[  8] ^ x[  8];       // 0x00910000 = 0x00810000 ^ 0x00100000
    r[ 25] = r[ 10] ^ x[ 14];       // 0x04040004 = 0x04000004 ^ 0x00040000
    r[ 26] = r[ 10] ^ r[  6];       // 0x04202004 = 0x04000004 ^ 0x00202000
    r[ 27] = r[ 11] ^ x[ 18];       // 0x08404000 = 0x08400000 ^ 0x00004000
    r[ 28] = r[ 12] ^ x[  9];       // 0x20200020 = 0x20000020 ^ 0x00200000
    r[ 29] = r[ 13] ^ r[ 11];       // 0x48400008 = 0x40000008 ^ 0x08400000
```

```
r[ 30] = r[ 14] ^ r[  7];        // 0x80808002 = 0x80000002 ^ 0x00808000
r[ 31] = r[ 15] ^ x[  0];        // 0x91000000 = 0x81000000 ^ 0x10000000
r[ 32] = r[ 15] ^ r[ 10];        // 0x85000004 = 0x81000000 ^ 0x04000004

r[ 33] = r[ 16] ^ x[ 15];        // 0x00084840 = 0x00004840 ^ 0x00080000
r[ 34] = r[ 18] ^ x[  5];        // 0x02009100 = 0x00009100 ^ 0x02000000
r[ 35] = r[ 19] ^ r[  9];        // 0x05050400 = 0x04040400 ^ 0x01010000
r[ 36] = r[ 21] ^ x[ 27];        // 0x00060084 = 0x00060004 ^ 0x00000080
r[ 37] = r[ 25] ^ r[  1];        // 0x04048104 = 0x04040004 ^ 0x00008100
r[ 38] = r[ 26] ^ x[ 25];        // 0x04202024 = 0x04202004 ^ 0x00000020
y[  2] = r[ 27] ^ r[  5];        // 0x08484040 = 0x08404000 ^ 0x00080040
y[ 26] = r[ 27] ^ r[ 13];        // 0x48404008 = 0x08404000 ^ 0x40000008
r[ 41] = r[ 28] ^ x[ 20];        // 0x20200120 = 0x20200020 ^ 0x00000100
r[ 42] = r[ 28] ^ x[ 13];        // 0x20220020 = 0x20200020 ^ 0x00020000
r[ 43] = r[ 29] ^ x[ 27];        // 0x48400088 = 0x48400008 ^ 0x00000080
y[ 18] = r[ 29] ^ r[ 17];        // 0x40400848 = 0x48400008 ^ 0x08000840
r[ 45] = r[ 29] ^ r[ 20];        // 0x48460608 = 0x48400008 ^ 0x00060600
r[ 46] = r[ 31] ^ r[  2];        // 0x91020200 = 0x91000000 ^ 0x00020200
r[ 47] = r[ 32] ^ r[  8];        // 0x85810004 = 0x85000004 ^ 0x00810000

r[ 48] = r[ 33] ^ x[ 29];        // 0x00084842 = 0x00084840 ^ 0x00000002
y[ 10] = r[ 33] ^ x[  2];        // 0x40084840 = 0x00084840 ^ 0x40000000
r[ 50] = r[ 34] ^ r[ 22];        // 0x2220b100 = 0x02009100 ^ 0x20202000
r[ 51] = r[ 34] ^ r[ 24];        // 0x02919100 = 0x02009100 ^ 0x00910000
r[ 52] = r[ 35] ^ r[ 22];        // 0x25252400 = 0x05050400 ^ 0x20202000
r[ 53] = r[ 38] ^ x[ 31];        // 0x0420202c = 0x04202024 ^ 0x00000008
r[ 54] = r[ 38] ^ x[ 10];        // 0x04602024 = 0x04202024 ^ 0x00400000
r[ 55] = r[ 38] ^ r[ 35];        // 0x01252424 = 0x04202024 ^ 0x05050400
r[ 56] = r[ 41] ^ x[ 28];        // 0x20200121 = 0x20200120 ^ 0x00000001
r[ 57] = r[ 41] ^ r[ 23];        // 0x20082160 = 0x20200120 ^ 0x00282040
r[ 58] = r[ 43] ^ x[ 28];        // 0x48400089 = 0x48400088 ^ 0x00000001
r[ 59] = r[ 43] ^ x[  3];        // 0xc8400088 = 0x48400088 ^ 0x80000000
r[ 60] = r[ 45] ^ r[  0];        // 0x48460e48 = 0x48460608 ^ 0x00000840
r[ 61] = r[ 45] ^ r[ 21];        // 0x4840060c = 0x48460608 ^ 0x00060004
r[ 62] = r[ 47] ^ y[  2];        // 0x8dc94044 = 0x85810004 ^ 0x08484040

r[ 63] = r[ 48] ^ x[ 31];        // 0x0008484a = 0x00084842 ^ 0x00000008
r[ 64] = r[ 48] ^ r[ 27];        // 0x08480842 = 0x00084842 ^ 0x08404000
r[ 65] = y[ 10] ^ x[  5];        // 0x42084840 = 0x40084840 ^ 0x02000000
r[ 66] = y[ 10] ^ r[  8];        // 0x40894840 = 0x40084840 ^ 0x00810000
r[ 67] = r[ 51] ^ r[ 42];        // 0x22b39120 = 0x02919100 ^ 0x20220020
y[ 20] = r[ 52] ^ x[ 28];        // 0x25252401 = 0x25252400 ^ 0x00000001
r[ 69] = r[ 54] ^ r[ 16];        // 0x04606864 = 0x04602024 ^ 0x00004840
y[ 28] = r[ 55] ^ x[ 20];        // 0x01252524 = 0x01252424 ^ 0x00000100
r[ 71] = r[ 56] ^ x[ 12];        // 0x20210121 = 0x20200121 ^ 0x00010000
r[ 72] = r[ 56] ^ r[ 25];        // 0x24240125 = 0x20200121 ^ 0x04040004
r[ 73] = r[ 57] ^ x[ 12];        // 0x20092160 = 0x20082160 ^ 0x00010000
```

```
r[ 74] = r[ 57] ^ r[ 37];       // 0x240ca064 = 0x20082160 ^ 0x04048104
r[ 75] = r[ 58] ^ x[ 24];       // 0x48400099 = 0x48400089 ^ 0x00000010
r[ 76] = r[ 58] ^ r[ 19];       // 0x4c440489 = 0x48400089 ^ 0x04040400
r[ 77] = r[ 59] ^ r[ 10];       // 0xcc40008c = 0xc8400088 ^ 0x04000004
r[ 78] = r[ 60] ^ y[ 10];       // 0x084e4608 = 0x48460e48 ^ 0x40084840
r[ 79] = r[ 61] ^ r[ 48];       // 0x48484e4e = 0x4840060c ^ 0x00084842
y[  6] = r[ 62] ^ x[ 22];       // 0x8dc94444 = 0x8dc94044 ^ 0x00000400

r[ 81] = r[ 63] ^ x[ 26];       // 0x0008480a = 0x0008484a ^ 0x00000040
r[ 82] = r[ 63] ^ r[ 12];       // 0x2008486a = 0x0008484a ^ 0x20000020
r[ 83] = r[ 65] ^ r[ 53];       // 0x4628686c = 0x42084840 ^ 0x0420202c
y[ 14] = r[ 66] ^ r[ 37];       // 0x448dc944 = 0x40894840 ^ 0x04048104
r[ 85] = r[ 67] ^ r[ 26];       // 0x2693b124 = 0x22b39120 ^ 0x04202004
r[ 86] = r[ 67] ^ r[ 60];       // 0x6af59f68 = 0x22b39120 ^ 0x48460e48
r[ 87] = r[ 69] ^ r[ 43];       // 0x4c2068ec = 0x04606864 ^ 0x48400088
r[ 88] = r[ 71] ^ r[ 26];       // 0x24012125 = 0x20210121 ^ 0x04202004
y[ 12] = r[ 72] ^ x[  4];       // 0x25240125 = 0x24240125 ^ 0x01000000
y[  9] = r[ 73] ^ r[ 45];       // 0x684f2768 = 0x20092160 ^ 0x48460608
y[ 11] = r[ 74] ^ r[ 59] ^ 1;   // 0xec4ca0ec = 0x240ca064 ^ 0xc8400088 XNOR
r[ 92] = r[ 75] ^ r[ 46];       // 0xd9420299 = 0x48400099 ^ 0x91020200
r[ 93] = r[ 76] ^ r[  1];       // 0x4c448589 = 0x4c440489 ^ 0x00008100
r[ 94] = r[ 76] ^ r[ 32];       // 0xc944048d = 0x4c440489 ^ 0x85000004
r[ 95] = r[ 77] ^ x[  5];       // 0xce40008c = 0xcc40008c ^ 0x02000000
y[ 15] = r[ 78] ^ r[  7];       // 0x08cec608 = 0x084e4608 ^ 0x00808000
r[ 97] = r[ 79] ^ x[  7];       // 0x40484e4e = 0x48484e4e ^ 0x08000000
r[ 98] = r[ 79] ^ r[  3];       // 0x484c4a4e = 0x48484e4e ^ 0x00040400

r[ 99] = r[ 81] ^ r[ 56];       // 0x2028492b = 0x0008480a ^ 0x20200121
r[100] = r[ 81] ^ r[ 67];       // 0x22bbd92a = 0x0008480a ^ 0x22b39120
r[101] = r[ 82] ^ r[ 19];       // 0x240c4c6a = 0x2008486a ^ 0x04040400
r[102] = r[ 82] ^ r[ 31];       // 0xb108486a = 0x2008486a ^ 0x91000000
r[103] = r[ 85] ^ r[ 75];       // 0x6ed3b1bd = 0x2693b124 ^ 0x48400099
y[ 13] = r[ 86] ^ x[ 29] ^ 1;   // 0x6af59f6a = 0x6af59f68 ^ 0x00000002 XNOR
r[105] = r[ 87] ^ r[  7];       // 0x4ca0e8ec = 0x4c2068ec ^ 0x00808000
y[  4] = r[ 88] ^ x[ 22];       // 0x24012525 = 0x24012125 ^ 0x00000400
r[107] = r[ 92] ^ r[ 18];       // 0xd9429399 = 0xd9420299 ^ 0x00009100
r[108] = r[ 92] ^ r[ 50];       // 0xfb62b399 = 0xd9420299 ^ 0x2220b100
y[ 29] = r[ 92] ^ r[ 83] ^ 1;   // 0x9f6a6af5 = 0xd9420299 ^ 0x4628686c XNOR
y[ 22] = r[ 93] ^ r[ 17];       // 0x44448dc9 = 0x4c448589 ^ 0x08000840
r[111] = r[ 93] ^ r[ 30];       // 0xccc4058b = 0x4c448589 ^ 0x80808002
y[ 30] = r[ 94] ^ x[ 18];       // 0xc944448d = 0xc944048d ^ 0x00004000
r[113] = r[ 95] ^ r[ 36];       // 0xce460008 = 0xce40008c ^ 0x00060084
y[ 31] = r[ 95] ^ r[ 64];       // 0xc60808ce = 0xce40008c ^ 0x08480842
r[115] = r[ 97] ^ r[ 43];       // 0x08084ec6 = 0x40484e4e ^ 0x48400088
r[116] = r[ 98] ^ r[ 11];       // 0x400c4a4e = 0x484c4a4e ^ 0x08400000
r[117] = r[ 98] ^ r[ 60];       // 0x000a4406 = 0x484c4a4e ^ 0x48460e48
```

```
    y[ 17] = r[ 99] ^ r[ 61];       // 0x68684f27 = 0x2028492b ^ 0x4840060c
    r[119] = r[101] ^ r[ 54];       // 0x206c6c4e = 0x240c4c6a ^ 0x04602024
    r[120] = r[102] ^ r[ 51];       // 0xb399d96a = 0xb108486a ^ 0x02919100
    y[  0] = r[103] ^ r[ 69] ^ 1;   // 0x6ab3d9d9 = 0x6ed3b1bd ^ 0x04606864 XNOR
    y[  3] = r[105] ^ x[ 22] ^ 1;   // 0x4ca0ecec = 0x4ca0e8ec ^ 0x00000400 XNOR
    y[  8] = r[107] ^ r[ 23] ^ 1;   // 0xd96ab3d9 = 0xd9429399 ^ 0x00282040 XNOR
    y[ 16] = r[108] ^ r[100] ^ 1;   // 0xd9d96ab3 = 0xfb62b399 ^ 0x22bbd92a XNOR
    y[ 19] = r[111] ^ r[ 99] ^ 1;   // 0xecec4ca0 = 0xccc4058b ^ 0x2028492b XNOR
    r[126] = r[113] ^ x[ 11];       // 0xcec60008 = 0xce460008 ^ 0x00800000
    y[ 23] = r[115] ^ x[ 19];       // 0x0808cec6 = 0x08084ec6 ^ 0x00008000
    r[128] = r[116] ^ r[ 85];       // 0x669ffb6a = 0x400c4a4e ^ 0x2693b124
    r[129] = r[117] ^ r[ 75];       // 0x484a449f = 0x000a4406 ^ 0x48400099

    y[ 27] = r[119] ^ r[ 30] ^ 1;   // 0xa0ecec4c = 0x206c6c4e ^ 0x80808002 XNOR
    r[131] = r[119] ^ r[ 95];       // 0xee2c6cc2 = 0x206c6c4e ^ 0xce40008c
    y[ 24] = r[120] ^ x[ 10] ^ 1;   // 0xb3d9d96a = 0xb399d96a ^ 0x00400000 XNOR
    y[  7] = r[126] ^ x[ 23];       // 0xcec60808 = 0xcec60008 ^ 0x00000800
    r[134] = r[126] ^ r[ 47];       // 0x4b47000c = 0xcec60008 ^ 0x85810004
    r[135] = r[128] ^ r[ 31];       // 0xf79ffb6a = 0x669ffb6a ^ 0x91000000
    y[ 21] = r[129] ^ r[ 50] ^ 1;   // 0x6a6af59f = 0x484a449f ^ 0x2220b100 XNOR

    y[ 25] = r[131] ^ r[ 94];       // 0x2768684f = 0xee2c6cc2 ^ 0xc944048d
    y[  1] = r[134] ^ r[ 69];       // 0x4f276868 = 0x4b47000c ^ 0x04606864
    y[  5] = r[135] ^ r[ 34] ^ 1;   // 0xf59f6a6a = 0xf79ffb6a ^ 0x02009100 XNOR
// Depth 10
}

// ShiftInv does ShiftRows & Inv step of Sbox, no affine
// same as Inv8 per byte : 55 ^ 28 & 8 | ; d = 20
void ShiftInv(byte x[], byte b[]) {
    int r,c;

    for (c=0;c<4;c++)
    for (r=0;r<4;r++){
        Inv8( x + rc(r,r+c), b + rc(r,c) ); // ShiftRows too
        }
}

// SclMix does scaling and MixColumns from unscaled buffer
// same as AfMix32 per col : 140 ^ ; d = 10
void SclMix(byte b[], byte y[]) {
    int r,c;

    for (c=0;c<4;c++){
        AfMix32( b + rc(0,c) , y + rc(0,c) );
        }
}
```

```
// Round does one round from State[n-1] to State[n] : 98 ^ 28 & 8 | ; d = 31
void Round(int n) {
    ShiftInv( State[n-1], b );
    SclMix( b, State[n] );
    AddBlock(Key[n], State[n], State[n]);
}

// Round0 does round #0, just adds Key : 21 ^ ; d = 7
void Round0(int n) {
    int i;

    memcpy(b, State[0], 16*8);      // cannot do "in place"
    for (i=0;i<16;i++) {    // change basis
        Tb1( b + i*8, State[0]+i*8);
        }
    AddBlock(Key[0], State[0], State[0]);  // from KeySched in tower basis
}

// RoundX does round 10, no Mix : 75 ^ 28 & 8 | ; d = 27
void RoundX(int n) {
    byte s[8], *x,*y,*z;
    int r,c;

    for (c=0;c<4;c++)
    for (r=0;r<4;r++){
        Inv8( State[n-1] + rc(r,r+c), s );  // ShiftRows too
        TeX( s, State[n] + rc(r,c) );       // affine & to std basis
        }
    AddBlock(Key[n], State[n], State[n]);
}

// SboxK does single byte Sbox for Key Schedule : 65 ^ 28 & 8 | ; d = 23
void SboxK(byte x[], byte b[]) {
    byte s[8];

    Inv8( x, s );
    TeK( s, b );    // affine
}

// KeySchedule does key schedule  (not optimized)
void KeySchedule(void) {
    int r, c, n;
    byte t[16], rcon = 1;
    byte rconb[8];

    memcpy(b, Key[0], 16*8);    // transform key to tower basis
    for (r = 0; r < 16; r++)
        Tb1( b + r*8, Key[0] + r*8 );

    i2b(Xi(1), rconb, 8);       // round const
    for (n = 1; n <= 10; n++) { // calculate new columns until enough
        SboxK(Key[n-1] + rc(1,3), t);
        for (r = 2; r < 5; r++)
        SboxK(Key[n-1] + rc(r,3), Key[n] + rc(r-1,0));
        AddByte(t, rconb, Key[n]);
        memcpy(t+8, rconb, 8);  // next round const
        Tx2(t+8,rconb);
    for (r = 0; r < 4; r++)
        AddByte( Key[n-1] + rc(r,0), Key[n] + rc(r,0), Key[n] + rc(r,0) );

    for (c = 1; c < 4; c++)
    for (r = 0; r < 4; r++)
        AddByte( Key[n-1] + rc(r,c), Key[n] + rc(r,c-1), Key[n] + rc(r,c) );
    }

    memcpy(b, Key[n-1], 16*8);  // transform last key back
    for (r = 0; r < 16; r++)
        Tb2( b + r*8, Key[n-1] + r*8 );
}
```

B Compact inverters for $GF(2^4)$

These first two inverters use the same normal basis for $GF((2^2)^2)$ as in [2, 3, 10], of the form $[Z, Z^4] \times [W, W^2]$; below we call these "normal" inverters. However, the compact AES of this paper uses a different basis, of the form $[1, Z] \times [W, W^2]$; below we call these "mixed" inverters.

Inv4M *GF*(16) normal inverter with 1 AND, 2 XNOR, 5 MUX = **8 ops; depth = 4**.

```
r4 = and(r3, r1); // 0033

r7 = xnor(r4, r2); // f0c3
r5 = xnor(r4, r0); // aa99

r6 = mux(r5, r4, r2); // a093
r9 = mux(r1, r0, r7); // 5371
r11 = mux(r3, r2, r5); // 0a6f

r8 = mux(r0, r6, r1); // 6457
r10 = mux(r2, r6, r3); // 0f93
```

Inv4n *GF*(16) normal inverter with 7 NAND, 7 X(N)OR = **14 ops; depth = 4**.

```
r4 = ~(r3 & r1);    r5 = r3 ^ r2;       r6 = r1 ^ r0;       r11 = ~(r2 & r1);
r7 = r4 ^ r2;       r8 = ~(r5 & r0);    r12 = ~(r11 & r6);
r9 = ~(r7 & r4);    r10 = ~(r7 & r6);   r13 = ~(r8 & r3);   r14 = r12 ^ r4;
r15 = ~(r10 ^ r1);  r16 = r13 ^ r7;     r17 = ~(r9 ^ r8);
```

Below are three versions for the 4-bit inverter used in our new work, with varying tradeoffs between size and depth, from most compact (14 ops, depth 10) to much less deep (16 ops, depth 4).
[Note: for one standard gate library[19], in terms of gate equivalents, the second version below is smaller, at 26 GE, than the first, at 29.33 GE, and the third, at 27.67 GE.]

Inv4 *GF*(16) mixed inverter (as in the C code above, shown here for comparison) with 8 XOR, 5 AND, 1 OR = **14 ops; depth = 10**. outputs = (r4, r2, r5, r1)

```
r4 = r1 & r3;
r5 = r2 ^ r4;
r2 = r1 & r5;    r1 = r1 ^ r5;
r0 = r0 ^ r2;
r4 = r0 & r3;
r5 = r4 ^ r5;
r4 = r0 & r5;    r6 = r1 & r5;
r2 = r2 ^ r4;    r4 = r1 ^ r4;    r1 = r6 ^ r3;
r2 = r1 | r2;
r2 = r0 ^ r2;
```

Inv4 *GF*(16) mixed inverter *version 2* with 6 X(N)OR, 5 (N)AND, 4 NOR = **15 ops; depth = 5**. outputs = (r6, r5, r0, r2)

```
r4 = ~(r0 & r3);    r5 = ~(r1 & r3);    r6 = r0 ^ r2;
r1 = r1 ^~ r4;      r0 = r2 & r5;       r5 = r3 & r5;
r3 = r2 ^~ r3;      r2 = r1 & r2;       r1 = ~(r0 | r1);
r0 = r0 ^~ r4;      r4 = ~(r2 | r6);    r6 = ~(r3 | r6);
r6 = ~(r1 | r6);    r2 = r2 ^~ r3;      r5 = r4 ^~ r5;
```

Inv4 *GF*(16) mixed inverter *version 3* with 7 XOR, 6 AND, 3 OR = **16 ops; depth = 4**. outputs = (r16, r19, r18, r17)

```
r4 = ~(r0 & r2);    r5 = r2 ^ r3;       r6 = ~(r0 | r3);    r7 = ~(r1 & r2);
r8 = ~(r0 | r6);    r9 = r1 ^ ~r4;      r10 = ~(r7 & r5);   r11 = ~(r0 & r7);
r12 = ~(r3 & r9);   r13 = ~(r9 | r8);   r14 = r11 ^ r5;     r15 = r10 ^ ~r8;
r16 = r10 ^ r13;    r17 = ~(r10 & r12); r18 = r7 ^ ~r15;    r19 = r12 ^ r14;
```

Inv4 *GF*(16) mixed inverter *version 4* with 3 MXI2, 3 OAI21, 2 AOI21, 1 XNOR2, 1 NAND3B, 1 NAND3, 1 NAND2 = **13 gates; depth = 4; 17 GE on TSMC180**. outputs = (r16, r15, r13, r12)

C S-box Software Implementations

This section contains example results for 12 different S-box functions. Each is given in C notation, with the final line listing the output values from least to most significant bit. Inputs are, also from least to most significant bit, in r0 to r3. Parallelism is indicated with multiple statements per line.

C.1 2AC

LAC/Lblock S0

```
r4  = r3;
r1 ^= r0;
r3 |= r2;
r3 ^= r1;
r1 &= r2;
r1 ^= r4;
r0 ^= r1;
r0 =~ r0;
r2 ^= r0;
r4 &= r2;
r4 ^= r0;
r0 |= r3;
r0 ^= r1;

// r3 r2 r4 r0
```

Luffa

```
r4  = r1;
r2 ^= r3;
r1 |= r0;
r4 ^= r2;
r1 ^= r3;
r3 &= r0;
r3 ^= r2;
r2 &= r1;
r4 =~ r4;
r2 ^= r4;
r4 |= r3;
r3 ^= r2;
r0 ^= r4;
r4 =~ r4;
r4 ^= r1;

// r2 r3 r4 r0
```

Minalpher

```
r4  = r3;
r3 |= r2;
r3 ^= r0;
r0 |= r4;
r0 ^= r1;
r4 ^= r0;
r1 &= r4;
r1 ^= r3;
r3 ^= r2;
r2 |= r4;
r2 &= r3;
r2 ^= r0;
r0 |= r1;
r1 = ~r1;
r3 ^= r4;
r3 |= r1;
r4 ^= r3;
r3 ^= r0;

// r4 r3 r2 r1
```

Noekeon

```
r4  = r3;
r3 &= r2;
r3 ^= r1;
r1 &= r2;
r0 ^= r1;
r1 |= r2;
r1 ^= r3;
r3 ^= r0;
r1 |= r3;
r3 =~ r3;
r2 |= r3;
r4 ^= r1;
r2 ^= r4;
r4 |= r3;
r1 ^= r4;

// r1 r2 r3 r0
```

Piccolo/Joltik

```
r4  = r1;
r1 |= r2;
r1 ^= r3;
r3 |= r2;
r0 ^= r3;
r3 |= r0;
r3 &= r1;
r1 =~ r1;
r3 &= r0;
r0 =~ r0;
r3 ^= r4;
r4 |= r0;
r2 ^= r4;

// r3 r2 r1 r0
```

Present

```
r1 ^= r2;
r4  = r1;
r1 =~ r1;
r4 |= r3;
r4 ^= r2;
r2 |= r1;
r2 ^= r0;
r3 ^= r2;
r1 ^= r3;
r3 ^= r4;
r0 &= r3;
r0 ^= r4;
r3 ^= r0;
r4 |= r1;
r4 ^= r2;

// r3 r4 r0 r1
```

Prst

```
r4  = r3;
r3 &= r2;
r3 ^= r1;
r1 &= r2;
r4 |= r1;
r1 ^= r0;
r0 &= r3;
r4 ^= r0;
r0 |= r1;
r0 &= r4;
r0 ^= r2;

// r0 r4 r1 r3
```

Qarma

```
r4  = r2;
r2 |= r0;
r2 ^= r1;
r3 ^= r2;
r0 ^= r3;
r2 &= r0;
r1 =~ r1;
r3 ^= r2;
r2 |= r4;
r2 ^= r1;
r2 ^= r3;
r4 &= r2;
r0 ^= r3;
r1 |= r2;
r4 ^= r1;
r1 ^= r0;
r1 |= r4;
r1 ^= r3;
r2 ^= r4;

// r4 r0 r1 r2
```

Qarma2

```
r4  = r2;
r2 |= r1;
r2 ^= r3;
r3 |= r0;
r3 ^= r1;
r3 ^= r2;
r1 |= r0;
r0 ^= r2;
r2 ^= r4;
r4 |= r3;
r0 &= r1;
r1 ^= r4;
r0 ^= r3;
r3 ^= r1;
r1 =~ r1;
r4 &= r1;
r1 |= r0;
r1 ^= r2;

// r0 r1 r4 r3
```

Rectangle

```
r4  = r1;
r1 =~ r1;
r1 |= r3;
r1 ^= r0;
r0 |= r4;
r0 ^= r3;
r3 ^= r2;
r3 &= r1;
r4 ^= r2;
r3 ^= r4;
r2 ^= r1;
r4 ^= r0;
r0 |= r4;
r1 ^= r0;

// r4 r2 r1 r3
```

Skinny S4

```
r0 ^= r3;
r4  = r2;
r3 =~ r3;
r2 |= r1;
r2 ^= r3;
r3 &= r4;
r3 ^= r0;
r1 ^= r3;
r0 &= r2;
r0 ^= r1;
r1 &= r3;
r3 =~ r3;
r1 ^= r4;

// r0 r1 r2 r3
```

Twine

```
r4  = r3;
r3 |= r1;
r3 ^= r2;
r2 |= r1;
r0 ^= r3;
r2 ^= r0;
r1 ^= r2;
r3 ^= r4;
r3 ^= r1;
r0 &= r3;
r0 ^= r4;
r4 |= r3;
r0 =~ r0;
r4 ^= r1;
r1 |= r0;
r1 ^= r2;
r2 |= r4;
r3 ^= r2;

// r3 r4 r1 r0
```

LAC/Lblock S0

```
r0 ^= r1;   r3 =~ r3;
r4  = r3;   r3 &= r2;
r1 ^= r3;   r0 ^= r4;
r3 ^= r0;   r0 |= r2;
r0 ^= r1;   r1 &= r3;
r1 ^= r4;   r4 &= r0;
r2 ^= r4;   r3 =~ r3;

// r3 r0 r2 r1
```

Luffa

```
r4  = r0;   r0 |= r1;
r0 ^= r3;   r2 ^= r3;
r3 &= r4;   r1 ^= r2;
r3 ^= r2;   r2 &= r0;
r0 =~ r0;   r1 =~ r1;
r2 ^= r1;   r1 |= r3;
r4 ^= r1;   r1 ^= r0;
r3 ^= r2;

// r2 r3 r1 r4
```

Minalpher

```
r4  = r0;   r0 |= r3;
r3 =~ r3;   r4 ^= r2;
r4 |= r3;   r0 ^= r3;
r1 ^= r4;   r4 ^= r0;
r0 &= r1;   r3 ^= r1;
r0 ^= r2;   r2 |= r1;
r1 &= r0;   r3 ^= r4;
r1 ^= r4;   r4 &= r2;
r4 ^= r3;   r3 &= r0;
r2 ^= r3;

// r1 r2 r4 r0
```

Noekeon

```
r4  = r3;   r3 &= r2;
r3 ^= r1;   r1 &= r2;
r2 ^= r3;   r3 =~ r3;
r4 ^= r2;   r1 ^= r0;
r0 |= r3;   r3 ^= r1;
r0 &= r4;   r4 |= r3;
r2 ^= r4;   r4 |= r1;
r4 ^= r0;

// r2 r4 r3 r1
```

Piccolo/Joltik

```
r4  = r1;   r1 |= r2;
r1 ^= r3;   r3 =~ r3;
r0 ^= r3;   r3 &= r2;
r3 ^= r0;   r0 |= r1;
r1 =~ r1;   r0 ^= r4;
r0 ^= r3;   r4 |= r3;
r2 ^= r4;

// r0 r2 r1 r3
```

Present

```
r4  = r3;   r1 ^= r2;
r3 |= r1;   r1 =~ r1;
r3 ^= r2;   r4 ^= r0;
r2 |= r1;   ;
r2 ^= r4;   r4 |= r3;
r3 ^= r2;   r4 ^= r1;
r4 ^= r0;   r0 &= r3;
r1 ^= r2;   r2 ^= r0;
r3 ^= r2;

// r2 r4 r3 r1
```

Prst

```
r4  = r3;   r3 &= r2;
r3 ^= r1;   r1 &= r2;
r4 |= r1;   ;
r1 ^= r0;   r0 &= r3;
r4 ^= r0;   r0 |= r1;
r0 &= r4;   ;
r2 ^= r0;

// r2 r4 r1 r3
```

Qarma

```
r4  = r2;   r2 ^= r0;
r2 |= r3;   r3 =~ r3;
r2 ^= r1;   r1 |= r3;
r3 ^= r0;   r0 |= r2;
r1 &= r0;   r0 &= r4;
r4 ^= r2;   r2 ^= r3;
r2 &= r4;   r4 ^= r1;
r4 |= r2;   r2 ^= r0;
r0 &= r1;   r3 ^= r2;
r4 ^= r3;   r3 ^= r0;

// r3 r1 r4 r2
```

Qarma2

```
r4  = r2;   r2 |= r1;
r2 ^= r3;   r3 |= r0;
r3 ^= r1;   r1 |= r0;
r0 ^= r2;   ;
r3 ^= r2;   r2 ^= r4;
r0 &= r1;   r4 |= r3;
r4 ^= r1;   r0 ^= r3;
r3 ^= r4;   r4 =~ r4;
r1 &= r4;   r4 |= r0;
r2 ^= r4;

// r0 r2 r1 r3
```

Rectangle

```
r4  = r3;   r1 =~ r1;
r3 ^= r2;   r4 |= r1;
r4 ^= r0;   r0 &= r1;
r0 ^= r3;   r1 ^= r2;
r3 &= r4;   r1 =~ r1;
r3 ^= r1;   r1 |= r0;
r1 ^= r4;   r2 ^= r4;

// r0 r2 r1 r3
```

Skinny S4

```
r0 ^= r3;   r3 =~ r3;
r4  = r1;   r1 |= r2;
r1 ^= r3;   r3 &= r2;
r3 ^= r0;   ;
r0 &= r1;   r4 ^= r3;
r0 ^= r4;   r4 &= r3;
r2 ^= r4;   r3 =~ r3;

// r0 r2 r1 r3
```

Twine

```
r4  = r1;   r2 =~ r2;
r1 |= r2;   r2 ^= r3;
r2 &= r4;   r0 ^= r3;
r1 ^= r0;   r0 ^= r2;
r2 |= r1;   ;
r1 =~ r1;   r2 ^= r3;
r4 ^= r0;   r3 |= r1;
r3 ^= r0;   r0 |= r2;
r0 ^= r4;   r4 |= r3;
r4 ^= r1;

// r4 r3 r0 r2
```

LAC/Lblock S0

```
r1 ^= r0;   r4  = r2;   r2 |= r3;
r0 ^= r4;   r2 ^= r1;   r1 &= r4;
r1 =~ r1;   r3 ^= r0;   ;
r0 ^= r1;   r1 ^= r3;   r3 &= r2;
r3 ^= r0;   r0 &= r1;   ;
r0 ^= r4;

// r2 r1 r0 r3
```

Luffa

```
r4  = r1;   r2 ^= r3;   r1 |= r0;
r4 =~ r4;   r1 ^= r3;   r3 &= r0;
r4 ^= r3;   r3 ^= r2;   r2 &= r1;
r1 =~ r1;   r2 ^= r4;   r4 |= r3;
r3 ^= r2;   r1 ^= r4;   r4 ^= r0;

// r3 r2 r1 r4
```

Minalpher

```
r3 =~ r3;   r4  = r0;   r0 ^= r2;
r4 &= r3;   r0 |= r3;   ;
r0 ^= r1;   r1 ^= r4;   r4 ^= r2;
r4 &= r0;   r1 =~ r1;   ;
r2 |= r0;   r0 ^= r1;   r3 ^= r1;
r1 ^= r4;   r4 ^= r2;   r0 &= r2;
r0 ^= r3;   r3 &= r4;   ;
r2 ^= r3;

// r1 r2 r0 r4
```

Noekeon

```
r4  = r3;   r3 &= r2;   r2 =~ r2;
r0 ^= r1;   r3 ^= r1;   r1 &= r2;
r1 ^= r0;   r3 =~ r3;   r2 ^= r4;
r0 &= r3;   r3 ^= r1;   r2 ^= r1;
r0 ^= r2;   r2 &= r3;   ;
r2 ^= r4;

// r2 r0 r3 r1
```

Piccolo/Joltik

```
r0 ^= r3;   r4  = r1;   r1 |= r2;
r1 ^= r3;   r3 &= r2;   r0 ^= r2;
r3 ^= r0;   r0 &= r1;   r2 =~ r2;
r0 ^= r4;   r4 &= r3;   r2 ^= r3;
r2 ^= r4;   r3 =~ r3;   r1 =~ r1;

// r0 r2 r1 r3
```

Present

```
r4  = r1;   r1 &= r2;   ;
r2 |= r4;   r4 ^= r3;   r3 |= r1;
r3 &= r2;   r1 ^= r0;   r2 ^= r4;
r4 ^= r1;   r1 |= r3;   r3 =~ r3;
r1 ^= r4;   r3 ^= r0;   ;
r4 |= r3;   r0 ^= r2;   r3 ^= r1;
r4 ^= r2;

// r3 r4 r1 r0
```

Prst

```
r4  = r3;   r0 ^= r1;   r3 &= r2;
r3 ^= r1;   r1 |= r2;   r0 ^= r2;
r1 ^= r0;   r0 &= r3;   r4 ^= r3;
r0 ^= r4;   r4 &= r1;   ;
r2 ^= r4;

// r2 r0 r1 r3
```

Qarma

```
r4  = r0;   r3 ^= r0;   r0 &= r2;
r0 ^= r3;   r3 |= r1;   r1 =~ r1;
r3 ^= r0;   r4 ^= r0;   r0 |= r2;
r2 ^= r1;   r4 ^= r0;   r1 |= r3;
r2 ^= r4;   r4 &= r0;   r0 &= r1;
r1 ^= r3;   r4 ^= r3;   r3 ^= r0;
r2 ^= r0;   r1 |= r3;   r3 &= r4;
r3 ^= r2;

// r1 r4 r3 r0
```

Qarma2

```
r4  = r0;   r0 |= r1;   r1 ^= r2;
r2 &= r0;   r0 ^= r4;   r3 =~ r3;
r4 ^= r2;   r1 ^= r3;   r3 |= r0;
r0 ^= r4;   r4 &= r3;   r1 ^= r2;
r3 ^= r1;   r2 |= r4;   r1 |= r4;
r0 ^= r1;   r4 ^= r3;   r1 |= r2;
r4 ^= r0;   r1 ^= r0;

// r3 r4 r2 r1
```

Rectangle

```
r1 =~ r1;   ;           ;
r4  = r1;   r1 |= r3;   r3 ^= r2;
r1 ^= r0;   r0 &= r4;   r4 ^= r2;
r0 ^= r3;   r4 =~ r4;   r3 &= r1;
r3 ^= r4;   r2 ^= r1;   r4 |= r0;
r4 ^= r1;

// r0 r2 r4 r3
```

Skinny S4

```
r4  = r2;   r0 ^= r3;   r2 |= r1;
r2 ^= r3;   r3 &= r4;   r0 ^= r4;
r3 ^= r0;   r0 &= r2;   r2 =~ r2;
r4 ^= r3;   r0 ^= r1;   r1 &= r3;
r3 =~ r3;   r4 ^= r1;

// r0 r4 r2 r3
```

Twine

```
r4  = r2;   r0 ^= r3;   r2 ^= r3;
r4 |= r1;   r2 &= r1;   r1 ^= r0;
r4 ^= r1;   r0 =~ r0;   r1 ^= r2;
r2 ^= r0;   r3 ^= r4;   r0 |= r1;
r0 ^= r3;   r3 |= r4;   ;
r3 ^= r1;   r1 |= r0;   r2 =~ r2;
r1 ^= r2;   r2 |= r3;   ;
r4 ^= r2;

// r4 r3 r1 r0
```

LAC/Lblock S0

```
r1 ^= r0;   r0 ^= r3;   r4  = r2;   r2 |= r3;
r2 ^= r1;   r1 &= r4;   r4 ^= r0;   r0 =~ r0;
r1 ^= r4;   r4 ^= r0;   r0 |= r2;   ;
r0 ^= r3;   r3 |= r1;   r1 =~ r1;   ;
r3 ^= r4;

// r2 r1 r3 r0
```

Luffa

```
r4  = r3;   r2 ^= r3;   r1 =~ r1;   r3 &= r0;
r2 ^= r3;   r3 ^= r1;   r4 ^= r0;   r1 |= r0;
r4 ^= r1;   r1 &= r3;   r3 |= r2;   ;
r0 ^= r3;   r3 ^= r4;   r4 &= r2;   r2 ^= r1;
r1 ^= r4;   r2 ^= r4;

// r1 r2 r3 r0
```

Minalpher

```
r4  = r2;   r2 ^= r1;   r1 |= r3;   ;
r4 &= r1;   r2 ^= r0;   r1 ^= r3;   ;
r3 ^= r2;   r2 |= r4;   r1 =~ r1;   ;
r3 &= r1;   r4 ^= r1;   r0 ^= r1;   r1 &= r2;
r4 ^= r3;   r3 ^= r2;   r2 |= r0;   r0 ^= r1;
r3 &= r2;   r2 &= r0;   r0 &= r4;   ;
r0 ^= r1;

// r2 r0 r3 r4
```

Noekeon

```
r4  = r0;   r0 ^= r3;   r3 |= r2;   r2 =~ r2;
r3 ^= r1;   r1 |= r2;   r2 ^= r4;   r4 =~ r4;
r4 ^= r3;   r2 ^= r1;   r1 ^= r0;   r0 &= r3;
r0 ^= r4;   r4 &= r3;   r3 ^= r1;   r1 |= r2;
r4 ^= r1;

// r0 r4 r3 r2
```

Piccolo/Joltik

```
r4  = r2;   r0 ^= r2;   r2 |= r1;   ;
r2 ^= r3;   r0 ^= r3;   r3 &= r4;   ;
r3 ^= r0;   r0 &= r2;   r2 =~ r2;   r4 ^= r1;
r0 ^= r1;   r1 |= r3;   r3 =~ r3;   r4 =~ r4;
r1 ^= r4;

// r0 r1 r2 r3
```

Present

```
r4  = r0;   r0 ^= r3;   r3 ^= r1;   r1 ^= r2;
r2 &= r1;   r3 |= r1;   r1 ^= r0;   ;
r0 ^= r2;   r2 ^= r3;   r3 ^= r1;   r1 ^= r4;
r4 |= r3;   r3 ^= r2;   r2 =~ r2;   ;
r3 ^= r4;   r4 ^= r2;   r2 |= r0;   ;
r1 ^= r2;

// r4 r1 r3 r0
```

Prst

```
r4  = r3;   r3 &= r2;   r0 ^= r1;   ;
r3 ^= r1;   r1 |= r2;   r0 ^= r2;   ;
r1 ^= r0;   r0 &= r3;   r4 ^= r3;   ;
r0 ^= r4;   r4 &= r1;   ;           ;
r4 ^= r2;

// r4 r0 r1 r3
```

Qarma

```
r4  = r2;   r2 &= r0;   r0 ^= r3;   r3 =~ r3;
r3 ^= r2;   r2 ^= r0;   r0 |= r1;   ;
r0 ^= r2;   r2 |= r4;   r4 ^= r3;   ;
r4 ^= r1;   r3 &= r2;   r1 |= r0;   ;
r4 ^= r1;   r1 ^= r2;   r2 =~ r2;   r3 ^= r0;
r2 |= r1;   r4 ^= r0;   r0 ^= r1;   r1 &= r3;
r4 ^= r1;

// r2 r3 r4 r0
```

Qarma2

```
r4  = r2;   r3 =~ r3;   r2 ^= r1;   r1 |= r0;
r4 &= r1;   r2 ^= r3;   r1 ^= r0;   ;
r0 ^= r4;   r2 ^= r4;   r3 |= r1;   r4 |= r1;
r4 ^= r1;   r1 ^= r0;   r0 &= r3;   r3 ^= r2;
r1 ^= r2;   r2 |= r4;   r4 |= r0;   r0 |= r3;
r2 ^= r1;   r1 ^= r0;

// r3 r1 r4 r2
```

Rectangle

```
r4  = r1;   r2 ^= r1;   r1 |= r0;   r0 =~ r0;
r1 ^= r3;   r3 &= r4;   r4 ^= r0;   r0 ^= r2;
r3 ^= r0;   r0 |= r1;   r4 &= r1;   ;
r1 ^= r2;   r2 &= r0;   r0 ^= r4;   ;
r4 |= r2;

// r1 r3 r0 r4
```

Skinny S4

```
r0 ^= r2;   r4  = r2;   r2 |= r1;   ;
r0 ^= r3;   r2 ^= r3;   r3 &= r4;   ;
r3 ^= r0;   r0 &= r2;   r2 =~ r2;   ;
r0 ^= r1;   r1 &= r3;   r4 ^= r3;   r3 =~ r3;
r1 ^= r4;

// r0 r1 r2 r3
```

Twine

```
r4  = r1;   r1 ^= r2;   r2 &= r0;   r0 =~ r0;
r1 ^= r0;   r2 ^= r3;   r3 ^= r4;   ;
r4 ^= r1;   r1 &= r2;   r3 |= r2;   ;
r2 ^= r3;   r3 ^= r0;   r0 |= r1;   ;
r3 ^= r1;   r1 ^= r4;   r0 |= r2;   r4 |= r2;
r2 ^= r4;   r4 &= r0;   r0 ^= r3;   r3 |= r1;
r3 ^= r2;

// r0 r3 r1 r4
```

C.2 3AC

LAC/Lblock S0

```
r4 = r2 | r3;
r1 = r0 ^ r1;
r4 = r1 ^ r4;
r1 = r2 & r4;
r0 = r0 ^ r1;
r1 = ~r3;
r0 = r0 ^ r1;
r1 = r0 & r1;
r1 = r1 ^ r2;
r2 = r0 | r4;
r3 = r2 ^ r3;

// r4 r0 r1 r3
```

Luffa

```
r4 = r0 | r3;
r4 = r2 ^ r4;
r2 = ~r1;
r1 = r0 | r1;
r3 = r1 ^ r3;
r1 = r3 & r4;
r2 = r1 ^ r2;
r1 = r0 ^ r2;
r2 = r2 ^ r4;
r4 = r1 | r2;
r0 = r0 ^ r4;
r4 = ~r4;
r3 = r3 ^ r4;

// r2 r1 r3 r0
```

Minalpher

```
r4 = r0 & r1;
r2 = r2 ^ r4;
r4 = r2 & r3;
r2 = r0 ^ r2;
r1 = r1 ^ r4;
r4 = ~r2;
r3 = r1 ^ r3;
r3 = r3 ^ r4;
r4 = r1 | r4;
r0 = r0 ^ r4;
r4 = r0 & r3;
r4 = r2 ^ r4;
r1 = r0 ^ r1;
r2 = r1 & r4;
r2 = r0 ^ r2;

// r1 r4 r2 r3
```

Noekeon

```
r4 = r2 | r3;
r4 = r1 ^ r4;
r3 = r3 ^ r4;
r1 = r1 & r2;
r2 = r2 ^ r3;
r0 = r0 ^ r1;
r1 = ~r2;
r2 = r0 | r1;
r2 = r2 ^ r4;
r1 = r0 ^ r1;
r4 = r1 | r4;
r3 = r3 ^ r4;

// r3 r2 r1 r0
```

Piccolo/Joltik

```
r4 = r1 | r2;
r4 = r3 ^ r4;
r3 = r2 | r3;
r3 = r0 ^ r3;
r0 = r3 & r4;
r0 = r0 ^ r1;
r4 = ~r4;
r3 = ~r3;
r1 = r1 | r3;
r1 = r1 ^ r2;

// r0 r1 r4 r3
```

Present

```
r4 = r1 ^ r2;
r1 = r2 & r4;
r3 = r1 ^ r3;
r1 = r3 & r4;
r2 = r1 ^ r2;
r1 = ~r2;
r1 = r0 ^ r1;
r2 = r0 | r2;
r0 = r0 ^ r3;
r4 = r0 ^ r4;
r2 = r2 ^ r4;
r4 = r1 | r4;
r4 = r3 ^ r4;
r1 = r1 ^ r2;

// r1 r4 r2 r0
```

Prst

```
r4 = r1 & r2;
r0 = r0 ^ r4;
r4 = r2 & r3;
r4 = r1 ^ r4;
r1 = r0 & r4;
r1 = r1 ^ r3;
r3 = r0 & r1;
r3 = r2 ^ r3;

// r3 r1 r0 r4
```

Qarma

```
r4 = r0 & r3;
r4 = r2 | r4;
r4 = r1 ^ r4;
r1 = r0 ^ r3;
r0 = r1 | r4;
r0 = r0 ^ r3;
r3 = ~r4;
r4 = r0 & r2;
r4 = r1 ^ r4;
r1 = r3 & r4;
r1 = r1 ^ r2;
r2 = r1 & r4;
r2 = r0 | r2;
r2 = r2 ^ r3;
r3 = ~r1;
r3 = r3 ^ r4;

// r3 r0 r2 r1
```

Qarma2

```
r4 = r1 | r2;
r4 = r3 ^ r4;
r3 = r0 & r4;
r1 = r1 ^ r3;
r3 = r0 | r1;
r4 = r2 ^ r4;
r4 = ~r4;
r0 = r0 ^ r4;
r0 = r0 & r3;
r2 = r1 ^ r2;
r0 = r0 ^ r2;
r2 = r1 & r2;
r4 = r2 ^ r4;
r3 = r2 ^ r3;
r2 = r0 & r2;
r4 = r2 ^ r4;
r1 = r1 ^ r3;

// r0 r4 r3 r1
```

Rectangle

```
r4 = r0 | r1;
r3 = r3 ^ r4;
r4 = r1 & r3;
r0 = r0 ^ r4;
r0 = ~r0;
r1 = r1 ^ r2;
r2 = r0 ^ r2;
r4 = r1 | r3;
r4 = r0 ^ r4;
r0 = r1 ^ r3;
r1 = r0 & r4;
r1 = r1 ^ r3;

// r0 r2 r4 r1
```

Skinny S4

```
r4 = r1 | r2;
r4 = r3 ^ r4;
r3 = r2 | r3;
r0 = r0 ^ r3;
r3 = r0 & r4;
r3 = r1 ^ r3;
r1 = r0 & r1;
r2 = r1 ^ r2;
r1 = ~r4;
r4 = ~r0;
r0 = r0 ^ r2;

// r3 r0 r1 r4
```

Twine

```
r4 = r1 | r2;
r0 = r0 ^ r4;
r4 = ~r3;
r0 = r0 ^ r4;
r4 = r0 & r4;
r4 = r2 ^ r4;
r2 = r0 ^ r1;
r0 = r0 & r4;
r0 = r0 | r2;
r3 = r0 ^ r3;
r0 = r3 & r4;
r1 = r0 ^ r1;
r0 = r2 ^ r4;
r2 = r0 | r1;
r4 = r2 ^ r4;

// r4 r0 r1 r3
```

LAC/Lblock S0

```
r4 = r2 | r3;   r1 = r0 ^ r1;
r3 = ~r3;       r4 = r1 ^ r4;
r1 = r2 & r4;   r0 = r0 ^ r3;
r1 = r0 ^ r1;   r0 = r0 | r4;
r0 = r0 ^ r3;   r3 = r1 & r3;
r3 = r2 ^ r3;   r0 = ~r0;

// r4 r1 r3 r0
```

Luffa

```
r4 = ~r1;       r1 = r0 | r1;
r2 = r1 ^ r2;   r3 = r1 ^ r3;
r1 = r2 & r3;   ;
r1 = r1 ^ r4;   r4 = r0 | r3;
r4 = r1 ^ r4;   r1 = r1 ^ r2;
r3 = ~r3;       r2 = r1 | r4;
r0 = r0 ^ r2;   r3 = r2 ^ r3;

// r1 r4 r3 r0
```

Minalpher

```
r4 = r0 & r1;   ;
r2 = r2 ^ r4;   r4 = ~r0;
r0 = r0 ^ r2;   r2 = r2 & r3;
r1 = r1 ^ r2;   r2 = r0 ^ r3;
r2 = r1 ^ r2;   r0 = r0 | r1;
r4 = r0 ^ r4;   r0 = r1 & r2;
r0 = r0 ^ r3;   r3 = r2 | r4;
r3 = r0 ^ r3;   r0 = r0 & r4;
r1 = r0 ^ r1;   r0 = ~r2;

// r4 r3 r1 r0
```

Noekeon

```
r4 = ~r2;       r2 = r2 | r3;
r2 = r1 ^ r2;   r1 = r1 | r4;
r0 = r0 ^ r1;   ;
r3 = r0 ^ r3;   r4 = r0 ^ r4;
r1 = r2 ^ r3;   r3 = r2 & r3;
r0 = r0 ^ r3;   r3 = r1 | r4;
r3 = r2 ^ r3;

// r0 r3 r1 r4
```

Piccolo/Joltik

```
r4 = r1 | r2;   ;
r4 = r3 ^ r4;   r3 = r2 | r3;
r0 = r0 ^ r3;   r3 = ~r4;
r4 = r0 & r4;   r0 = ~r0;
r4 = r1 ^ r4;   r1 = r0 | r1;
r1 = r1 ^ r2;

// r4 r1 r3 r0
```

Present

```
r4 = r1 ^ r2;   r1 = r1 & r2;
r1 = r0 ^ r1;   r2 = r2 ^ r3;
r3 = r3 & r4;   r2 = r1 ^ r2;
r1 = r1 ^ r3;   r4 = r2 ^ r4;
r3 = r0 | r1;   r1 = ~r1;
r3 = r3 ^ r4;   r4 = r1 | r4;
r1 = r1 ^ r3;   r0 = r0 ^ r2;
r0 = r0 ^ r4;

// r1 r0 r3 r2
```

Prst

```
r4 = r1 & r2;   ;
r4 = r0 ^ r4;   r0 = r2 & r3;
r0 = r0 ^ r1;   r1 = r3 & r4;
r2 = r1 ^ r2;   r1 = r0 & r4;
r2 = r1 ^ r2;   r1 = r1 ^ r3;

// r2 r1 r4 r0
```

Qarma

```
r4 = r0 ^ r3;   r3 = r0 & r3;
r1 = ~r1;       r3 = r2 | r3;
r3 = r1 ^ r3;   r1 = r1 & r4;
r4 = ~r4;       r1 = r0 | r1;
r0 = r0 ^ r3;   r3 = r3 | r4;
r1 = r1 ^ r3;   r0 = r0 ^ r3;
r3 = r2 ^ r3;   r2 = r0 & r2;
r4 = r2 ^ r4;   r2 = r1 & r2;
r2 = r2 ^ r3;   ;
r4 = r2 ^ r4;

// r2 r0 r1 r4
```

Qarma2

```
r4 = ~r3;       r3 = r1 & r2;
r3 = r3 ^ r4;   r4 = r0 | r1;
r1 = r1 ^ r3;   r3 = r0 & r3;
r0 = r0 ^ r2;   r2 = r2 | r3;
r1 = r1 ^ r2;   r2 = r2 & r4;
r3 = r3 ^ r4;   r4 = r1 & r4;
r4 = r0 ^ r4;   r0 = r0 & r2;
r3 = r0 ^ r3;   r0 = r2 | r4;
r4 = r3 ^ r4;   r0 = r0 ^ r1;

// r4 r0 r2 r3
```

Rectangle

```
r4 = r0 | r1;   ;
r3 = r3 ^ r4;   r4 = r1 ^ r2;
r1 = r1 & r3;   r0 = ~r0;
r0 = r0 ^ r1;   r1 = r3 | r4;
r4 = r3 ^ r4;   r1 = r0 ^ r1;
r0 = r0 ^ r2;   r2 = r1 & r4;
r3 = r2 ^ r3;

// r4 r0 r1 r3
```

Skinny S4

```
r4 = r1 ^ r2;   r2 = r2 | r3;
r2 = r0 ^ r2;   r0 = r1 | r4;
r0 = r0 ^ r3;   r3 = r1 | r2;
r3 = r3 ^ r4;   r4 = r0 & r2;
r4 = r1 ^ r4;   r1 = ~r0;
r2 = ~r2;

// r4 r3 r1 r2
```

Twine

```
r4 = ~r2;       r0 = r0 ^ r3;
r2 = r1 | r4;   r4 = r3 ^ r4;
r2 = r0 ^ r2;   r4 = r1 & r4;
r0 = r0 ^ r4;   r4 = r2 | r4;
r1 = r0 ^ r1;   r2 = ~r2;
r4 = r3 ^ r4;   r3 = r2 | r3;
r3 = r0 ^ r3;   r0 = r0 | r4;
r0 = r0 ^ r1;   r1 = r1 | r3;
r2 = r1 ^ r2;

// r2 r3 r0 r4
```

D S-box Hardware Implementations

This section contains results for five different S-box functions optimized for the standard cell library UMC180[20]. Like the software implementations, they are given in C notation, with the final line listing the output values from least to most significant bit. Inputs are likewise in r0 to r3 for the 4-input S-boxes and r0 to r4 for the 5-input ones. Parallelism is indicated with a blank line, so that each layer of circuitry is in one group of lines. Actual circuit delay is not taken into account, only logical depth in terms of the number of standard cells.

LAC/LBlock S0

```
 r4 = or2   (r2, r3);
 r5 = inv   (r3);

 r6 = nand2 (r1, r5);
 r7 = nor2  (r1, r5);
 r8 = xnor2 (r0, r4);

 r9 = nand2 (r2, r6);
r10 = xnor2 (r1, r8);
r11 = nor2  (r2, r8);
r12 = moai1 (r0, r4, r2, r6);
r13 = nor2  (r7, r9);

r14 = moai1 (r0, r7, r5, r10);
r15 = nor2  (r11, r13);

// r10 r15 r12 r14
```

Present

```
 r4 = nor2   (r1, r3);
 r5 = inv    (r2);

 r6 = moai1  (r0, r4, r2, r3);
 r7 = nor2   (r1, r5);
 r8 = nand2  (r1, r5);

 r9 = xnor2  (r3, r8);
r10 = nand2b (r8, r7);

r11 = inv    (r9);
r12 = xnor2  (r0, r9);
r13 = xnor2  (r6, r10);

r14 = moai1  (r4, r6, r11, r12);
r15 = xnor2  (r10, r12);
r16 = moai1  (r2, r11, r6, r12);

// r14 r13 r16 r15
```

Twine

```
r4 = nand2  (r1, r2);
r5 = xnor2  (r0, r2);
r6 = inv    (r1);
r7 = nor2   (r0, r1);

r8 = nor2   (r3, r5);
r9 = xnor2  (r3, r7);
r10 = moai1  (r3, r6, r0, r2);
r11 = or2    (r2, r7);

r12 = inv    (r9);
r13 = moai1  (r8, r11, r2, r9);
r14 = moai1  (r5, r9, r4, r10);

r15 = moai1  (r6, r8, r5, r12);

r16 = moai1  (r6, r15, r4, r5);

// r13 r14 r15 r16
```

Ascon

```
 r5 = inv    (r1);
 r6 = xnor2  (r2, r3);
 r7 = xnor2  (r0, r4);

 r8 = nand2  (r0, r5);
 r9 = xnor2  (r2, r5);
r10 = nand2  (r3, r7);
r11 = xnor2  (r0, r5);
r12 = or2    (r3, r6);

r13 = nand2b (r6, r9);
r14 = or2    (r4, r11);
r15 = xnor2  (r10, r11);
r16 = xnor2  (r7, r12);
r17 = xnor2  (r8, r6);

r18 = xnor2  (r6, r14);
r19 = xnor2  (r15, r16);
r20 = xnor2  (r13, r7);

// r15 r18 r17 r20 r19
```

Keccak

```
 r5 = inv    (r2);
 r6 = inv    (r0);
 r7 = nand2b (r3, r4);

 r8 = nor2   (r1, r5);
 r9 = nand2  (r3, r5);
r10 = or2    (r4, r6);
r11 = nand2  (r1, r6);

r12 = xnor2  (r8, r6);
r13 = xnor2  (r1, r9);
r14 = xnor2  (r2, r7);
r15 = xnor2  (r3, r10);
r16 = xnor2  (r4, r11);

// r12 r13 r14 r15 r16
```

E GF256 Inverter Function with Rotational Symmetry

Below is an implementation of the 8-input 1-output core of the T normal basis GF256 inverter function. This implementation is based on a subset of the digital standard cells available in the UMC180 standard cell library [20]. Its area is 79.3 GE, and it is 15 gates deep.

```
a0 = nand2 (x0, x2);      a1 = nand2 (x0, x3);      a2 = or2   (x0, x3);
a3 = or2   (x0, x2);

b0 = nand2 (x2, a0);      b1 = nand3 (x1, a3, a0);  b2 = nand3 (x1, a2, a1);

c0 = nand2 (x1, b2);      c1 = nand2 (x1, b1);      c2 = nand3 (a2, a3, b1);
c3 = nand3 (x3, a0, b1);

d0 = nand2 (c2, c0);      d1 = or2   (x2, c0);      d2 = or2   (x1, a1);
d3 = nand3 (a3, c0, c3);  d4 = nand3 (a2, a0, c1);  d5 = and2  (b0, c3);

e0 = nand3 (b0, d1, d2);  e1 = nand3 (a1, d4, d1);  e2 = nand3 (a0, d0, d1);
e3 = nand3 (b1, d0, d3);

f0 = nand2 (d2, e1);      f1 = nand2 (d4, e0);      f2 = nand2 (e0, e1);
f3 = nor2  (a1, e3);      f4 = nand2 (b1, e1);      f5 = nand3 (a2, e1, e2);

g0 = nand2 (c3, f2);      g1 = nand2 (d3, f0);      g2 = and2  (d0, f1);

h0 = nand2 (d0, g1);      h1 = nand2 (c2, g0);      h2 = nand3 (x1, e3, g1);
h3 = nand3 (c0, e3, g1);

i0 = nand3 (b0, d2, h2);  i1 = nand3 (b0, c1, h0);  i2 = nand3 (b1, d2, h1);

j0 = nand3 (a1, e2, i0);  j1 = nand2 (h2, i1);      j2 = nand3 (b1, d4, i1);

k0 = and2  (e3, j0);

l0 = moai1 (x4, f3, x4, g2);    l1 = moai1 (x4, d5, x4, i2);
l2 = moai1 (x4, h3, x4, i0);    l3 = moai1 (x4, h0, x4, h1);
l4 = moai1 (x4, k0, x4, f4);    l5 = moai1 (x4, e2, x4, j1);
l6 = moai1 (x4, j2, x4, f5);    l7 = moai1 (x4, f0, x4, f2);

m0 = moai1 (x5, l0, x5, l1);    m1 = moai1 (x5, l2, x5, l3);
m2 = moai1 (x5, l4, x5, l5);    m3 = moai1 (x5, l6, x5, l7);

n0 = moai1 (x6, m0, x6, m1);    n1 = moai1 (x6, m2, x6, m3);

y  = moai1 (x7, n0, x7, n1);
```

Here is another alternative circuit for this function. At 84 GE it is not quite as small, but its depth is only 11 gates.

```
a0 = xnor2 (x0, x1);    a1 = xnor2 (x1, x2);    a2 = nor2  (x0, x2);
a3 = nor2  (x2, x3);

b0 = nand2 (x2, a0);    b1 = nor2  (x2, a0);    b2 = xnor2 (x3, a0);

c0 = nand2 (x1, b0);    c1 = nor2  (x2, b2);    c2 = xnor2 (b2, a2);
c3 = or2   (x3, b1);    c4 = nand2 (a0, b0);

d0 = nand2 (x0, c0);    d1 = nand2 (x0, c1);    d2 = nand2 (b2, c3);
d3 = nor2  (x0, c2);    d4 = nand2 (b0, c2);    d5 = nand2 (a1, c2);
d6 = nand2 (c0, c4);

e0 = nor2  (a1, d2);    e1 = nand2 (b0, d1);    e2 = nor2  (b1, d3);
e3 = nand2 (d2, d4);    e4 = nand2 (d5, c4);    e5 = and2  (x0, d1);
e6 = nor2  (c0, d4);    e7 = nor2  (a0, d4);    e8 = nand2 (c0, d4);
e9 = nor2  (c2, d0);    ea = inv   (d2);        eb = nand2 (c3, d6);

f0 = nand2 (x3, e1);    f1 = or2   (a2, e2);    f2 = nor2  (e0, d3);
f3 = nand2 (e4, b0);    f4 = nor2  (e0, e5);    f5 = nand2 (e5, d4);
f6 = nor2  (e1, e6);    f7 = nand2 (e2, e3);    f8 = nand2 (x1, e4);
f9 = or2   (e5, a3);    fa = nor2  (e9, a2);    fb = nor2  (ea, e6);
fc = nor2  (e7, e0);

g0 = nor2  (e0, f0);    g1 = nand2 (f5, f0);    g2 = nand2 (f8, e3);
g3 = nand2 (f2, f0);    g4 = nand2 (f3, f1);

i0 = moai1 (x4, g0, x4, g1);    i1 = moai1 (x4, f9, x4, e8);
i2 = moai1 (x4, g3, x4, fa);    i3 = moai1 (x4, f4, x4, eb);
i4 = moai1 (x4, f6, x4, g2);    i5 = moai1 (x4, f1, x4, e4);
i6 = moai1 (x4, g4, x4, f7);    i7 = moai1 (x4, fb, x4, fc);

j0 = moai1 (x5, i0, x5, i1);    j1 = moai1 (x5, i2, x5, i3);
j2 = moai1 (x5, i4, x5, i5);    j3 = moai1 (x5, i6, x5, i7);

k0 = moai1 (x6, j0, x6, j1);    k1 = moai1 (x6, j2, x6, j3);

y  = moai1 (x7, k0, x7, k1);
```

F AES MixColumns with 94 XORs and depth 5

```
r64 = r8 ^ r0;      // 00000101 = 00000100 ^ 00000001
r65 = r8 ^ r1;      // 00000102 = 00000100 ^ 00000002
r66 = r9 ^ r1;      // 00000202 = 00000200 ^ 00000002
r67 = r10 ^ r2;     // 00000404 = 00000400 ^ 00000004
r68 = r11 ^ r3;     // 00000808 = 00000800 ^ 00000008
r69 = r12 ^ r4;     // 00001010 = 00001000 ^ 00000010
r70 = r14 ^ r6;     // 00004040 = 00004000 ^ 00000040
r71 = r15 ^ r7;     // 00008080 = 00008000 ^ 00000080
r72 = r16 ^ r8;     // 00010100 = 00010000 ^ 00000100
r73 = r21 ^ r13;    // 00202000 = 00200000 ^ 00002000
r74 = r24 ^ r0;     // 01000001 = 01000000 ^ 00000001
r75 = r24 ^ r16;    // 01010000 = 01000000 ^ 00010000
r76 = r25 ^ r17;    // 02020000 = 02000000 ^ 00020000
r77 = r26 ^ r18;    // 04040000 = 04000000 ^ 00040000
r78 = r27 ^ r19;    // 08080000 = 08000000 ^ 00080000
r79 = r28 ^ r20;    // 10100000 = 10000000 ^ 00100000
r80 = r29 ^ r5;     // 20000020 = 20000000 ^ 00000020
r81 = r29 ^ r13;    // 20002000 = 20000000 ^ 00002000
r82 = r30 ^ r22;    // 40400000 = 40000000 ^ 00400000
r83 = r31 ^ r7;     // 80000080 = 80000000 ^ 00000080

r84 = r66 ^ r16;    // 00010202 = 00000202 ^ 00010000
r85 = r66 ^ r17;    // 00020202 = 00000202 ^ 00020000
r86 = r67 ^ r3;     // 0000040c = 00000404 ^ 00000008
r87 = r67 ^ r18;    // 00040404 = 00000404 ^ 00040000
r88 = r68 ^ r19;    // 00080808 = 00000808 ^ 00080000
r89 = r69 ^ r28;    // 10001010 = 00001010 ^ 10000000
r90 = r70 ^ r21;    // 00204040 = 00004040 ^ 00200000
r91 = r70 ^ r23;    // 00804040 = 00004040 ^ 00800000
r92 = r71 ^ r23;    // 00808080 = 00008080 ^ 00800000
r93 = r72 ^ r15;    // 00018100 = 00010100 ^ 00008000
r94 = r74 ^ r12;    // 01001001 = 01000001 ^ 00001000
r95 = r74 ^ r65;    // 01000103 = 01000001 ^ 00000102
r96 = r75 ^ r25;    // 03010000 = 01010000 ^ 02000000
r97 = r76 ^ r1;     // 02020002 = 02020000 ^ 00000002
r98 = r76 ^ r18;    // 02060000 = 02020000 ^ 00040000
r99 = r76 ^ r72;    // 02030100 = 02020000 ^ 00010100
r100 = r77 ^ r2;    // 04040004 = 04040000 ^ 00000004
r101 = r78 ^ r64;   // 08080101 = 08080000 ^ 00000101
r102 = r79 ^ r75;   // 11110000 = 10100000 ^ 01010000
r103 = r80 ^ r79;   // 30100020 = 20000020 ^ 10100000
r104 = r81 ^ r4;    // 20002010 = 20002000 ^ 00000010
r105 = r81 ^ r30;   // 60002000 = 20002000 ^ 40000000
```

```
r106 = r82 ^ r5;    // 40400020 = 40400000 ^ 00000020
r107 = r82 ^ r14;   // 40404000 = 40400000 ^ 00004000
r108 = r82 ^ r64;   // 40400101 = 40400000 ^ 00000101
r109 = r83 ^ r23;   // 80800080 = 80000080 ^ 00800000
r110 = r83 ^ r74;   // 81000081 = 80000080 ^ 01000001

r111 = r84 ^ r74;   // 01010203 = 00010202 ^ 01000001
r112 = r85 ^ r77;   // 04060202 = 00020202 ^ 04040000
r113 = r87 ^ r78;   // 080c0404 = 00040404 ^ 08080000
r114 = r88 ^ r86;   // 00080c04 = 00080808 ^ 0000040c
r115 = r89 ^ r73;   // 10203010 = 10001010 ^ 00202000
r116 = r90 ^ r80;   // 20204060 = 00204040 ^ 20000020
r117 = r92 ^ r75;   // 01818080 = 00808080 ^ 01010000
r118 = r93 ^ r83;   // 80018180 = 00018100 ^ 80000080
r119 = r93 ^ r91;   // 0081c140 = 00018100 ^ 00804040
r120 = r94 ^ r89;   // 11000011 = 01001001 ^ 10001010
r121 = r95 ^ r85;   // 01020301 = 01000103 ^ 00020202
r122 = r96 ^ r65;   // 03010102 = 03010000 ^ 00000102
r123 = r97 ^ r67;   // 02020406 = 02020002 ^ 00000404
r124 = r98 ^ r85;   // 02040202 = 02060000 ^ 00020202
r125 = r99 ^ r0;    // 02030101 = 02030100 ^ 00000001
r126 = r100 ^ r68;  // 0404080c = 04040004 ^ 00000808
r127 = r100 ^ r86;  // 04040408 = 04040004 ^ 0000040c
r128 = r100 ^ r98;  // 06020004 = 04040004 ^ 02060000
r129 = r101 ^ r69;  // 08081111 = 08080101 ^ 00001010
r130 = r102 ^ r81;  // 31112000 = 11110000 ^ 20002000
r131 = r102 ^ r88;  // 11190808 = 11110000 ^ 00080808
r132 = r103 ^ r94;  // 31101021 = 30100020 ^ 01001001
r133 = r104 ^ r103; // 10102030 = 20002010 ^ 30100020
r134 = r105 ^ r14;  // 60006000 = 60002000 ^ 00004000
r135 = r106 ^ r73;  // 40602020 = 40400020 ^ 00202000
r136 = r108 ^ r71;  // 40408181 = 40400101 ^ 00008080
r137 = r109 ^ r64;  // 80800181 = 80800080 ^ 00000101
r138 = r110 ^ r92;  // 81808001 = 81000081 ^ 00808080
r139 = r110 ^ r107; // c1404081 = 81000081 ^ 40404000

r140 = r114 ^ r26;  // 04080c04 = 00080c04 ^ 04000000
r141 = r115 ^ r72;  // 10213110 = 10203010 ^ 00010100
r142 = r119 ^ r30;  // 4081c140 = 0081c140 ^ 40000000
r143 = r120 ^ r78;  // 19080011 = 11000011 ^ 08080000
r144 = r124 ^ r10;  // 02040602 = 02040202 ^ 00000400
r145 = r127 ^ r27;  // 0c040408 = 04040408 ^ 08000000
r146 = r128 ^ r9;   // 06020204 = 06020004 ^ 00000200
r147 = r129 ^ r3;   // 08081119 = 08081111 ^ 00000008
r148 = r130 ^ r115; // 21311010 = 31112000 ^ 10203010
```

```
r149 = r131 ^ r120; // 00190819 = 11190808 ^ 11000011
r150 = r133 ^ r64;  // 10102131 = 10102030 ^ 00000101
r151 = r134 ^ r90;  // 60202040 = 60006000 ^ 00204040
r152 = r134 ^ r106; // 20406020 = 60006000 ^ 40400020
r153 = r136 ^ r6;   // 404081c1 = 40408181 ^ 00000040
r154 = r136 ^ r110; // c1408100 = 40408181 ^ 81000081

r155 = r143 ^ r11;  // 19080811 = 19080011 ^ 00000800
r156 = r149 ^ r129; // 08111908 = 00190819 ^ 08081111
r157 = r154 ^ r142; // 81c14040 = c1408100 ^ 4081c140
```

G AES MixColumns with 97 XORs and depth 4

```
r64 = r8 ^ r0;      // 00000101 = 00000100 ^ 00000001
r65 = r10 ^ r2;     // 00000404 = 00000400 ^ 00000004
r66 = r14 ^ r6;     // 00004040 = 00004000 ^ 00000040
r67 = r15 ^ r7;     // 00008080 = 00008000 ^ 00000080
r68 = r16 ^ r8;     // 00010100 = 00010000 ^ 00000100
r69 = r17 ^ r9;     // 00020200 = 00020000 ^ 00000200
r70 = r19 ^ r11;    // 00080800 = 00080000 ^ 00000800
r71 = r19 ^ r18;    // 000c0000 = 00080000 ^ 00040000
r72 = r20 ^ r11;    // 00100800 = 00100000 ^ 00000800
r73 = r21 ^ r6;     // 00200040 = 00200000 ^ 00000040
r74 = r21 ^ r13;    // 00202000 = 00200000 ^ 00002000
r75 = r22 ^ r13;    // 00402000 = 00400000 ^ 00002000
r76 = r24 ^ r0;     // 01000001 = 01000000 ^ 00000001
r77 = r24 ^ r16;    // 01010000 = 01000000 ^ 00010000
r78 = r25 ^ r0;     // 02000001 = 02000000 ^ 00000001
r79 = r25 ^ r1;     // 02000002 = 02000000 ^ 00000002
r80 = r26 ^ r18;    // 04040000 = 04000000 ^ 00040000
r81 = r26 ^ r25;    // 06000000 = 04000000 ^ 02000000
r82 = r27 ^ r3;     // 08000008 = 08000000 ^ 00000008
r83 = r28 ^ r4;     // 10000010 = 10000000 ^ 00000010
r84 = r28 ^ r20;    // 10100000 = 10000000 ^ 00100000
r85 = r29 ^ r5;     // 20000020 = 20000000 ^ 00000020
r86 = r30 ^ r22;    // 40400000 = 40000000 ^ 00400000
r87 = r31 ^ r23;    // 80800000 = 80000000 ^ 00800000

r88 = r64 ^ r4;     // 00000111 = 00000101 ^ 00000010
r89 = r64 ^ r24;    // 01000101 = 00000101 ^ 01000000
r90 = r65 ^ r26;    // 04000404 = 00000404 ^ 04000000
r91 = r65 ^ r27;    // 08000404 = 00000404 ^ 08000000
r92 = r66 ^ r30;    // 40004040 = 00004040 ^ 40000000
r93 = r67 ^ r23;    // 00808080 = 00008080 ^ 00800000
```

```
r94 = r67 ^ r64;    // 00008181 = 00008080 ^ 00000101
r95 = r68 ^ r17;    // 00030100 = 00010100 ^ 00020000
r96 = r68 ^ r24;    // 01010100 = 00010100 ^ 01000000
r97 = r68 ^ r31;    // 80010100 = 00010100 ^ 80000000
r98 = r69 ^ r2;     // 00020204 = 00020200 ^ 00000004
r99 = r70 ^ r3;     // 00080808 = 00080800 ^ 00000008
r100 = r70 ^ r27;   // 08080800 = 00080800 ^ 08000000
r101 = r74 ^ r5;    // 00202020 = 00202000 ^ 00000020
r102 = r74 ^ r68;   // 00212100 = 00202000 ^ 00010100
r103 = r77 ^ r66;   // 01014040 = 01010000 ^ 00004040
r104 = r79 ^ r9;    // 02000202 = 02000002 ^ 00000200
r105 = r80 ^ r1;    // 04040002 = 04040000 ^ 00000002
r106 = r80 ^ r3;    // 04040008 = 04040000 ^ 00000008
r107 = r82 ^ r12;   // 08001008 = 08000008 ^ 00001000
r108 = r83 ^ r12;   // 10001010 = 10000010 ^ 00001000
r109 = r83 ^ r76;   // 11000011 = 10000010 ^ 01000001
r110 = r84 ^ r5;    // 10100020 = 10100000 ^ 00000020
r111 = r84 ^ r77;   // 11110000 = 10100000 ^ 01010000
r112 = r85 ^ r14;   // 20004020 = 20000020 ^ 00004000
r113 = r86 ^ r6;    // 40400040 = 40400000 ^ 00000040
r114 = r86 ^ r14;   // 40404000 = 40400000 ^ 00004000
r115 = r87 ^ r15;   // 80808000 = 80800000 ^ 00008000
r116 = r87 ^ r22;   // 80c00000 = 80800000 ^ 00400000

r117 = r88 ^ r13;   // 00002111 = 00000111 ^ 00002000
r118 = r88 ^ r19;   // 00080111 = 00000111 ^ 00080000
r119 = r89 ^ r69;   // 01020301 = 01000101 ^ 00020200
r120 = r90 ^ r70;   // 04080c04 = 04000404 ^ 00080800
r121 = r91 ^ r71;   // 080c0404 = 08000404 ^ 000c0000
r122 = r93 ^ r77;   // 01818080 = 00808080 ^ 01010000
r123 = r95 ^ r78;   // 02030101 = 00030100 ^ 02000001
r124 = r96 ^ r79;   // 03010102 = 01010100 ^ 02000002
r125 = r97 ^ r67;   // 80018180 = 80010100 ^ 00008080
r126 = r98 ^ r81;   // 06020204 = 00020204 ^ 06000000
r127 = r101 ^ r86;  // 40602020 = 00202020 ^ 40400000
r128 = r101 ^ r92;  // 40206060 = 00202020 ^ 40004040
r129 = r104 ^ r10;  // 02000602 = 02000202 ^ 00000400
r130 = r104 ^ r77;  // 03010202 = 02000202 ^ 01010000
r131 = r105 ^ r69;  // 04060202 = 04040002 ^ 00020200
r132 = r106 ^ r2;   // 0404000c = 04040008 ^ 00000004
r133 = r107 ^ r68;  // 08011108 = 08001008 ^ 00010100
r134 = r108 ^ r29;  // 30001010 = 10001010 ^ 20000000
r135 = r108 ^ r102; // 10213110 = 10001010 ^ 00212100
r136 = r109 ^ r100; // 19080811 = 11000011 ^ 08080800
r137 = r110 ^ r109; // 01100031 = 10100020 ^ 11000011
```

```
r138 = r111 ^ r21;  // 11310000 = 11110000 ^ 00200000
r139 = r111 ^ r99;  // 11190808 = 11110000 ^ 00080808
r140 = r112 ^ r73;  // 20204060 = 20004020 ^ 00200040
r141 = r112 ^ r75;  // 20406020 = 20004020 ^ 00402000
r142 = r113 ^ r94;  // 404081c1 = 40400040 ^ 00008181
r143 = r114 ^ r93;  // 40c0c080 = 40404000 ^ 00808080
r144 = r115 ^ r76;  // 81808001 = 80808000 ^ 01000001
r145 = r115 ^ r94;  // 80800181 = 80808000 ^ 00008181
r146 = r115 ^ r97;  // 00818100 = 80808000 ^ 80010100
r147 = r116 ^ r103; // 81c14040 = 80c00000 ^ 01014040

r148 = r117 ^ r110; // 10102131 = 00002111 ^ 10100020
r149 = r118 ^ r107; // 08081119 = 00080111 ^ 08001008
r150 = r128 ^ r112; // 60202040 = 40206060 ^ 20004020
r151 = r129 ^ r18;  // 02040602 = 02000602 ^ 00040000
r152 = r129 ^ r98;  // 02020406 = 02000602 ^ 00020204
r153 = r130 ^ r78;  // 01010203 = 03010202 ^ 02000001
r154 = r132 ^ r11;  // 0404080c = 0404000c ^ 00000800
r155 = r132 ^ r91;  // 0c040408 = 0404000c ^ 08000404
r156 = r133 ^ r72;  // 08111908 = 08011108 ^ 00100800
r157 = r137 ^ r134; // 31101021 = 01100031 ^ 30001010
r158 = r138 ^ r134; // 21311010 = 11310000 ^ 30001010
r159 = r144 ^ r143; // c1404081 = 81808001 ^ 40c0c080
r160 = r146 ^ r92;  // 4081c140 = 00818100 ^ 40004040
```