Optimizing linear maps modulo 2

Daniel J. Bernstein *

Department of Computer Science (MC 152)
The University of Illinois at Chicago
Chicago, IL 60607–7053
djb@cr.yp.to

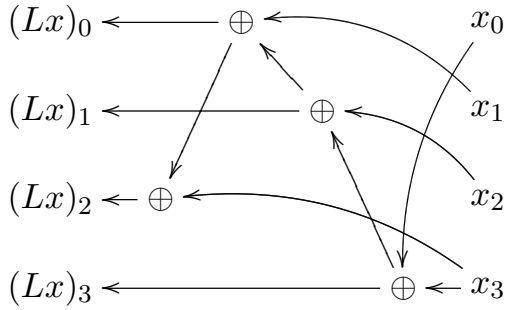
Abstract. This paper introduces and analyzes an algorithm to compile a series of exclusive-or operations. The compiled series is quite efficient, almost always beating the so-called "Four Russians" approach, and uses no temporary storage beyond its outputs. The algorithm is reasonably fast and surprisingly simple.

1 Introduction

Consider the 4-bit-to-4-bit $\mathbf{F}_2$-linear function $L : \mathbf{F}_2^4 \to \mathbf{F}_2^4$ defined by $L(x_0, x_1, x_2, x_3) = (x_0 \oplus x_1 \oplus x_2 \oplus x_3, x_0 \oplus x_2 \oplus x_3, x_0 \oplus x_1 \oplus x_2, x_0 \oplus x_3)$; i.e.,

$$L \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1\,1\,1\,1 \\ 1\,0\,1\,1 \\ 1\,1\,1\,0 \\ 1\,0\,0\,1 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix}.$$

Evaluating L directly from its definition takes 8 bit xors: xor x_0 and x_1, then xor x_2, then xor x_3, obtaining $(Lx)_0$; xor x_0 and x_2, then xor x_3, obtaining $(Lx)_1$; xor x_0 and x_1, then xor x_2, obtaining $(Lx)_2$; finally xor x_0 and x_3, obtaining $(Lx)_3$. However, this computation has several obvious redundancies, and a closer look shows that L can be computed in just 4 xors:



* Permanent ID of this document: e5c3095f5c423e2fe19fa072e23bd5d7. Date of this document: 2009.10.05. This work was supported by the National Science Foundation under grant ITR–0716498.

Similarly, if x_0, x_1, x_2, x_3 are (e.g.) 128-bit vector registers, then the 8 vector xors in $(x_0 \oplus x_1 \oplus x_2 \oplus x_3, x_0 \oplus x_2 \oplus x_3, x_0 \oplus x_1 \oplus x_2, x_0 \oplus x_3)$ can be replaced by 4 vector xors.

This paper presents an algorithm that, given the matrix for a p-bit-to-q-bit $\mathbf{F}_2$-linear function $L : \mathbf{F}_2^p \to \mathbf{F}_2^q$, prints code to compute L. The resulting code has several attractive features:

- Minimal input loads: The code reads each of the p input bits exactly once, in order, except that it skips unused bits.
- Minimal storage: The code writes only to the q output bits and does not need any extra storage.
- Low algebraic complexity: Heuristics for large p, q suggest that for most functions L the code uses approximately $pq/(\lg q - \lg \lg q)$ xors. Computer experiments support these heuristics.
- Low two-operand complexity: On two-operand platforms (platforms that allow "set a to $a \oplus b$" but not "set a to $b \oplus c$"), the code requires only p extra copies.

The compilation algorithm per se is reasonably fast: for each line of code that it prints, it uses a logarithmic number of vector comparisons and one vector xor. The algorithm is also surprisingly simple.

Applications. Bitsliced binary-finite-field arithmetic has recently set software speed records for public-key cryptography; see [6]. The software described in [6] spends most of its time in short hand-optimized linear computations such as

```
vec h0 = h[i];
vec h12 = h[i + n] ^ h[i + 2 * n];
vec h34 = h[i + 3 * n] ^ h[i + 4 * n];
vec h56 = h[i + 5 * n] ^ h[i + 6 * n];
vec h7 = h[i + 7 * n];
vec x0 = u[i];
vec x12 = u[i + n] ^ u[i + 2 * n];
vec x3 = u[i + 3 * n];
vec h1 = h12 ^ h0 ^ u2[i];
vec b = h34 ^ h12 ^ u2[i + n];
vec c = h34 ^ h56 ^ u3[i];
vec h6 = h7 ^ h56 ^ u3[i + n];
h[i + n] = h1;
h[i + 2 * n] = b ^ h0 ^ x0;
h[i + 3 * n] = c ^ h1 ^ x12 ^ x0 ^ u4[i];
```

```
h[i + 4 * n] = h6 ^ b ^ x12 ^ x3 ^ u4[i + n];
h[i + 5 * n] = h7 ^ c ^ x3;
h[i + 6 * n] = h6;
```

where each ^ is a 128-bit vector xor. Unfortunately, hand optimization is time-consuming even for small examples.

This paper's algorithm can be used as a baseline compilation technique for all $\mathbf{F}_2$-linear functions L. In many cases the algorithm is competitive with the best hand-optimized code, saving time for the programmer. The algorithm can also be applied to extremely large examples: for example, I have used it to generate fast unrolled bitsliced normal-basis conversion functions for the cryptanalytic computation described in [4], converting a 131×131 basis-conversion matrix into a sequence of 3380 xors and converting a 163×163 matrix into a sequence of 5078 xors.

One should not think of this algorithm as magically discovering every fast linear computation in the literature. For example, if $\varphi \in \mathbf{F}_2[x]$ is a polynomial of degree n, then the $\mathbf{F}_2$-linear function $f \mapsto f^2$ on $\mathbf{F}_2[x]/\varphi$, with basis $1, x, \ldots, x^{n-1}$, can be computed with $n^{1+o(1)}$ xors by fast-multiplication techniques. For most choices of φ, feeding the same linear function to this paper's algorithm would use many more xors, at least for large n.

2 The algorithm

An efficient multi-scalar-multiplication method appears in [8, Section 4] with credit to Bos and Coster. To compute $n_0x_0 + n_1x_1 + n_2x_2 + \cdots$, where $n_0 \geq n_1 \geq n_2 \geq \cdots \geq 0$, Bos and Coster recursively compute $(n_0 - n_1)x_0 + n_1(x_0 + x_1) + n_2x_2 + \cdots$. They use a more complicated step in the case that n_0 is much larger than n_1, since subtracting n_1 from n_0 is then ineffective at reducing n_0, although this case rarely occurs for random scalars.

A transposed version of the Bos–Coster method computes multiples $n_0x, n_1x, n_2x, \ldots$, where $n_0 \geq n_1 \geq n_2 \geq \cdots$, by recursively computing $(n_0 - n_1)x, n_1x, n_2x, \ldots$ and then adding output 1 into output 0. (Coster gives credit for this algorithm to Brun; however, I do not see any evidence that Brun considered the construction of addition chains.)

This paper's algorithm has the same outline but uses xor instead of subtraction: it computes several dot products $L_0x, L_1x, L_2x, \ldots$, where $L_0 \geq L_1 \geq L_2 \geq \cdots$, by recursively computing $(L_0 \oplus L_1)x, L_1x, L_2x, \ldots$ and then xoring output 1 into output 0. The case that L_0 is much larger than L_1 (specifically, that it has its most significant bit at a different

position) is much more common here, and requires different treatment: the algorithm reduces L_0 by simply clearing its most significant bit.

Details. The input to the algorithm is a $q \times p$ matrix of bits, viewed as a sequence of q rows $L_0, L_1, \ldots, L_{q-1} \in \mathbf{F}_2^p$, where p and q are non-negative integers. The p bits $L_j[0], L_j[1], \ldots, L_j[p-1]$ of L_j specify the linear function $x_0, x_1, \ldots, x_{p-1} \mapsto L_j[0]x_0 \oplus L_j[1]x_1 \oplus \cdots \oplus L_j[p-1]x_{p-1}$; the algorithm produces code that computes these q linear functions. The algorithm works as follows:

- If $q = 0$: Stop. (There is nothing to compute.)
- If $p = 0$: Generate code that sets each output bit to 0. Stop.
- Find $j \in \{0, 1, \ldots, q-1\}$ that maximizes L_j in reverse lexicographic order (i.e., maximizes $L_j[p-1]$; secondarily maximizes $L_j[p-2]$; etc.).
- If $L_j[p-1] = 0$: Define L'_k as $(L_k[0], \ldots, L_k[p-2])$ for each $k \in \{0, 1, \ldots, q-1\}$. Recursively apply the algorithm to $L'_0, L'_1, \ldots, L'_{q-1}$. Stop. (All $L_k[p-1]$ are 0; i.e., x_{p-1} is unused.)
- If $q \geq 2$: Find $i \in \{0, 1, \ldots, q-1\} - \{j\}$ that maximizes L_i in reverse lexicographic order. If $L_i[p-1] = 1$: Define $L'_k = L_k$ for each $k \in \{0, 1, \ldots, q-1\}$, except that $L'_j = L_j \oplus L_i$. Recursively apply the algorithm to $L'_0, L'_1, \ldots, L'_{q-1}$. Generate code that xors output bit i into output bit j. Stop.
- Define $L'_k = L_k$ for each $k \in \{0, 1, \ldots, q-1\}$, except that $L'_j[p-1] = 0$. Recursively apply the algorithm to $L'_0, L'_1, \ldots, L'_{q-1}$. Generate code that xors *input* bit $p-1$ into output bit j. Stop.

The recursive steps in the algorithm can and should store L' on top of L, eliminating the space for L' and almost all of the time to compute L'. If the output code is generated in reverse order then the recursive steps can be replaced by tail-recursive steps, eliminating all other storage. If the rows, or pointers to the rows, are stored in a heap then identifying the two largest rows takes only a logarithmic number of row comparisons.

The code generated by this algorithm starts with code to set each output bit to 0. Except in the extreme case $L_j = 0$, the code to set output bit j to 0 can and should be merged with a subsequent xor involving output bit j, turning the xor into a copy. One can, as an option for three-operand architectures, merge the copies with further xors, although in some cases this is incompatible with reading each input exactly once.

A transposed version of the same algorithm writes each of the output bits exactly once, in order, and otherwise works entirely within the p input bits.

Example. Consider the four rows appearing at the start of this paper: $L_0 = (1\,1\,1\,1)$; $L_1 = (1\,0\,1\,1)$; $L_2 = (1\,1\,1\,0)$; $L_3 = (1\,0\,0\,1)$.

The largest row in reverse lexicographic order is $L_0 = (1\,1\,1\,1)$, and the second largest is $L_1 = (1\,0\,1\,1)$. The last instruction in the compiled code is to xor output bit 1 into output bit 0. The goal of the previous instructions is to compute $L'_0 = (0\,1\,0\,0)$; $L'_1 = (1\,0\,1\,1)$; $L'_2 = (1\,1\,1\,0)$; $L'_3 = (1\,0\,0\,1)$. Here $L'_0 = L_0 \oplus L_1$.

The largest remaining row is $L'_1 = (1\,0\,1\,1)$, followed by $L'_3 = (1\,0\,0\,1)$. The second-to-last instruction in the compiled code is to xor output bit 3 into output bit 1. The goal of the previous instructions is to compute $L''_0 = (0\,1\,0\,0)$; $L''_1 = (0\,0\,1\,0)$; $L''_2 = (1\,1\,1\,0)$; $L''_3 = (1\,0\,0\,1)$. Here $L''_1 = L'_1 \oplus L'_3$.

The largest remaining row is $L''_3 = (1\,0\,0\,1)$, followed by $L''_2 = (1\,1\,1\,0)$. The third-to-last instruction in the compiled code is to xor *input* bit 3 into output bit 3. The goal of the previous instructions is to compute $L'''_0 = (0\,1\,0)$; $L'''_1 = (0\,0\,1)$; $L'''_2 = (1\,1\,1)$; $L'''_3 = (1\,0\,0)$.

The largest remaining row is $L'''_2 = (1\,1\,1)$, followed by $L'''_1 = (0\,0\,1)$. The fourth-to-last instruction in the compiled code is to xor output bit 1 into output bit 2. Et cetera. The algorithm finishes with the following sequence of instructions:

- Store 0 in output bit 0.
- Store 0 in output bit 1.
- Store 0 in output bit 2.
- Store 0 in output bit 3.
- Xor input bit 0 into output bit 3.
- Xor output bit 3 into output bit 2.
- Xor input bit 1 into output bit 0.
- Xor output bit 0 into output bit 2.
- Xor input bit 2 into output bit 1.
- Xor output bit 1 into output bit 2.
- Xor input bit 3 into output bit 3.
- Xor output bit 3 into output bit 1.
- Xor output bit 1 into output bit 0.

Eliminating 0 produces the following sequence of instructions:

- Copy input bit 0 into output bit 3.
- Copy output bit 3 into output bit 2.
- Copy input bit 1 into output bit 0.
- Xor output bit 0 into output bit 2.
- Copy input bit 2 into output bit 1.

- Xor output bit 1 into output bit 2.
- Xor input bit 3 into output bit 3.
- Xor output bit 3 into output bit 1.
- Xor output bit 1 into output bit 0.

Optional copy elimination produces the following sequence of three-operand instructions:

- Xor input bit 1 and input bit 0, producing output bit 2.
- Xor input bit 2 into output bit 2.
- Xor input bit 3 to input bit 0, producing output bit 3.
- Xor output bit 3 to input bit 2, producing output bit 1.
- Xor output bit 1 to input bit 1, producing output bit 0.

3 Experimental results

A straightforward implementation of this paper's algorithm, including zero elimination (but not copy elimination), is available from http://binary.cr.yp.to/linearmod2.html. Running

```
wget http://binary.cr.yp.to/linearmod2/sort1.cpp
g++ -o sort1 sort1.cpp
echo 1111101111101001 | ./sort1 4 4 > test.c
gcc -o test test.c
./test
```

applies the algorithm to the function L shown at the beginning of this paper; creates a test program `test.c` for the resulting code; and prints 5, indicating that all 4 outputs were computed using a total of 5 xors.

More generally, `./sort1` p q reads $q \times p$ input bits, applies this paper's algorithm, and prints a test program that includes the resulting code. The program packs each matrix row into four `unsigned long long` variables, so it is limited to $p \in \{0, 1, 2, \ldots, 256\}$, but it allows any $q \in \{1, 2, 3, \ldots\}$ that fits into memory. The test program prints the algebraic complexity of the code, i.e., the number of xors. The test program also checks that the code computes the desired outputs; if this check fails, the program prints 999999999.

Table 3.1 shows the average algebraic complexity, divided by pq, of this code for 10000 random $q \times p$ matrices obtained from `/dev/urandom`, the Linux cryptographic random number generator. Table 3.2 shows the standard deviation of the algebraic complexity. For example, for $(p, q) = (64, 128)$, the two tables have entries 0.1922 and 0.0011 respectively; this

	$p=1$	$p=2$	$p=4$	$p=8$	$p=16$	$p=32$	$p=64$	$p=128$	$p=256$
$q=1$	0.00000	0.12535	0.26980	0.37604	0.43864	0.47102	0.48548	0.49192	0.49588
$q=2$	0.00000	0.10705	0.23604	0.34278	0.39923	0.42865	0.44310	0.45128	0.45476
$q=4$	0.00000	0.08594	0.20378	0.30151	0.35778	0.38589	0.40016	0.40726	0.41060
$q=8$	0.00000	0.05613	0.16508	0.25995	0.31452	0.34270	0.35636	0.36294	0.36655
$q=16$	0.00000	0.03096	0.12111	0.21510	0.26961	0.29670	0.31020	0.31687	0.32030
$q=32$	0.00000	0.01562	0.07705	0.17217	0.22477	0.25136	0.26477	0.27134	0.27468
$q=64$	0.00000	0.00781	0.04239	0.13501	0.18583	0.21194	0.22499	0.23148	0.23475
$q=128$	0.00000	0.00391	0.02148	0.10544	0.15372	0.17938	0.19223	0.19864	0.20186
$q=256$	0.00000	0.00195	0.01074	0.07889	0.12824	0.15304	0.16575	0.17209	0.17527
$q=512$	0.00000	0.00098	0.00537	0.05251	0.10657	0.13178	0.14431	0.15062	0.15377
$q=1024$	0.00000	0.00049	0.00269	0.02962	0.08683	0.11403	0.12686	0.13315	0.13629
$q=2048$	0.00000	0.00024	0.00134	0.01507	0.07298	0.10034	0.11248	0.11878	0.12192
$q=4096$	0.00000	0.00012	0.00067	0.00754	0.06516	0.08859	0.10062	0.10683	0.10998

Table 3.1. Average number of xors for 10000 random $q \times p$ matrices. All xor chains were produced by this paper's algorithm with zero elimination.

algorithm evaluated 10000 random 64-bit-to-128-bit linear maps using approximately $0.1922 \cdot 128 \cdot 64 \approx 1575$ xors on average, with standard deviation approximately $0.0011 \cdot 128 \cdot 64 \approx 9$.

Heuristic analysis. To understand the performance of this algorithm for $q = 128$, assume that exactly 64 of the 128 input rows involve the most significant input bit x_{p-1}. Performing 64 xors of adjacent rows typically produces 6 or more clear bits in each new row:

- At most 1 of the new rows can start with x_{p-2}: scanning through the sorted rows produces only one transition from $\ldots, 0, 1$ to $\ldots, 1, 1$.
- At most 2 of the new rows can start with x_{p-3}: one for the transition from $\ldots, 0, 1, 1$ to $\ldots, 1, 1, 1$, and one for the transition from $\ldots, 0, 0, 1$ to $\ldots, 1, 0, 1$.
- At most 4 of the new rows can start with x_{p-4}.
- At most 8 of the new rows can start with x_{p-5}.
- At most 16 of the new rows can start with x_{p-6}.
- The remaining 33 new rows must start with x_{p-7} or beyond.

At this point there are about 33 rows starting with x_{p-2}, and the next 33 xors typically produce 5 or more clear bits. After a few more iterations the algorithm settles down on a steady state consuming fewer than 30 xors for each bit.

For general q, the steady state appears to have approximately q/c rows starting with the most significant bit, where $c \in \mathbf{R}$ satisfies $2^c = q/c$. The

	$p=1$	$p=2$	$p=4$	$p=8$	$p=16$	$p=32$	$p=64$	$p=128$	$p=256$
$q=1$	0.00000	0.21671	0.22778	0.17718	0.12495	0.08752	0.06241	0.04426	0.03132
$q=2$	0.00000	0.12370	0.13096	0.10243	0.07333	0.05152	0.03644	0.02538	0.01796
$q=4$	0.00000	0.05794	0.07686	0.06064	0.04254	0.03003	0.02127	0.01500	0.01065
$q=8$	0.00000	0.01891	0.04178	0.03413	0.02456	0.01708	0.01200	0.00846	0.00593
$q=16$	0.00000	0.00300	0.01972	0.01845	0.01307	0.00914	0.00633	0.00447	0.00320
$q=32$	0.00000	0.00016	0.00720	0.01008	0.00699	0.00486	0.00334	0.00234	0.00167
$q=64$	0.00000	0.00000	0.00148	0.00610	0.00395	0.00266	0.00187	0.00131	0.00091
$q=128$	0.00000	0.00000	0.00010	0.00399	0.00231	0.00157	0.00106	0.00073	0.00052
$q=256$	0.00000	0.00000	0.00000	0.00241	0.00148	0.00092	0.00062	0.00043	0.00030
$q=512$	0.00000	0.00000	0.00000	0.00107	0.00087	0.00056	0.00037	0.00026	0.00018
$q=1024$	0.00000	0.00000	0.00000	0.00024	0.00043	0.00033	0.00023	0.00016	0.00011
$q=2048$	0.00000	0.00000	0.00000	0.00002	0.00026	0.00022	0.00014	0.00010	0.00007
$q=4096$	0.00000	0.00000	0.00000	0.00000	0.00019	0.00015	0.00009	0.00006	0.00004

Table 3.2. Standard deviation of the number of xors for the same $q \times p$ matrices used in Table 3.1. All xor chains were produced by this paper's algorithm with zero elimination.

algorithm thus uses approximately $pq/(\lg q - \lg\lg q)$ xors. This heuristic also suggests that the algorithm uses $O(pq)$ vector comparisons; perhaps this can be proven.

Modifications. One can *completely* sort the input rows, compute xors of adjacent rows having the first bit set, sort the rows again (by sorting the xors and merging with the other rows), handle all rows having the second bit set, etc. This sorting does not significantly slow down the algorithm, and might even speed up the algorithm, for example by allowing non-comparison-based sorting algorithms such as radix sort.

Perhaps more importantly, this sorting allows several variations in how the first-bit-set rows are handled:

- The algorithm chooses the second-largest row as a source to xor into the largest row. One can instead choose, e.g., the row that produces the smallest xor; sorting allows this row to be located more quickly. There are many plausible heuristics here.
- The algorithm reduces the first-bit-set rows in order, ending with the smallest of these rows. One can instead choose to reduce the rows from both ends, finishing with an intermediate row, such as a row that shares many leading bits with a non-first-bit-set row.
- One can dynamically choose an order to reduce the first-bit-set rows, for example tracing a low-Hamming-distance spanning tree.

Note that some of these changes also increase parallelism.

One can also permute inputs at the start of, or during, the algorithm. This produces different—and perhaps faster—computations of the same outputs, at the expense of the feature that the inputs are read in order. Perhaps there is a heuristic that chooses good permutations.

4 Comparison to input partitioning

"Input partitioning" is the following classic method to compute q sums of subsequences of $x_0, x_1, \ldots, x_{p-1}$:

- Partition $\{0, 1, \ldots, p-1\}$ into p/c parts of size c. This description assumes for simplicity that p is a multiple of c.
- For each part, and for each nonempty subset S of the part, compute the subset sum $\sum_{i \in S} x_i$. This takes $(p/c)(2^c - c - 1)$ additions; note that each new subset sum requires only one addition.
- Compute each of the output sums as a sum of (at worst) p/c subset sums. This takes $q(p/c - 1)$ additions.

A standard analysis chooses $c \in \lg q - \lg\lg q - \lg\log 2 + o(1)$ and produces a bound of $pq(1 + (1/\log 2 + \lg\log 2 + o(1))/\lg q)/(\lg q - \lg\lg q)$ additions. The total number of additions is therefore asymptotically $(1+o(1))q^2/\lg q$ in the typical case $p = q$.

Input partitioning for vectors of Boolean truth values was introduced by Lupanov in [11]. Obviously input partitioning also works for vectors of integers, vectors of integers modulo 2, etc. Fifteen years later the same construction appeared in [3, "Lemma (M. Kronrod)"] as a tool for Boolean matrix multiplication. Input partitioning is often called the "Four-Russians algorithm" by people who

- see that [3] was written by Arlazarov, Dinic, Kronrod, and Faradžev, all of which sound like Russian names to the ignorant observer;
- have not actually read [3], and are thus unaware that the method is credited to Kronrod alone; and
- are unaware of previous work such as [11].

See, e.g., [18], [12], [5], and [7].

Algorithm comparison. This paper's algorithm produces xor chains that, for $p = q$, have length $(1 + o(1))q^2/\lg q$. Input partitioning also produces chains of length $(1 + o(1))q^2/\lg q$, but a closer look at the $o(1)$ suggests that this paper's chains are shorter by a factor of nearly $1+1/\lg q$. See below for a detailed analysis of the illustrative case $(p, q) = (64, 128)$.

Input partitioning also appears to be somewhat less register-friendly than this paper's algorithm. The number of subset sums computed is typically on the scale of $pq/(\lg q)^2$, nearly quadratic in the total number of inputs and outputs. One can xor each subset sum into all relevant output registers immediately after computing the sum, but the active sums still require several temporary registers.

Analysis for $p = 64$ and $q = 128$. The following analysis assumes that input partitioning is combined with dead-expression removal: a subset sum that is not actually used will not be computed. The number of subsets computed is at least the number of subsets used *in the outputs*; it can be larger if a subset unused in the outputs is used to compute another subset.

Consider partitioning 64 inputs into 16 4-bit parts, and computing 128 outputs from xors of subsets of the parts. One expects each of the 128 outputs to involve $16(1-1/2^4)$ nonzero subsets on average, and therefore to consume at least $16(1-1/2^4)-1$ xors on average. Each part has 2^4-4-1 subsets of size 2 or larger; each subset is needed in the outputs with probability $1-(1-1/2^4)^{128}$, and therefore consumes on average at least $1-(1-1/2^4)^{128}$ xors. The total algebraic complexity is at least $128(16(1-1/2^4)-1)+16(2^4-4-1)(1-(1-1/2^4)^{128}) \approx 1967.95$ xors on average.

A better strategy is to partition 64 inputs into 10 6-bit parts and 1 4-bit part. The algebraic complexity is, by a similar analysis, at least $128(10(1-1/2^6)+1(1-1/2^4)-1)+10(2^6-6-1)(1-(1-1/2^6)^{128})+1(2^4-4-1)(1-(1-1/2^4)^{128}) \approx 1757.06$ xors on average.

One can consider other strategies, including "fractional" possibilities such as partitioning 64 inputs into 4 6-bit parts and 8 5-bit parts, but none of these strategies seem to come close to the 1575 xors used by the algorithm introduced in this paper. The gap is even larger on two-operand architectures.

5 Comparison to greedy additive CSE

Input partitioning chooses a partition, and computes sums of subsets of parts, without paying attention to the structure of the target sums. Dead-expression removal eliminates unused subset sums but does not address the underlying problem: namely, most of these subset sums are less useful than other subset sums would have been.

Intuition suggests that one can do better by starting with the two-input sum that is in fact most commonly used. By repeating this idea one

obtains an additive (i.e., commutative and associative) variant of greedy common-subexpression elimination (CSE). The complete algorithm is as follows:

- If each output involves at most one input, stop.
- Select the pair of inputs whose sum appears most frequently in the outputs. If there is more than one such pair, choose the first.
- Generate code to compute the sum of this pair of inputs.
- Use this sum as an additional input; shorten the outputs accordingly.
- Repeat.

For example, consider again the problem of computing $x_0 \oplus x_1 \oplus x_2 \oplus x_3, x_0 \oplus x_2 \oplus x_3, x_0 \oplus x_1 \oplus x_2, x_0 \oplus x_3$. This algorithm observes that $x_0 \oplus x_2$ is used three times, and that no other input xor is used more than three times. It then computes $x_4 = x_0 \oplus x_2$ and recursively solves the problem of computing $x_4 \oplus x_1 \oplus x_3, x_4 \oplus x_3, x_4 \oplus x_1, x_0 \oplus x_3$. Next the algorithm computes $x_5 = x_1 \oplus x_4$ and recursively solves the problem of computing $x_5 \oplus x_3, x_4 \oplus x_3, x_5, x_0 \oplus x_3$. The algorithm continues in this way and ends up using a total of 5 xors.

Paar in [14] illustrated this algorithm using another example: the 7×7 matrix

$$\begin{pmatrix} 1&0&0&1&1&1&1 \\ 1&1&0&1&0&0&0 \\ 1&1&1&0&1&0&0 \\ 1&1&1&1&0&1&0 \\ 1&1&1&1&1&0&1 \\ 0&1&1&1&1&1&0 \\ 0&0&1&1&1&1&1 \end{pmatrix}$$

for a constant-multiplication problem in a field of size 128. Paar presented a 16-xor computation of this matrix obtained by greedy additive CSE.

Recall that, when the maximum frequency is achieved by several sums simultaneously, greedy additive CSE makes an arbitrary choice of sum. Paar considered a variant of greedy additive CSE that combinatorially explores every choice, and as a result presented a 14-xor computation of the same 7×7 matrix shown above. For comparison, this paper's algorithm also uses 14 xors for this matrix.

Even without this combinatorial exploration, greedy additive CSE can be extremely time-consuming. Paar's fastest algorithm takes $q^{6+o(1)}$ vector operations for a typical $q \times q$ matrix: the number of inputs quickly grows from q to $q^{2+o(1)}$, and the algorithm ends up considering $q^{4+o(1)}$ input pairs for each of the $q^{2+o(1)}$ lines of code printed. On the other hand,

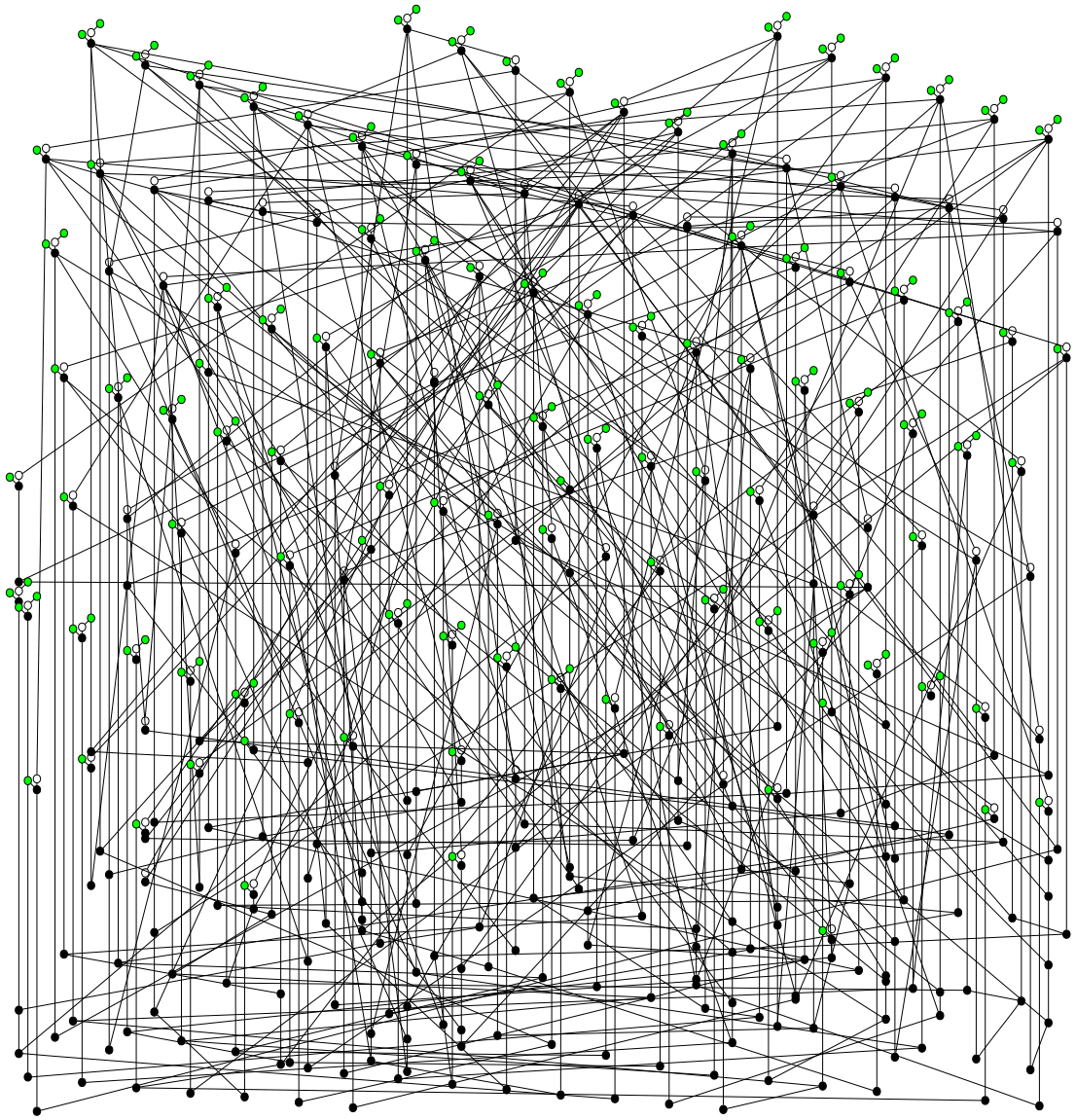


Fig. 5.1. Code produced by greedy additive CSE for a 41×41 basis-conversion matrix. Each white node has one edge leading down to it and copies the edge's source. Each black node has two edges leading down to it and computes the xor of the two sources. Each green node is an input.

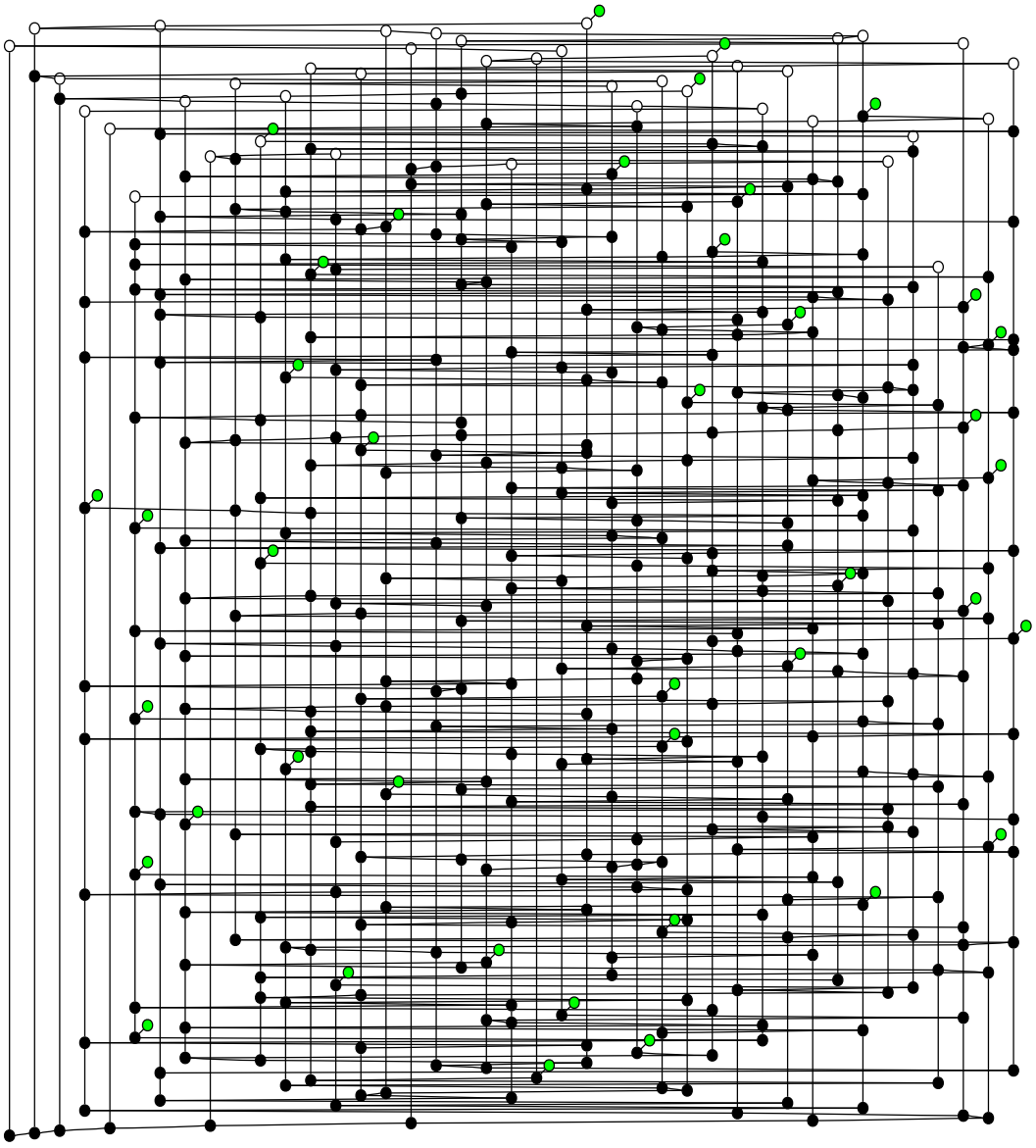


Fig. 5.2. Code produced by this paper's algorithm for a 41×41 basis-conversion matrix. Each white node has one edge leading down to it and copies the edge's source. Each black node has two edges leading down to it and computes the xor of the two sources. Each green node is an input.

one can reduce the exponent of q in greedy additive CSE by maintaining a database of active input-pair frequencies and updating the database whenever an output is shortened.

Algorithm comparison. Experiments indicate that greedy additive CSE has one advantage over this paper's algorithm: the output code has, on average, lower algebraic complexity. However, greedy additive CSE also has some disadvantages: the code uses much more storage, and has much higher two-operand complexity.

Figures 5.1 and 5.2 visually illustrate these effects for a 41×41 matrix, specifically the matrix converting $\mathbf{F}_2[x]/(x^{41} + x^3 + 1)$ from polynomial basis $1, x, x^2, \ldots, x^{40}$ to normal basis $x+1, (x+1)^2, \ldots, (x+1)^{2^{40}}$. Greedy additive CSE generates the 309 xors and 132 copies shown in Figure 5.1, fitting into 117 registers indicated by vertical lines. This paper's algorithm generates the 363 xors and 41 copies shown in Figure 5.2, fitting into 41 registers indicated by vertical lines.

6 Other algorithms

Recall that input partitioning has algebraic complexity asymptotically $(1 + o(1))q^2/\lg q$. Combining input partitioning with "output clumping" reduces the algebraic complexity to $(1 + o(1))q^2/\lg(q^2)$, saving a factor of $2 + o(1)$.

Output clumping was introduced by Nechiporuk and pushed much further by Pippenger; see [15], [16], and [17]. Pippenger's addition chains are, for a wide variety of problems, within a factor of $1 + o(1)$ of the best possible solutions, as measured by algebraic complexity. See generally [16, Section 2] and [17, Section 2].

The asymptotics show that Pippenger's chains are shorter than this paper's chains for sufficiently large p, q. However, preliminary experiments indicate that this paper's xor chains are shorter than Pippenger's addition chains for $p, q \leq 256$. Perhaps there is some way to combine the ideas of these algorithms.

Trifonov in [19] states that input partitioning produces addition chains of length $2q^2/\lg q$ ("$2k^2/\log_2 k$ summations"), and that a much slower—but apparently still usable—algorithm finds "considerably smaller" xor chains. Input-partitioning chains actually have length $(1 + o(1))q^2/\lg q$, so the comparison in [19] is clearly erroneous; it is nevertheless possible that the algorithm of [19] has some merit.

References

[1] — (no editor), *17th annual symposium on foundations of computer science*, IEEE Computer Society, 1976. MR 56:1766. See [15].

[2] — (no editor), *Proceedings of 1997 IEEE international symposium on information theory*, IEEE Computer Society, 1997. See [13].

[3] V. L. Arlazarov, E. A. Dinic, M. A. Kronrod, I. A. Faradžev, *On economical construction of the transitive closure of an oriented graph*, Soviet Mathematics Doklady **11** (1970), 1209–1210. ISSN 0197–6788. MR 42:4441. URL: http://cr.yp.to/bib/entries.html#1970/arlazarov. Citations in this document: §4, §4, §4.

[4] Daniel V. Bailey, Brian Baldwin, Lejla Batina, Daniel J. Bernstein, Peter Birkner, Joppe W. Bos, Gauthier van Damme, Giacomo de Meulenaer, Junfeng Fan, Tim Güneysu, Frank Gurkaynak, Thorsten Kleinjung, Tanja Lange, Nele Mentens, Christof Paar, Francesco Regazzoni, Peter Schwabe, Leif Uhsadel, *The Certicom challenges ECC2-X*, in Workshop Record of SHARCS'09 (2009). URL: http://cr.yp.to/papers.html#ecc2x. Citations in this document: §1.

[5] Gregory Bard, *Accelerating cryptanalysis with the Method of Four Russians* (2006). URL: http://eprint.iacr.org/2006/251. Citations in this document: §4.

[6] Daniel J. Bernstein, *Batch binary Edwards*, in [10] (2009), 317–336. URL: http://cr.yp.to/papers.html#bbe. Citations in this document: §1, §1.

[7] Tomas J. Boothby, Robert W. Bradshaw, *Bitslicing and the method of four Russians over larger finite fields*. URL: http://arxiv.org/abs/0901.1413. Citations in this document: §4.

[8] Peter de Rooij, *Efficient exponentiation using precomputation and vector addition chains*, in [9] (1995), 389–399. MR 1479665. Citations in this document: §2.

[9] Alfredo De Santis (editor), *Advances in cryptology: EUROCRYPT '94*, Lecture Notes in Computer Science, 950, Springer, 1995. ISBN 3–540–60176–7. MR 98h:94001. See [8].

[10] Shai Halevi (editor), *Advances in Cryptology—CRYPTO 2009, 29th annual international cryptology conference, Santa Barbara, CA, USA, August 16–20, 2009, proceedings*, Lecture Notes in Computer Science, 5677, Springer, 2009. See [6].

[11] O. B. Lupanov, *On rectifier and contact-rectifier circuits*, Doklady Akademii Nauk SSSR **111** (1956), 1171–1174. ISSN 0002–3264. URL: http://cr.yp.to/bib/entries.html#1956/lupanov. Citations in this document: §4, §4.

[12] Eugene W. Myers, *A four Russians algorithm for regular expression pattern matching*, Journal of the ACM **39** (1992), 430–448. Citations in this document: §4.

[13] Christof Paar, *Optimized arithmetic for Reed–Solomon encoders*, in [2] (1997), 250–250; see also newer version [14].

[14] Christof Paar, *Optimized arithmetic for Reed–Solomon encoders* (1997); see also older version [13]. URL: http://www.crypto.rub.de/imperia/md/content/texte/cnst.ps. Citations in this document: §5.

[15] Nicholas Pippenger, *On the evaluation of powers and related problems (preliminary version)*, in [1] (1976), 258–263; newer version split into [16] and [17]. MR 58:3682. URL: http://cr.yp.to/bib/entries.html#1976/pippenger. Citations in this document: §6.

[16] Nicholas Pippenger, *The minimum number of edges in graphs with prescribed paths*, Mathematical Systems Theory **12** (1979), 325–346; see also older version [15]. ISSN 0025–5661. MR 81e:05079. URL: http://cr.yp.to/bib/entries.html#1979/pippenger. Citations in this document: §6, §6.

[17] Nicholas Pippenger, *On the evaluation of powers and monomials*, SIAM Journal on Computing **9** (1980), 230–250; see also older version [15]. ISSN 0097–5397. MR 82c:10064. URL: http://cr.yp.to/bib/entries.html#1980/pippenger. Citations in this document: §6, §6.

[18] Nicola Santoro, *Extending the four Russians' bound to general matrix multiplication*, Information Processing Letters **10** (1980), 87–88. Citations in this document: §4.

[19] Peter Trifonov, *Matrix-vector multiplication via erasure decoding* (2007). URL: http://dcn.infos.ru/~petert/. Citations in this document: §6, §6, §6.