# An Implementation of Bitsliced DES on the Pentium MMX™ Processor

Lauren May, Lyta Penna, and Andrew Clark

Information Security Research Centre,
Queensland University of Technology,
GPO Box 2434, Brisbane, 4000, Australia.
{lj.may, l.penna, a.clark}@qut.edu.au

**Abstract.** In this paper we describe an implementation of the DES (and Triple-DES) algorithm using the bitslicing technique on an Intel Pentium MMX processor. Implementation specifics are discussed and comparisons made with an optimised C-coded DES implementation and an assembly language DES implementation. This paper sets the scene for future research of the inter-relation between design and implementation of the newer 128-bit symmetric block ciphers.

## 1 Introduction

Symmetric block cipher design is, by necessity, influenced by the technology to be used in the implementation of the cipher. The primary goal of a cipher is to provide security, with its optimisation of implementation being a very important secondary consideration.

Early cryptographic processing was generally a bottleneck in communications and its implementation in hardware was necessary to optimise its speed. Today commodity microprocessors are available which can, in general, process software implementations of ciphers quickly enough for most purposes. Of course there are benefits of hardware devices for performing cryptographic processing such as the secure storage of keys due to tamper-resistance. However, there are numerous applications such as web browsers and email encryption utilities where ciphers are implemented in software and hence, here we focus on optimisation issues in software implementations of block ciphers.

Many optimisation techniques tend to be a trade-off between storage and speed. The storage of large amounts of information may be acceptable on platforms that are unrestricted by memory limitations but may not be possible on other platforms where the amount of available memory is minimal (such as a smart card).

Optimised cryptographic code is generally either purchased from a vendor or written in-house. People who optimise cryptographic algorithms, complemented by an appreciation of architectures and platform instruction sets, possess a very specialised skill. As technological advances and platform diversity proliferate the demand for these skills increases accordingly. Optimised code and methods for

achieving it are generally closely-guarded secrets, with information in the public domain being scarce. In this and consequent papers, the authors' contribution is to make our findings and applications available in the public literature.

The Data Encryption Standard (DES) [3] has been the symmetric block cipher standard since 1976 and is the most commonly studied cipher due to its widespread use. Many different techniques can be used to improve the efficiency of DES (for example [2]). One recent technique which indicates a path to faster implementation of some encryption algorithms is the use of bitslicing. This technique was applied by Eli Biham in 1997 [1]. Section 2 discusses some implementation and optimisation issues with the Pentium MMX platform. Section 3 compares the C, assembler and bitsliced MMX DES implementations and investigates three potential S-box algorithms. Section 4 discusses some quite restrictive limitations of MMX bitslicing regarding modes of operation and the platform itself, whilst Section 5 concludes the paper.

## 2 Issues in Implementing Bitslicing on a Pentium

Bitslicing enables the DES S-boxes to be represented in software by their logical gate circuits. Bitslicing, essentially, encrypts one bit at a time. If the implementation platform allows for parallel data processing then, in effect, a number of single-bits are encrypted simultaneously. Intel Pentium MMX technology is one such platform that uses a 64-bit register for parallel data processing. These processors are very relevant to cryptographic application as they are in such widespread use, particularly in commerce which is one of the heaviest users of cryptography.

In the case of a 64-bit processor, bitslicing enables sixty-four datablocks to be processed concurrently. This section is concerned with implementing the complete DES cipher using the bitslicing technique. The Pentium MMX bitsliced code in this paper was produced by using the Visual C++ compiler with in-line assembly.

In his paper Biham described a new optimised bitsliced DES implementation on a 300MHz Alpha 8400 consisting of thirty-two 64-bit integer registers. As was shown on such a computer, it is possible to gain a significant speed-up over standard implementations using the bitslicing technique as sixty-four datablocks are encrypted simultaneously. Biham's implementation ran three times as fast as the fastest implementation at that time (Eric Young's *libdes* [8] which was designed for 32-bit architectures).

Section 2.1 summarises the MMX instructions employed in the bitsliced implementation and also notes some subtleties in their use. Section 2.2 discusses some Pentium-specific optimisation techniques.

### 2.1 MMX Instructions

Intel's MMX architecture was designed specifically to enhance the performance of advanced media, multimedia and communication applications. It achieves this

through the introduction of new dedicated 64-bit integer registers, a new instruction set which allows for data parallelisation and a superscalar architecture which enables instruction-level parallelisation.

The MMX instruction set adds fifty-seven new instructions. The six instructions used in the MMX implementations discussed in this paper are:

| | |
|---|---|
| EMMS | enables the MMX registers to be aliased on the floating point unit |
| MOVQ | moves a 64-bit word between registers/memory and registers/memory |
| PXOR | binary EXCLUSIVE-OR |
| PAND | binary AND |
| PANDN | binary NOT followed by binary AND |
| POR | binary OR |

The EMMS, MOVQ, PXOR and POR instructions are straightforward and give no surprises in their application. The use of the PAND and PANDN instructions, however, contain subtleties which we now address.

The PAND instruction performs a binary AND on sixty-four bits of data from an MMX register or memory to an MMX register. For example, the code *PAND MM0,MM1* performs a bitwise logical AND of the contents of MM0 and MM1 and stores the result in MM0.

The PANDN instruction firstly inverts (complements) the sixty-four bits in an MMX register and then ANDs the inverted MMX register contents with another MMX register or memory. For example, the code *PANDN MM0,MM1* initially inverts the contents of MM0, and then performs a bitwise logical AND of the result and the contents of MM1. The result is stored in MM0.

Using the PANDN instruction requires multiple additional instructions when compared to storing a NOT value using C code and then using the PAND assembly instruction. This is because the value to be inverted must be in the first operand and this value is destroyed during the execution of the instruction, resulting in multiple additional instructions being required to retain the initial value in the first operand.

Bitslicing DES S-boxes requires the use of a bitwise logical AND and a bitwise logical NOT instruction that performs a bitwise logical NOT on a 64-bit datablock. As there is no MMX NOT instruction, it is necessary to use the PANDN instruction. This limits the processing speed of the S-boxes severely due to the multiple extra instructions required to maintain the state of the registers.

### 2.2 Pentium

The management of Intel's branch prediction technique is an important tool to be used in any attempt at optimisation on the Pentium. Branch prediction occurs when a probabilistic decision, based on past events, is made by the compiler as to the address of the next instruction. Optimising branch prediction techniques is most beneficial where the code uses *if/then/else* statements, particularly in connection with looping structures. In this paper's MMX implementation, the use

of *if/then/else* statements is kept to a minimum, so this optimisation technique is not beneficial to our implementation.

The dual-pipelining architecture in the Pentium enables two individual instructions to execute in two processors (the *u pipe* and the *v pipe*) simultaneously, referred to as pairing the instructions. Maximal pairing can result in significant reductions in the number of clock cycles needed to implement code sections. The *u pipe* is the master processor whilst the *v pipe* is the slave processor. If there is no pairing of instructions, all instructions are fed through the *u pipe* whilst the *v pipe* remains empty. If there is maximal pairing of the instructions, half the instructions are fed through the *u pipe* and the other half are fed through the *v pipe* simultaneously. Therefore the aim of optimising by pairing instructions is to write code in such a way that the compiler keeps both pipes full. This dual-pipeline instruction processing is called superscalar parallelism. There are certain restrictions on the type of instructions that can be paired, mostly to do with the complexity of the individual instructions and the limitations of the supporting hardware. This approach applied to each of the eight S-box implementations resulted in a reduction of, on average, 23.6% of the original number of clock cycles.

## 3 Comparison of DES Implementations

This section compares the C, assembly (Eric Young's C and assembly implementations [8]) and MMX implementations with respect to various techniques. Code optimisation is, in general, a function of both the algorithm and the implementation language conjointly. Several different techniques are presented and discussed in this section.

Section 3.1 discusses the initial and final permutations. As the implementation of the DES S-boxes is the overriding factor in its optimisation, several S-box Boolean function algorithms are investigated with regard to their suitability for MMX implementation. Section 3.2 discusses the applicability of MMX instructions to these S-box algorithms. Section 3.3 gives a single-round comparison. A complete DES implementation comparison is given in Section 3.4, while Section 3.5 addresses Triple-DES implementations.

### 3.1 Permutations

Permutations are inefficient in software. Patterns in the permutations make the use of look-up tables and streamlined operations possible. The SWAPMOVE technique, which we now describe, requires no memory and is extremely efficient for the IP, FP and PC1 permutations of DES. This technique is utilised in versions of DES available from the Internet (for example Eric Young's *libdes* [8]). Consider the following process:

| **SWAPMOVE(A, B, N, M)** |
|---|
| T = ((A >> N) $\oplus$ B) & M; |
| B = B $\oplus$ T; |
| A = A $\oplus$ (T << N); |

In this process the bits in B, masked by M, are swapped with the bits in A, masked by (M << N).

The IP can be performed using five SWAPMOVE operations, totalling thirty logical operations. This is an extremely efficient implementation technique. The permutation FP is performed by reversing the order of the five SWAPMOVE operations used in IP. PC1 can be performed in a similar manner, requiring sixty-four logical operations. The optimised C and assembler codes use the SWAPMOVE procedure, but the bitsliced implementation does not.

SWAPMOVE is a function based on the positions of the bits within a block. Although it is an extremely efficient technique for a C-coded implementation it cannot be used in the bitsliced implementation as the block bit positions are changed; therefore a standard IP permutation was used in the MMX code and, as can be seen in Table 1, it was faster than the optimised DES version that used the SWAPMOVE procedure. This speed-up is gained due to the 64-bit parallel processing bitslicing allows.

| Code | Clock cycles/ 64 DES data blocks |
|---|---|
| C DES | 2659 |
| Assembler DES | 1541 |
| Bitsliced DES | 148 |

Table 1: Initial Permutation (IP)

### 3.2 S-box Implementation

S-boxes that are implemented using Boolean functions are generally slower than when implemented using look-up tables however, when using them to process sixty-four word blocks in parallel, they give a more optimised S-box implementation.

There are multiple Boolean functions which can represent the DES S-boxes and be implemented efficiently. Schaumuller-Bichl's *Method of Formal Coding* [6], Shimoyama's algorithm [7] and Kwan's algorithm [4] were implemented for comparative purposes.

Kwan's algorithm is clearly the fastest when written in the C language (Table 2). All algorithms were also programmed using MMX instructions (Table 3). Kwan's algorithm remained the fastest, although there was also a substantial speed improvement for Schaumuller-Bichl's algorithm. Shimoyama's algorithm written in MMX assembly language (Table 3) did not show significant speed improvement due to the multiple variables involved in the algorithm.

A note on timings: With each new Intel Pentium processor some instructions execute in a reduced number of clock cycles. In our experience we found that the difference between the Pentium II and Pentium III was not substantial. The timings for all tables in this paper were performed on a 500 MHz Pentium III. We found the Pentium with MMX processor to be, on average, twenty percent slower than the Pentium II/III.

| | Code | Clock cycles/ 64 S-box 1 inputs |
|---|---|---|
| 1. | Schaumuller-Bichl (SB) | 243 |
| 2. | Shimoyama (S) | 235 |
| 3. | Kwan (K) | 150 |

Table 2: Comparison table of C-coded S-box algorithms

| | Code | Clock cycles/ 64 S-box 1 inputs |
|---|---|---|
| 1. | Schaumuller-Bichl | 137 |
| 2. | Shimoyama | 226 |
| 3. | Kwan | 90 |

Table 3: Comparison table of MMX-coded S-box 1 implementations

Table 3 clearly indicates that the most suitable algorithms for MMX implementation are Schaumuller-Bichl's *Method of Formal Coding* and Kwan's algorithm.

Using Schaumuller-Bichl's algorithm, S-box 1 was implemented in four different ways (Table 4).

| | Code | Clock cycles/ 64 S-box 1 inputs |
|---|---|---|
| 1. | Using variables and PANDN | 166 |
| 2. | Using minimal variables and PANDN | 245 |
| 3. | Using variables and store NOT values | 174 |
| 4. | Using minimal variables and store NOT values | 137 |

Table 4: Comparison table of S-box 1 implementations using the Schaumuller-Bichl algorithm

In Table 4 *Implementations 1* and *2* are compared as they differ only in the number of variables used, as opposed to the number of registers used. This comparison indicates that using variables to store data, as opposed to trying to store them in the registers, makes for faster implementation as well as easier coding. Alternatively a comparison of *Implementations 3* and *4* (which differ only in that the latter uses fewer variables) indicates that using fewer variables is faster. Therefore, in general, the effect of the number of variables used is inconclusive. In Table 4, *Method 4* was the fastest, therefore it was used for all eight S-boxes in the final DES bitsliced implementation using Schaumuller-Bichl's algorithms.

Kwan's S-box 1 implementation was faster than Schaumuller-Bichl's implementation (Table 3), therefore Kwan's algorithms for the eight S-boxes were also implemented for the final DES bitsliced comparisons.

Table 5 gives the timings for each of the eight S-boxes for both algorithms. For each of the S-boxes Kwan's algorithm produced the fastest MMX code. However, as will be discussed in Section 3.4, the full DES implementation using Kwan's S-box algorithms actually executed slower due to pre-processing and overheads in execution.

| S-box (64 inputs) | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| S clock cycles | 137 | 111 | 123 | 172 | 152 | 124 | 140 | 135 |
| K clock cycles | 90 | 95 | 87 | 80 | 101 | 98 | 96 | 88 |

Table 5: Eight individual MMX S-box implementations

### 3.3 A Single Round of DES

A single round of DES for each of the four approaches was applied and the results compared. Table 6 indicates that the bitsliced implementation using the Schaumuller-Bichl S-box algorithms is the most optimal.

| Code | Clock cycles/ 64 DES data blocks |
|---|---|
| C DES | 17786 |
| Assembler DES | 14255 |
| SB Bitsliced DES | 10597 |
| K Bitsliced DES | 18831 |

Table 6: One round

### 3.4 Complete DES implementations

The complete DES implementation includes all the previously-mentioned components. Table 7 indicates that the complete bitsliced DES using Schaumuller-Bichl's S-box algorithms is 68% faster than the complete C-coded DES and 35% faster than the assembler DES. Timings were performed on a 500 MHz Pentium III.

| Code | Clock cycles/MByte | Mbps |
|---|---|---|
| C DES | 72857172 | 54.90 |
| Assembler DES | 58389896 | 68.50 |
| SB Bitsliced DES | 43405661 | 92.15 |
| K Bitsliced DES | 77130624 | 51.86 |

Table 7: Complete DES implementation

It is of interest that, even though the implementations of each of the individual S-boxes is, on average, 33% faster using Kwan's algorithms (Table 5), the final DES bitsliced implementation (Table 7) using Kwan's S-box algorithms

was 45% slower than using Schaumuller-Bichl's S-box algorithms. This significant difference is attributed to the fact that Kwan's S-box algorithms require substantially more variables to implement (and hence more pre-processing and overheads) than Schaumuller-Bichl's S-box algorithms.

This finding indicates that, using MMX technology, it is important to balance the number of variables and the number of logic gates used in the algorithm due to the limited number of MMX registers and the limited instruction set. If the MMX platform had thirty-two MMX integer registers instead of the currently-available eight, the full DES implementation using Kwan's S-box algorithms should be faster than the Schaumuller-Bichl algorithms.

### 3.5 Triple-DES implementation

Triple-DES was implemented using the procedures from both bitsliced DES implementations. From a programming perspective, the only difference between Triple-DES and standard DES (other than tripling the number of DES procedures performed) is that Triple-DES does not require the IP and FP permutations between the first and second single DES, and the second and third single DES, since they are the inverse of each other. The initial permutation before the first single DES and the final permutation after the third single DES are still required.

The Schaumuller-Bichl bitsliced Triple-DES (Table 8) is the fastest implementation. Timings were performed on a 500 MHz Pentium III.

| Code | Clock cycles/MByte | Mbps |
|---|---|---|
| C DES | 183315583 | 21.80 |
| Assembler DES | 140440785 | 28.48 |
| SB Bitsliced DES | 127064841 | 31.48 |
| K Bitsliced DES | 185454810 | 21.00 |

Table 8: Triple DES implementation

## 4 Limitations

There are some limitations in using bitslicing both in general and also on a Pentium MMX platform. The bitslicing technique has some restrictions in general in respect to common modes of block cipher implementation, which are discussed in Section 4.1. The Pentium MMX platform has some restrictive limitations for the bitslicing application when compared to the Alpha 8400 (Section 4.2), but it is in much more widespread use than the Alpha and so is a more acceptable block cipher implementation platform. Section 4.3 explains the requirements of data-format conversion for the bitslicing technique.

### 4.1 Modes of operation

Bitslicing implementations are suited to encryption and decryption using *Electronic Code Book (ECB)* mode. Since sixty-four data blocks are encrypted or decrypted simultaneously using the bitslicing technique, the method is not suitable for use with traditional block cipher modes where the current data block is reliant on the output from a previous data block encryption or decryption. These modes are *Cipher FeedBack (CFB)* encryption, *Cipher Block Chaining (CBC)* encryption, and *Output FeedBack (OFB)* encryption and decryption. With both the *CFB* and *CBC* modes the ciphertext is available (without the necessity of having to reproduce it in the decryption process), so the bitslicing implementation can be used for decryption in these modes. Traditional feedback and chaining modes of operation are used in most applications of DES, so this is a major limitation of the bitslicing implementation technique.

In his paper Biham proposed a CBC-like mode where the IV is 4096 bits and the entire 4096-bit ciphertext (64 $\times$ 64-bit blocks) block is XORed with the ensuing 4096-bit plaintext block. This mode would overcome some of the shortcomings associated with the *ECB* mode of operation.

There would, however, be security problems associated with using this CBC-like mode operated on a 4096-bit block. For example, the traditional CBC mode operated on the standard 64-bit block offers a means of producing a Message Authentication Code (MAC). One of the intrinsic security features of the MAC (produced by DES encryption in traditional 64-bit CBC mode) is that the MAC is a function of every bit of the message or, conversely, every bit of the message used to generate the MAC is diffused across the entire MAC. In the case of a 4096-bit CBC-like mode this would not be the case since diffusion of a particular bit of the message would be restricted to a particular 64-bit subblock and not the entire 4096-bit block, as is a basic requirement of a MAC.

*ECB* mode however is still used in some applications, and these applications will directly benefit from the DES bitslicing implementation.

### 4.2 Pentium MMX platform

A limiting restriction of the Pentium is that only eight MMX registers are available. A DES bitsliced implementation requires a minimum of six registers, one for each S-box input bit $x_j$. The remaining registers are used as accumulators and for other intermediary tasks. In the case of the Pentium this leaves only two registers for these housekeeping tasks, which is insufficient for producing optimised code.

Where the available MMX registers are inadequate for the necessary tasks the programmer must use non-MMX 32-bit registers. Moving data to and from the non-MMX registers requires more use of the MMX move command which increases the number of instructions necessary in the implementation, as well as the extra overheads involved with swapping between a 64-bit and a 32-bit word size. The Alpha 8400, having four times as many integer registers as the Pentium MMX, would not have these restrictions.

Another restriction on programmers is that the MMX instructions accept only two operands. This necessitates more frequent moving of data within the registers and, consequently, produces code with more instructions. The restrictive PANDN instruction and the absence of a 64-bit logical NOT instruction (Section 2.1) are also limiting factors.

Biham's fastest DES bitsliced implementation was 137 Mbps on a 300MHz Alpha 8400 processor. The fastest 500MHz Pentium MMX bitsliced implementation in this paper is 92 Mbps, which is a respectable result given the limitations of the platform and its instruction set, especially the minimal number of MMX registers.

### 4.3 Data conversion

The data for the bitslicing implementation needs to be read into the registers in a non-traditional format. The inverse of this initial reformatting must also be performed at the end of the data processing. There is an overhead cost involved with this reformatting. We did not include the cost of data preparation or reformatting in any implementation. When used in situ, the code would be executed by a call from the Application Program Interface (API) which would incorporate any necessary reformatting before and after encryption and decryption.

## 5 Conclusion

In this paper the technique of bitslicing is applied to the DES algorithm on the very widely-used Intel Pentium MMX processor at a speed of 92 Mbps. Pentium implementation and optimisation issues are discussed. Some application limitations of bitsliced DES are identified, and a comparison of the C, assembler and bitslicing approaches is made.

Our research goal is concerned with implementation issues of symmetric block ciphers and how they affect the design of ciphers.

In this paper we have discussed a wide range of implementation issues of the DES symmetric block cipher. Our future research will concentrate on investigating implementation issues with respect to the new 128-bit block ciphers (for example, the new AES candidates [5]), and how these implementation issues affect the design decisions of the ciphers themselves. This work will build on the framework set out in this paper.

## References

[1] E.Biham, *A Fast New DES Implementation in Software*, 4th International Workshop, FSE'97, Israel, January 1997; Proceedings, Lecture Notes in Computer Science, Springer, Vol 1267, pp 260-271.

[2] M.Davio, Y.Desmedt, J.Goubert, F.Hoornaert and J.J.Quisquater, *Efficient Hardware and Software Implementations for the DES*, Proceedings of CRYPTO'84, Lecture Notes in Computer Science 196, Springer-Verlag, pp 144-146.

[3] Federal Information Processing Standards Publications, FIPS PUB 46-1, *Data Encryption Standard*, USA.
[4] M.Kwan, *Bitslice DES, S-box Implementation*, March 1998, http://www.darkside.com.au/bitslice/sboxes.c .
[5] National Institute of Standards and Technology (NIST), *Request for Candidate Algorithm Nominations for the Advanced Encryption Standard (AES)*, Federal Register, Volume 62, Number 177.
[6] I.Schaumuller-Bichl, *Cryptanalysis of the Data Encryption Standard by the Method of Formal Coding*, Advances in Cryptology, EUROCRYPT'82 Proceedings, Springer, Verlag, 1992, pp 235-255.
[7] T.Shimoyama, S.Amada, S.Moriari, *Improved Fast Software Implementation of Block Ciphers*, 1st International Conference, ICIC'97, China, November 1997, Proceedings: Lecture Notes in Computer Science, Vol 1334, pp 269-273.
[8] E.Young*libdes*, http://www.SSLeay.org.