

# Speeding up Serpent

Dag Arne Osvik \*

March 15, 2000

## Abstract

We present a method for finding efficient instruction sequences for the Serpent S-boxes. Current implementations need many registers to store temporary variables, yet the common x86 processors only have 8 registers, of which even fewer are available for computations. The instructions are also destructive, replacing one input with the output. Alternative versions of the S-box instructions are presented, requiring only 5 registers and also utilizing parallelism. Speedup of C language implementations of 24% is shown on the Pentium Pro Processor, and 42% on the Pentium, both compared to the previously fastest known implementation of Serpent.

## 1 Introduction

The main aspect of the finalists for the Advanced Encryption Standard is the security level they provide, especially against already known attack methods. Another aspect is the encryption speed they allow in different applications. The goal of this work has been to find ways to improve the execution speed of the Serpent algorithm on the x86 processors, including use of two-way parallel execution.

Serpent[1], being an SP-network (it consists of substitutions and permutations), has two major parts; the S-boxes and the linear transformation. The latter has a simple structure, and is well suited for manual optimization. The S-boxes are 16-element permutations, and are performed in a bit parallel (also known as bitslice) style by simple boolean operations.

## 2 The problem

The x86 processors, which can be found in nearly every personal computer, have some clearly distinguishing features when compared to more modern architectures. One of these is the small number of registers, only 8. Another is the instruction set, where almost all instructions always modify one of their input registers.

---

\*University of Bergen, Department of Informatics, N-5020 Bergen, Norway. Email address: osvik@ii.uib.no

### 3 Previous work

Other efforts on optimizing Serpent have centered on the more purely mathematical problem of lowering the number of boolean operations needed to express the S-boxes [2]. Thus those essential properties of the x86 processors have been ignored. The result is a high so-called 'register pressure', meaning compilers have to put temporary variables in memory, issuing load and store instructions in addition to the actual computation. The compiler also gets the job of copying values when needed. One note is appropriate here, though; lowering the number of operations is a much better approach for RISC processors than it is for x86, as RISC instructions don't have to destroy an input value, and those processors typically have 32 registers, making register pressure a non-issue. A comparison of my results to those others (on x86) is given in a later section.

### 4 Our approach

One possible approach to solving a computational problem is to consider all possible computations, ordered by their length. Searching to the depth needed to find complete solutions in the case of Serpent S-boxes is infeasible using this simple approach, so we need substantial improvements.

#### 4.1 Serpent S-boxes

The Serpent S-boxes are 16-element permutations, implying that they belong to a somewhat special subset of functions in  $\{Z_{16} \rightarrow Z_{16}\}$ . Now, every number from 0 to 15 can be represented by a 4-digit binary number, so these functions map 4 input bits to 4 output bits. They can also be split into 4 functions mapping 4 input bits to 1 output bit, just like any 4-bit number may be split into 4 separate bits. Now recall that any function can be uniquely specified by telling its output value for every allowed input value. In the case of 4-to-1 bit functions this is simply a list of 16 binary digits, given some ordering of the input values.

#### 4.2 Finding solutions

We need some way to transform any 4 input bits into the corresponding 4 output bits using only those instructions available in the x86 instruction set, and in a bit parallel way. We'll use  $S_2$  as an example:

$x$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$S_2(x)$	8	6	7	9	3	12	10	15	13	1	14	4	0	11	5	2

Now rewrite  $x$  and  $S_2(x)$  in binary:

$x_3$	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
$x_2$	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
$x_1$	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
$x_0$	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
$S_{2,3}$	1	0	0	1	0	1	1	1	1	0	1	0	0	1	0	0
$S_{2,2}$	0	1	1	0	0	1	0	1	1	0	1	1	0	0	1	0
$S_{2,1}$	0	1	1	0	1	0	1	1	0	0	1	0	0	1	0	1
$S_{2,0}$	0	0	1	1	1	0	0	1	1	1	0	0	0	1	1	0

Each column in this table contains the bits of some value for  $x$ , as well as the bits of the corresponding  $S_2(x)$ . The set of all columns contains all possible values for  $x$ . The number of columns is thus determined by the number of possible inputs, and is *not* related to the word length of any processor.

If we find a way of combining the  $x_i$  rows by boolean operations so that we get the  $S_{2,i}$  rows, then applying those operations to the bits of an input value  $x$  is equivalent to looking up  $S_2(x)$ . To see how this is actually done, we will look at the execution of an instruction sequence for  $S_2$ .

The x86 instructions usable for the S-boxes are these:

Instruction	Effect	C expression
<i>and a, b</i>	$a := a \cdot b$	$a \&= b$
<i>or a, b</i>	$a := a + b$	$a  = b$
<i>xor a, b</i>	$a := a \oplus b$	$a ^= b$
<i>not a</i>	$a := a \oplus 1$	$a = \sim a$
<i>mov a, b</i>	$a := b$	$a = b$

Suppose we have 5 registers, named  $r_0, \dots, r_4$ , available for our computations, and 4 of them initially contain our 4 input bits ( $r_i$  contains  $x_i$ ,  $0 \leq i \leq 3$ ). As  $r_4$  is not an input register, we just ignore its previous contents. Thus we have this initial state:

$r_4$																
$r_3$	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
$r_2$	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
$r_1$	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
$r_0$	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

The instruction sequence found by the search program (with two-way parallelism shown) is this:

mov r4, r0	and r0, r2
xor r0, r3	xor r2, r1
xor r2, r0	or r3, r4
xor r3, r1	xor r4, r2
mov r1, r3	or r3, r4
xor r3, r0	and r0, r1
xor r4, r0	xor r1, r3
xor r1, r4	not r4

Executing the first line of instructions makes the modifications  $r_4 := r_0$ ;  $r_0 := r_0 \cdot r_2$ , giving us this new state:

$r_4$	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
$r_3$	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
$r_2$	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
$r_1$	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
$r_0$	0	0	0	0	0	1	0	1	0	0	0	0	0	1	0	1

Next, we perform  $r_0 := r_0 \oplus r_3$ ;  $r_2 := r_2 \oplus r_1$ .

$r_4$	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
$r_3$	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
$r_2$	0	0	1	1	1	1	0	0	0	0	1	1	1	1	0	0
$r_1$	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
$r_0$	0	0	0	0	0	1	0	1	1	1	1	1	1	0	1	0

Now things get more interesting. Notice the values in the  $r_2$  row after  $r_2 := r_2 \oplus r_0$ ;  $r_3 := r_3 + r_4$ .

$r_4$	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
$r_3$	0	1	0	1	0	1	0	1	1	1	1	1	1	1	1	1
$r_2$	0	0	1	1	1	0	0	1	1	1	0	0	0	1	1	0
$r_1$	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
$r_0$	0	0	0	0	0	1	0	1	1	1	1	1	1	0	1	0

$r_2$  is now the same as  $S_{2,0}$ , one of our wanted output bits.

Executing the next three lines of instruction pairs, we reach this state:

$r_4$	0	1	1	0	1	1	0	0	1	0	0	1	0	0	1	1
$r_3$	0	1	1	0	1	0	1	1	0	0	1	0	0	1	0	1
$r_2$	0	0	1	1	1	0	0	1	1	1	0	0	0	1	1	0
$r_1$	0	1	1	0	0	1	1	0	1	1	0	0	1	1	0	0
$r_0$	0	0	0	0	0	1	0	0	1	1	0	0	1	0	0	0

Now  $r_3$  is the same as  $S_{2,1}$ . The next two lines complete the work:

$r_4 = S_{2,3}$	1	0	0	1	0	1	1	1	1	0	1	0	0	1	0	0
$r_3 = S_{2,1}$	0	1	1	0	1	0	1	1	0	0	1	0	0	1	0	1
$r_2 = S_{2,0}$	0	0	1	1	1	0	0	1	1	1	0	0	0	1	1	0
$r_1 = S_{2,2}$	0	1	1	0	0	1	0	1	1	0	1	1	0	0	1	0
$r_0$	0	0	0	0	0	1	0	0	1	1	0	0	1	0	0	0

Thus we have a way of applying the function  $S_2$ , using only boolean operations with 1-bit input values. Now remember that the columns were initially just a list of possible input values. So the operations performed are actually independent of the number and contents of the columns. So we may now e.g. extend our table to 32 columns and allow any contents in each of the columns. Then, when the operations are performed, they perform  $S_2$  32 times in parallel. This is exactly what we do on a processor with 32-bit registers.

The search for such solutions basically tries all possible instruction sequences of a given length, looking for rows equal to those of the S-box wanted. Shorter sequences are generally preferred, so we start with a small length, progressing to longer ones when no solution is found. To search for sequences capable of parallel execution, like the one above, we require that an instruction not read the output of an earlier instruction on the same line. It may write to an input register of an earlier instruction, though, as that will in no way affect the outcome of the other instruction.

### 4.3 Optimizations

Below are short descriptions of the most important optimizations of the search algorithm. Almost all of these avoid removing solutions without keeping an equivalent solution.

- Recursion stops when the register contents can no longer generate a permutation.
- When two instruction sequences are identified as being equivalent, we remove one of them from the search.
- No instruction other than *mov* may make a register contain a copy of the value in another register.
- Unread registers may not be written to by the *mov* instruction.
- Negated registers (those last modified by a *not* instruction) are marked as such, and may not again be negated until they have been read.
- Lookahead functions efficiently calculate a set containing all values reachable in one or two cycles.
- The search is narrowed by requiring an increasing number of result values in the registers as the search goes deeper. This constraint is important for deep searches, but its most strict variant (increasing the required number

as soon as there is at least one sequence that reached it) often drops better solutions, and should be relaxed by postponing the requirement by one or two cycles.

- The instructions are limited to using only 5 registers.

First experiences using the search program with 7 registers available showed most solutions using 6 of those, while others only used 5 registers. Further testing always provided solutions using 5 registers whenever a 6 register solution was found. Given the reduced complexity of the search, and the advantages of having the S-boxes do all their computations in only 5 registers, I chose to limit the search accordingly.

## 5 Results

The S-box functions chosen from the search results have the properties shown in the table. Cycle count is for running these on processors like the Pentium, with two integer execution units running in parallel.

Function	Instructions	Cycles	Registers
$S_0$	18	9	5
$S_1$	18	10	5
$S_2$	16	8	5
$S_3$	19	10	5
$S_4$	20	10	5
$S_5$	19	10	5
$S_6$	18	10	5
$S_7$	20	11	5
$S_0^{-1}$	19	11	5
$S_1^{-1}$	19	11	5
$S_2^{-1}$	19	10	5
$S_3^{-1}$	18	10	5
$S_4^{-1}$	20	11	5
$S_5^{-1}$	19	10	5
$S_6^{-1}$	17	9	5
$S_7^{-1}$	19	10	5

The low register pressure of these functions makes their compiled code completely free from loads and stores. So we only load input data and round keys, and store the result. Except for the round key loads, no memory operations are issued during encryption. This is completely different from the S-boxes used in the AES submission package[1], as well as those found by Gladman and Simpson [2], which depend heavily on memory for storage during encryption. Also, the memory footprint of the encryption routines themselves is much reduced; a fully inlined encryption requires less than 4 kilobytes.

## 6 Optimized implementations

Due to the problem of making C compilers schedule instructions properly for the Pentium, the S-box instructions were also incorporated into assembly routines for Serpent encryption and decryption. The result was then manually tuned for this processor (which may make it slower on other processors). The implementation was made with these constraints:

- The stack pointer register is reserved for its normal use.
- Make the routines suitable as plug-in replacements for the C routines in the AES submittal of Serpent, allowing easy testing.
- One register contains a pointer to the round key table.

Keeping the stack and key table pointers, instead of using them as general purpose registers, allows multiple simultaneous use of the routines, such as in multithreaded environments.

A new set of four round keys is loaded 33 times during an encryption or decryption. Reserving a register to point to the key table avoids having to reload the pointer every time. The ideal solution for performance is to put the round keys on the stack or in a fixed location, as that would free up the key pointer (round keys would be fetched using the stack pointer). But, since the pointer to the round key table is a parameter to the routines we replace, it is needed.

Given these limitations, we still have those five registers needed for the S-boxes, plus one free for whatever use we might have for it, like early loading of a round key. This gives the opportunity to exploit the parallelism of the Pentium nearly to its full extent, thus usually executing two instructions per cycle (some instructions can only execute one at a time). The benefit of one more free register, as could be gained by fixing the location of round keys, will thus be minimal.

## 7 Performance comparison

Speed testing was done on these computers:

Processor	Clock speed	RAM size	OS
486 SX	33 MHz	20 MB	Linux 2.0
Pentium	100 MHz	64 MB	Linux 2.0
(Dual) Celeron	333 MHz	256 MB	Linux 2.2

The following tables give a comparison of the different implementations of Serpent on these computers. My speed figures in Mbit/s are scaled to given clock speeds, assuming all memory operations are performed in level 1 caches. In the case of Pentium Pro, I compare against the best of Gladman's most

recent numbers. On the others, numbers are compared to those reported by Granboulan [3], using Gladman’s code.

- 486 DX/2-50

Implementation	Encryption		Decryption	
	Mbit/s	cycles	Mbit/s	cycles
Gladman’s code	0.48	12900		
Osvik	3.8	1650	3.8	1660

- Pentium 90

Implementation	Encryption		Decryption	
	Mbit/s	cycles	Mbit/s	cycles
AES submission	7.17	1605	5.88	1956
Gladman’s code	8.56	1290		
Osvik	12.7	907	12.7	905
Osvik, asm	14.4	800		

- Pentium Pro 200

Implementation	Encryption		Decryption		Key setup cycles
	Mbit/s	cycles	Mbit/s	cycles	
AES submission	21.8	1170	20.6	1301	
Gladman	27.0	945	26.9	951	1290
Osvik	33.7	759	33.2	770	1106

The compiler used to compile both my own and the AES submission C code is PentiumGCC version 2.95.2. For my own code, I used the options “-O -mpentium -fPIC -fomit-frame-pointer” on Pentium and “-O2 -mpentium -fPIC -fomit-frame-pointer” on PPro. For the AES submission code I used “-O -mpentium”. Other optimization settings I tried reduced the speed achieved. All times are measured including parameter passing, function call and return from the function. Timings on the 486 are not nearly as accurate as the others, as it does not have a cycle counter.

Note: the figures quoted above are for Gladman’s results in C using a static array of round keys which frees up an extra register. This only allows multiple concurrent encryptions when they all use the same key. His C++ code, which does not have this limitation, shows a 3% performance reduction.

## 8 Future directions

- My implementations may be further tuned - actually, I expected the Pentium assembly implementation to come close to 735 cycles for encryption.

While trying to manually optimize the encryption, I found the Pentium to be very touchy regarding tight dependencies involving rotation instructions. Given the Pentium processor's slowdown when executing such instruction sequences, 735 cycles seems to be unreachable. Still, faster S-boxes might exist, as my search has not been exhaustive.

- The key setup function can generate the encryption code with round keys embedded directly in the instructions, thus removing the load instructions and saving upto 66 cycles on the Pentium. This will increase key setup time, though.
- 3-way parallelism on x86 (AMD Athlon). This only requires a (theoretically) simple extension of my current search program. The curious can quite easily verify that  $S_6^{-1}$  and  $S_7^{-1}$  both can execute in 7 cycles with up to 3 instructions/cycle, as opposed to 9 and 10 cycles on Pentium/.../Pentium III.
- Hardware implementations have a natural emphasis on parallelism. Preliminary results in this area look extremely promising; given 3-input nand and nor gates, and (at most) 2-input versions of other gates, all S-boxes can be performed with a gate depth of only 3. Combined with a depth of 4 for the linear transformation and 1 for key mixing, this indicates that several Gb/s should be possible in CBC mode with common technology. If we can also add 3-input (n)xor, the gate depth of one round is reduced to no more than 5.
- The instruction sets of RISC processors may be viewed as a set of gates from which we can build wide S-box functions. Their lack of 3-input logical operations raises the maximum gate depth needed to 4. That is, given enough parallelism on a RISC (or EPIC) chip, all S-boxes have solutions requiring no more than 4 cycles to execute. This hardware-style RISC optimization will be further investigated in the near future.

## 9 Acknowledgements

I would like to thank my supervisor, Lars R. Knudsen, for proposing this project, and for his advice and criticisms regarding this article.

My parents and some of my friends and fellow students have been helpful in various ways. Most of the search and timing tests were performed on Odd Egil Nerland's computers. Gisle Sælensminde has made an Ada implementation using my S-box functions, and in the process he made a Python script automating inlining of the encryption functions. This was necessary to avoid stressing the GCC register allocator with these rather intricate S-boxes. My C and assembly implementations were also made using slightly modified versions of his script, saving much work and time.

## References

- [1] RJ Anderson, E Biham, LR Knudsen, “Serpent: A Proposal for the Advanced Encryption Standard”
- [2] BR Gladman:  
*[http://www.btinternet.com/~brian.gladman/cryptography\\_technology/](http://www.btinternet.com/~brian.gladman/cryptography_technology/)*
- [3] L Granboulan:  
*<http://www.dmi.ens.fr/~granboul/recherche/AES/timings.html>*
- [4] Intel Corporation, “Intel Architecture Optimization Manual”, Order Number 242816-003, 1997.

## Appendix

Below are all the S-box functions selected from the search results. The functions expect their input values to be in r0 .. r3, ordered from least to most significant bit. The contents of r4 are ignored. Output values are given in the registers listed at the bottom of each table, again ordered from least to most significant bit.

$S_0$		$S_0^{-1}$	
r3 ^= r0	r4 = r1	r2 =~ r2	r4 = r1
r1 &= r3	r4 ^= r2	r1  = r0	r4 =~ r4
r1 ^= r0	r0  = r3	r1 ^= r2	r2  = r4
r0 ^= r4	r4 ^= r3	r1 ^= r3	r0 ^= r4
r3 ^= r2	r2  = r1	r2 ^= r0	r0 &= r3
r2 ^= r4	r4 =~ r4	r4 ^= r0	r0  = r1
r4  = r1	r1 ^= r3	r0 ^= r2	r3 ^= r4
r1 ^= r4	r3  = r0	r2 ^= r1	r3 ^= r0
r1 ^= r3	r4 ^= r3	r3 ^= r1	
		r2 &= r3	
		r4 ^= r2	
r1, r4, r2, r0		r0, r4, r1, r3	

$S_1$		$S_1^{-1}$	
r0 $\overset{\sim}{=}$ r0	r2 $\overset{\sim}{=}$ r2	r4 $=$ r1	r1 $\overset{\wedge}{=}$ r3
r4 $=$ r0	r0 $\&=$ r1	r3 $\&=$ r1	r4 $\overset{\wedge}{=}$ r2
r2 $\overset{\wedge}{=}$ r0	r0 $ =$ r3	r3 $\overset{\wedge}{=}$ r0	r0 $ =$ r1
r3 $\overset{\wedge}{=}$ r2	r1 $\overset{\wedge}{=}$ r0	r2 $\overset{\wedge}{=}$ r3	r0 $\overset{\wedge}{=}$ r4
r0 $\overset{\wedge}{=}$ r4	r4 $ =$ r1	r0 $ =$ r2	r1 $\overset{\wedge}{=}$ r3
r1 $\overset{\wedge}{=}$ r3	r2 $ =$ r0	r0 $\overset{\wedge}{=}$ r1	r1 $ =$ r3
r2 $\&=$ r4	r0 $\overset{\wedge}{=}$ r1	r1 $\overset{\wedge}{=}$ r0	r4 $\overset{\sim}{=}$ r4
r1 $\&=$ r2		r4 $\overset{\wedge}{=}$ r1	r1 $ =$ r0
r1 $\overset{\wedge}{=}$ r0	r0 $\&=$ r2	r1 $\overset{\wedge}{=}$ r0	
r0 $\overset{\wedge}{=}$ r4		r1 $ =$ r4	
		r3 $\overset{\wedge}{=}$ r1	
r2, r0, r3, r1		r4, r0, r3, r2	

$S_2$		$S_2^{-1}$	
r4 $=$ r0	r0 $\&=$ r2	r2 $\overset{\wedge}{=}$ r3	r3 $\overset{\wedge}{=}$ r0
r0 $\overset{\wedge}{=}$ r3	r2 $\overset{\wedge}{=}$ r1	r4 $=$ r3	r3 $\&=$ r2
r2 $\overset{\wedge}{=}$ r0	r3 $ =$ r4	r3 $\overset{\wedge}{=}$ r1	r1 $ =$ r2
r3 $\overset{\wedge}{=}$ r1	r4 $\overset{\wedge}{=}$ r2	r1 $\overset{\wedge}{=}$ r4	r4 $\&=$ r3
r1 $=$ r3	r3 $ =$ r4	r2 $\overset{\wedge}{=}$ r3	r4 $\&=$ r0
r3 $\overset{\wedge}{=}$ r0	r0 $\&=$ r1	r4 $\overset{\wedge}{=}$ r2	r2 $\&=$ r1
r4 $\overset{\wedge}{=}$ r0	r1 $\overset{\wedge}{=}$ r3	r2 $ =$ r0	r3 $\overset{\sim}{=}$ r3
r1 $\overset{\wedge}{=}$ r4	r4 $\overset{\sim}{=}$ r4	r2 $\overset{\wedge}{=}$ r3	r0 $\overset{\wedge}{=}$ r3
		r0 $\&=$ r1	r3 $\overset{\wedge}{=}$ r4
		r3 $\overset{\wedge}{=}$ r0	
r2, r3, r1, r4		r1, r4, r2, r3	

$S_3$		$S_3^{-1}$	
r4 $=$ r0	r0 $ =$ r3	r4 $=$ r2	r2 $\overset{\wedge}{=}$ r1
r3 $\overset{\wedge}{=}$ r1	r1 $\&=$ r4	r0 $\overset{\wedge}{=}$ r2	r4 $\&=$ r2
r4 $\overset{\wedge}{=}$ r2	r2 $\overset{\wedge}{=}$ r3	r4 $\overset{\wedge}{=}$ r0	r0 $\&=$ r1
r3 $\&=$ r0	r4 $ =$ r1	r1 $\overset{\wedge}{=}$ r3	r3 $ =$ r4
r3 $\overset{\wedge}{=}$ r4	r0 $\overset{\wedge}{=}$ r1	r2 $\overset{\wedge}{=}$ r3	r0 $\overset{\wedge}{=}$ r3
r4 $\&=$ r0	r1 $\overset{\wedge}{=}$ r3	r1 $\overset{\wedge}{=}$ r4	r3 $\&=$ r2
r4 $\overset{\wedge}{=}$ r2	r1 $ =$ r0	r3 $\overset{\wedge}{=}$ r1	r1 $\overset{\wedge}{=}$ r0
r1 $\overset{\wedge}{=}$ r2	r0 $\overset{\wedge}{=}$ r3	r1 $ =$ r2	r0 $\overset{\wedge}{=}$ r3
r2 $=$ r1	r1 $ =$ r3	r1 $\overset{\wedge}{=}$ r4	
r1 $\overset{\wedge}{=}$ r0		r0 $\overset{\wedge}{=}$ r1	
r1, r2, r3, r4		r2, r1, r3, r0	

$S_4$		$S_4^{-1}$	
r1 $\hat{=}$ r3	r3 $\hat{=}^{\sim}$ r3	r4 $=$ r2	r2 $\&=$ r3
r2 $\hat{=}$ r3	r3 $\hat{=}$ r0	r2 $\hat{=}$ r1	r1 $ =$ r3
r4 $=$ r1	r1 $\&=$ r3	r1 $\&=$ r0	r4 $\hat{=}$ r2
r1 $\hat{=}$ r2	r4 $\hat{=}$ r3	r4 $\hat{=}$ r1	r1 $\&=$ r2
r0 $\hat{=}$ r4	r2 $\&=$ r4	r0 $\hat{=}^{\sim}$ r0	r3 $\hat{=}$ r4
r2 $\hat{=}$ r0	r0 $\&=$ r1	r1 $\hat{=}$ r3	r3 $\&=$ r0
r3 $\hat{=}$ r0	r4 $ =$ r1	r3 $\hat{=}$ r2	r0 $\hat{=}$ r1
r4 $\hat{=}$ r0	r0 $ =$ r3	r2 $\&=$ r0	r3 $\hat{=}$ r0
r0 $\hat{=}$ r2	r2 $\&=$ r3	r2 $\hat{=}$ r4	
r0 $\hat{=}^{\sim}$ r0	r4 $\hat{=}$ r2	r2 $ =$ r3	r3 $\hat{=}$ r0
		r2 $\hat{=}$ r1	
r1, r4, r0, r3		r0, r3, r2, r4	

$S_5$		$S_5^{-1}$	
r0 $\hat{=}$ r1	r1 $\hat{=}$ r3	r1 $\hat{=}^{\sim}$ r1	r4 $=$ r3
r3 $\hat{=}^{\sim}$ r3	r4 $=$ r1	r2 $\hat{=}$ r1	r3 $ =$ r0
r1 $\&=$ r0	r2 $\hat{=}$ r3	r3 $\hat{=}$ r2	r2 $ =$ r1
r1 $\hat{=}$ r2	r2 $ =$ r4	r2 $\&=$ r0	r4 $\hat{=}$ r3
r4 $\hat{=}$ r3	r3 $\&=$ r1	r2 $\hat{=}$ r4	r4 $ =$ r0
r3 $\hat{=}$ r0	r4 $\hat{=}$ r1	r4 $\hat{=}$ r1	r1 $\&=$ r2
r4 $\hat{=}$ r2	r2 $\hat{=}$ r0	r1 $\hat{=}$ r3	r4 $\hat{=}$ r2
r0 $\&=$ r3	r2 $\hat{=}^{\sim}$ r2	r3 $\&=$ r4	r4 $\hat{=}$ r1
r0 $\hat{=}$ r4	r4 $ =$ r3	r3 $\hat{=}$ r4	r4 $\hat{=}^{\sim}$ r4
r2 $\hat{=}$ r4		r3 $\hat{=}$ r0	
r1, r3, r0, r2		r1, r4, r3, r2	

$S_6$		$S_6^{-1}$	
r2 $\hat{=}^{\sim}$ r2	r4 $=$ r3	r0 $\hat{=}$ r2	r4 $=$ r2
r3 $\&=$ r0	r0 $\hat{=}$ r4	r2 $\&=$ r0	r4 $\hat{=}$ r3
r3 $\hat{=}$ r2	r2 $ =$ r4	r2 $\hat{=}^{\sim}$ r2	r3 $\hat{=}$ r1
r1 $\hat{=}$ r3	r2 $\hat{=}$ r0	r2 $\hat{=}$ r3	r4 $ =$ r0
r0 $ =$ r1	r2 $\hat{=}$ r1	r0 $\hat{=}$ r2	r3 $\hat{=}$ r4
r4 $\hat{=}$ r0	r0 $ =$ r3	r4 $\hat{=}$ r1	r1 $\&=$ r3
r0 $\hat{=}$ r2	r4 $\hat{=}$ r3	r1 $\hat{=}$ r0	r0 $\hat{=}$ r3
r4 $\hat{=}$ r0	r3 $\hat{=}^{\sim}$ r3	r0 $ =$ r2	r3 $\hat{=}$ r1
r2 $\&=$ r4		r4 $\hat{=}$ r0	
r2 $\hat{=}$ r3			
r0, r1, r4, r2		r1, r2, r4, r3	

$S_7$			$S_7^{-1}$		
r4 = r1	r1   = r2		r4 = r2	r2 ^ = r0	
r1 ^ = r3	r4 ^ = r2		r0 & = r3	r4   = r3	
r2 ^ = r1	r3   = r4		r2 = ~ r2	r3 ^ = r1	
r3 & = r0	r4 ^ = r2		r1   = r0	r0 ^ = r2	
r3 ^ = r1	r1   = r4		r2 & = r4	r3 & = r4	
r1 ^ = r0	r0   = r4		r1 ^ = r2	r2 ^ = r0	
r0 ^ = r2	r1 ^ = r4		r0   = r2	r4 ^ = r1	
r2 ^ = r1	r1 & = r0		r0 ^ = r3	r3 ^ = r4	
r1 ^ = r4	r2 = ~ r2		r4   = r0	r3 ^ = r2	
r2   = r0			r4 ^ = r2		
r4 ^ = r2					
r4, r3, r1, r0			r3, r0, r1, r4		