Reducing the Gate Count of Bitslice DES

Matthew Kwan

mkwan@darkside.com.au

Abstract. This paper describes various techniques to reduce the number of logic gates needed to implement the DES S-boxes in bitslice software. Using standard logic gates, an average of 56 gates per S-box was achieved, while an average of 51 was produced when non-standard gates were utilized. This is an improvement over the previous best result, which used an average of 61 non-standard gates.

1 Introduction

In his 1997 paper [1], Dr Eli Biham described an implementation of the Data Encryption Standard (DES) [7] that produced significant performance advantages on 64-bit RISC architectures. It essentially involved converting the DES algorithm into an equivalent logic circuit, using AND, OR, XOR, and NOT gates. When this circuit is run as software using the equivalent AND, OR, XOR, and NOT instructions on a 64-bit machine, it has the effect of executing DES 64 times in parallel. Such an implementation is known as *bitslice* DES.

Bitslice implementations of DES have a speed advantage because they do not require additional instructions for the various permutation operations that slow down traditional DES software. Instead, their bottleneck is the number of gates required to implement the logic circuit. Since there is no way to reduce the number of gates used to XOR the subkeys with the expanded input to each round (48 gates per round), or to XOR the output of each round with the data (32 gates per round), the S-boxes provide the only room for improvement.

DES has eight S-boxes, each of which can be thought of as a circuit with six bits of input, and four bits of output. Given an average of N logic gates to implement each S-box, the total number of gates in a DES implementation is thus 16 x $(48 + 32 + 8N)$ or $1280 + 128N$.

The algorithms described in this paper produce circuits with an average of 56 gates per S-box using standard instructions, and 51 using non-standard instructions. This compares with the previously best known average of 61 (using non-standard gates) [9], and the previously best published average of 88 (using standard gates) [4] [5] [6].

2 The Original Algorithm

In Biham's paper [1], he describes an algorithm that produces an average of approximately 100 gates per S-box. What follows is a brief recap.

Basically, for each S-box, the technique is to take two of the input bits, expand them to all 16 possible functions of two variables, and use the remaining four S-box inputs to select from those 16 functions. However, the details are slightly more complicated.

Think of an S-box output bit as being a function of six boolean variables. This can be implemented by using one of the variables to select between two functions of the remaining five variables. To generate a function of five variables, use one of those variables to select between two functions of four variables. And so on, down to functions of two variables. There are only 16 functions of two variables, and they can all be generated with total of 12 gates (assuming they are all needed).

To select between two functions F_a and F_b, if you use standard multiplexing, requires three gates (not counting the inversion of the selection bit, which only has to be done once per S-box)

$$(F_a\ AND\ sel)\ OR\ (F_b\ AND\ NOT\ sel) \tag{1}$$

However, using the XOR operator, you can carry out a modified form of selection with two gates

$$(F_c\ AND\ sel)\ XOR\ F_b \tag{2}$$

where $F_c = F_a$ XOR F_b

Without optimization, the number of gates required to implement all four output bits of an S-box is 132 ; 12 to generate all functions of two inputs, plus 4 x 2 x (8 + 4 + 2 + 1) gates for the selections. Using unspecified optimizations, Biham's original version of this algorithm achieved an average gate count of approximately 100.

3 An Incremental Improvement

The first improvement to the algorithm was developed independently by [4], [5], and [6]. It involved the ordering of the S-box input bits.

The original algorithm always used the fourth and fifth S-box input bits to expand to the 16 functions of two variables. The remaining inputs were always used for selection in the same order. However, there are 6! (720) orderings for these input bits, so it made sense to try all combinations to see which produced the least number of gates. An average of 88 gates resulted, as shown in table 1.

S-boxes	S1	S2	S3	S4	S5	S6	S7	S8
Gates	95	84	89	77	96	87	86	88

Table 1. Gate counts after first improvement

The following optimizations were used, which are probably close to Biham's unspecified optimizations, since they yield similar gate counts.

The first form of optimization occurs when a function of N variables is a constant 1 or a constant 0. For example, if F_c has a value of 1 regardless of its inputs, then the selection function (2) can be reduced to

$$sel\ XOR\ F_b \tag{3}$$

Similarly, other reductions are possible when $F_c = 0$, $F_b = 0$, or $F_b = 1$.

The second form of optimization involves the elimination of common subexpressions. In other words, before a function of N variables is built you first check if it has already been generated. If it exists, you re-use it, and no extra gates are required.

4 The RSA DES challenge

As a result of the 1997 RSA DES challenge, a number of improved bitslice DES implementations surfaced. Darrel Kindred [3] achieved an average of 66.1 gates per S-box, while Rocke Verser [9] achieved 61. Verser's code was used by the DESCHALL team, which eventually won.

Although both authors have kept their designs confidential, they have revealed that they made use of non-standard gates. These are gates which, although not directly supported by the C programming language, are often supported by the underlying CPU. The instructions are NAND, NOR, NXOR, AND-NOT, and OR-NOT. The SPARC and Alpha architectures do not provide NAND and NOR, but it is a fairly simple matter to remove them from a circuit by turning them into AND and OR respectively, and pushing the inversions downstream.

5 A Major Redesign

The benchmark had dropped by over 30 percent, so a major redesign was called for if the author was going to reclaim the lead. The first step was to simplify the representation of the problem.

Each output bit of an S-box is a function of the six input bits. A convenient representation for this is a six-variable Karnaugh map, which can be held in a 64-bit integer (or a pair of 32-bit integers).

It turns out that the output of every gate in the circuit can also be expressed as a six-variable Karnaugh map, so the problem was formulated as follows: Given a target Karnaugh map (i.e. an output bit), try to construct it with combinations of the existing maps. To begin with, the only maps are the six inputs, but as the circuit builds up, more become available.

Because the circuit will be built using previously generated gates where possible, the order that the circuit is built is very important. As a result, all 6! orderings of input bits, and all 4! orderings of output bits have to be tried. This means that any design algorithm will be run with 17280 orderings, and the best result will be selected.

In general, the design philosophy is one of brute force. Try every different combination of design techniques, and use the result with the fewest gates.

6 Recursive Search

At the heart of the design is a recursive search function. It takes as its inputs the following.

1. The circuit so far. Initially this is just the six input bits.
2. The 64-bit Karnaugh map that it has to generate.
3. A 64-bit mask. All zero bits correspond to a *don't care* value in the Karnaugh map.
4. Which input bit to use for selecting between two functions.

The function tries to generate the desired Karnaugh map with the following techniques (in the order given). If, at any time, the size of the circuit gets bigger than the best found so far, the function returns a *not found* value.

1. Look through the existing circuit. If there is a gate that produces the desired map, simply return the ID of that gate.
2. If there are any gates whose *inverse* produces the desired map, append a NOT gate, and return the ID of the NOT gate.
3. Look at all pairs of gates in the existing circuit. If they can be combined with a single gate to produce the desired map, add that single gate and return its ID.
4. Look at all combinations of two or three gates in the circuit. If they can be combined with two gates to produce the desired map, add the gates, and return the ID of the one that produces the desired map.
5. Use the specified input bit to select between two Karnaugh maps. Call this function recursively to generate those two maps.

Techniques 3, 4, and 5 are fairly complex, and are described in more detail below.

7 Combining Existing Gates

The search routine can run in two different modes - using standard gates (AND, OR, XOR, and NOT), or using non-standard gates. The choice of the mode determines which options are tried.

Using standard gates, each pair of gates can be combined with any of three different operators (AND, OR, and XOR). With non-standard gates there are ten choices. All of these options are first tried to test if they generate the desired map.

If standard gates are being used, the next option is to emulate the other seven non-standard pairwise operations by using two standard gates. As before, they are used to combine all pairs of existing gates in an attempt to produce the desired map.

The next option is to combine all possible triplets of existing gates using all possible functions of two gates. With standard gates there are 21 unique functions of three variables that can be built with two gates. With non-standard gates there are 114. As might be expected, testing all possible sets of three existing gates with all of those functions is very time-consuming, especially with non-standard gates.

8 Selection Functions

To implement the selection function described in (2) we need to generate the functions F_b and F_c. Given that the desired Karnaugh map is denoted by F_{out}, and the map defined by the selecting input bit is denoted by F_{sel}, then

$$F_b = F_{out}\ AND\ NOT\ F_{sel} \tag{4}$$

$$F_c = F_b\ XOR\ F_{out} \tag{5}$$

When generating F_b, the recursive search function is called with the *don't care* mask ANDed with $NOT\ F_{sel}$, and using the next input bit for selection. When generating F_c, the *don't care* mask is ANDed with F_{sel}.

It is also possible to build selection functions using gates other than AND. For example, OR can be used.

$$F_{out} = (F_d\ OR\ F_{sel})\ XOR\ F_e \tag{6}$$

where

$$F_d = NOT\ F_{out}\ AND\ F_{sel} \tag{7}$$

$$F_e = F_d\ XOR\ F_{out} \tag{8}$$

Similarly, when using non-standard gates, the AND gate can be replaced with ANDNOT, ORNOT, NAND, and NOR.

In practice, the procedure was to try all the different selection functions to generate the desired map, and to choose the one that resulted in the fewest gates.

9 Results

Applying all the above techniques resulted in a drastic reduction in the number of gates. As can be seen in table 2, when using standard logic gates, the average gate count was reduced to 56.

Generating the S-boxes with non-standard gates was a very time-consuming task. It took approximately six weeks, utilizing an average of three 64-bit workstations simultaneously during nights and weekends. Eventually the average gate count was reduced to 51, as shown in table 3.

S-boxes	S1	S2	S3	S4	S5	S6	S7	S8
Gates	63	56	57	42	62	57	57	54

Table 2. Using standard gates

S-boxes	S1	S2	S3	S4	S5	S6	S7	S8
Gates	56	50	53	39	56	53	51	50

Table 3. Using non-standard gates

The actual gates generated by the previously-described techniques can be downloaded from `http://www.darkside.com.au/bitslice`, where they are available as C source code. The gates also appear in the appendices as circuit diagrams.

10 Summary

In situations permitting parallel execution of DES, bitslice implementations using these S-boxes are currently the fastest on all major architectures. As a result, the S-boxes (or slightly modified hand-crafted assembler code equivalents) are used in most Unix password crackers (e.g. *John the Ripper* [8]) and exhaustive DES key search applications (e.g. the `distributed.net` DES client [2]).

It is likely that the number of gates can be reduced still further, since the techniques described here are by no means optimal. They simply improve on existing methods, and it is quite possible that a completely different approach will yield better results.

In addition, they make no attempt to minimize the number of registers required to execute the instructions. The algorithms simply produce C code, and leave the problem of register allocation to the compiler. An algorithm which generates code tailored for a particular number of registers may well run a lot faster, especially on architectures with MMX instructions, which have eight 64-bit registers.

11 Acknowledgements

The author wishes to thank The Preston Group Pty Ltd for allowing him to run the search software on their machines week after week after week.

References

1. E. Biham, A Fast New DES Implementation in Software, *proceedings of Fast Software Encryption - Fourth International Workshop, Haifa, Israel*, Springer-Verlag, pp. 260-272, 1997
2. distributed.net, http://www.distributed.net
3. D. Kindred, http://www.cs.cmu.edu/People/dkindred/des/bitslice.html
4. M. Kwan, http://www.darkside.com.au/bitslice/old-alg.html
5. H. Laegreid, Differential Cryptanalysis on the Cray Origin 2000 *Cand. Scient. thesis*, Dept. of Informatics, University of Bergen, September 1997
6. S. Moriai, S. Amada and T. Shimoyama, Optimized Fast Software Implementation of Block Ciphers *IEICE Trans. Fundamentals*, Vol. E81-A, No. 1, January 1998
7. National Bureau of Standards, *Data Encryption Standard*, FIPS PUB 46, U.S. Department of Commerce, 1977
8. Solar Designer, http://www.false.com/security/john
9. R. Verser, Personal communication

A S-box circuit diagrams

Figures 1 and 2 show the circuit diagrams for the eight DES S-boxes, using standard and non-standard gates respectively.

This article was processed using the LaTeX macro package with LLNCS style

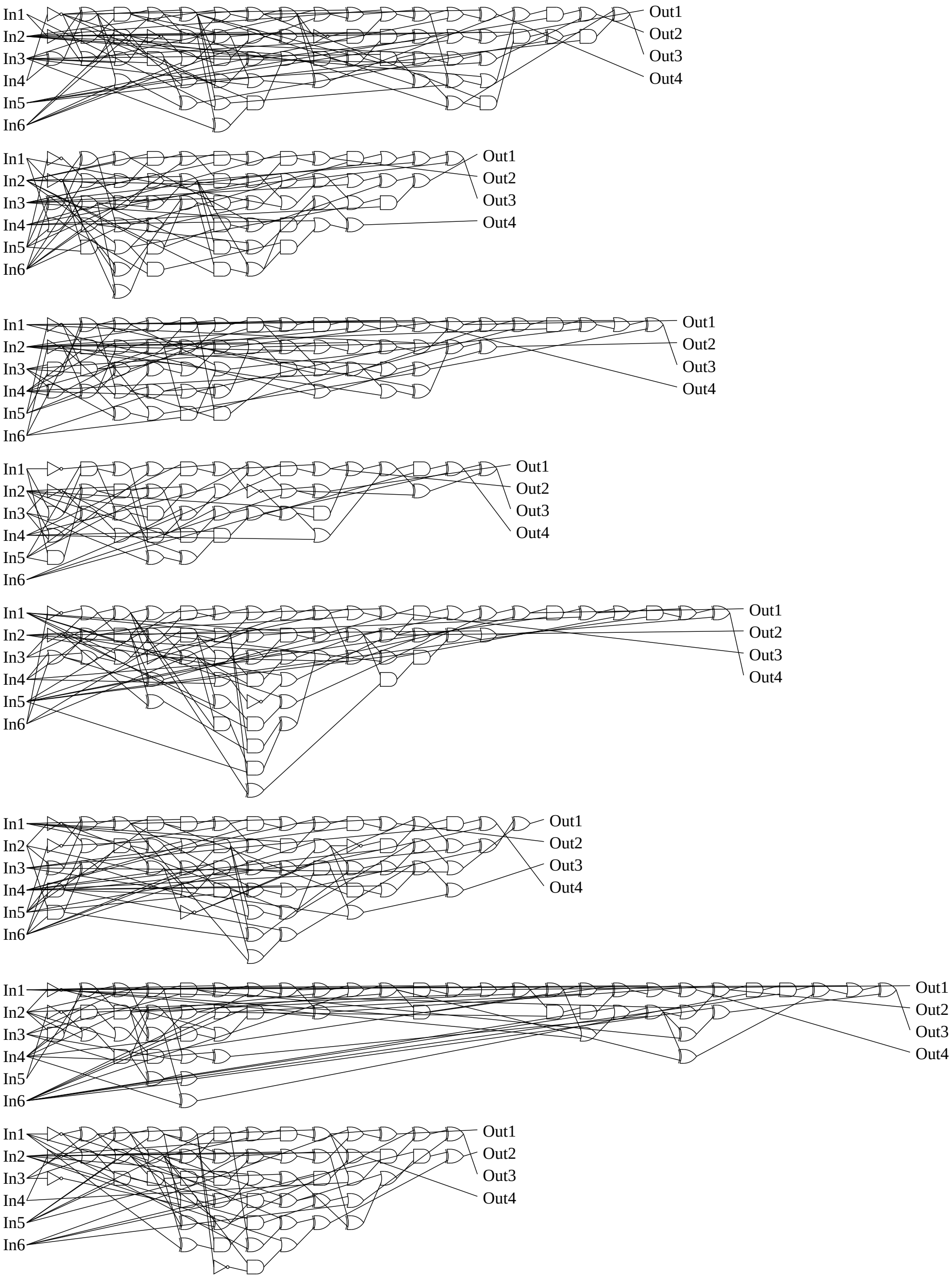

Fig. 1. DES S-boxes using standard gates

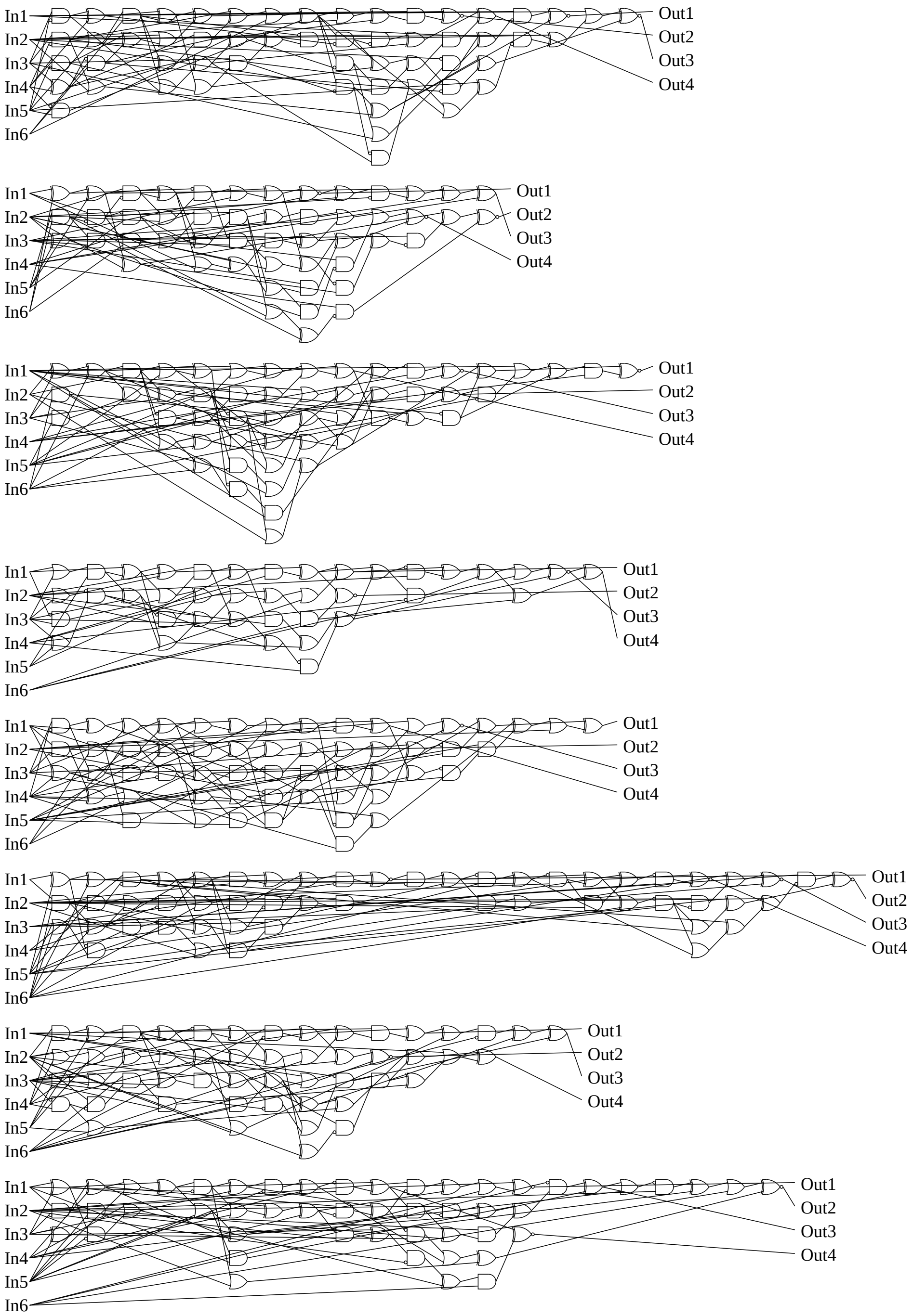

Fig. 2. DES S-boxes using non-standard gates