

Preparing Tomorrow's Cryptography: Parallel Computation via Multiple Processors, Vector Processing, and Multi-Cored Chips

Eric C. Seidel, advisor Joseph N. Gregg PhD

{seidele, greggj}@lawrence.edu

May 13th, 2003

Abstract

This paper focuses on the performance of cryptographic algorithms on modern parallel computers. I begin by identifying the growing discrepancy between the computer hardware for which current cryptographic standards were designed and the current and future hardware of consumers. I discuss the benefits of more efficient implementations of cryptographic algorithms. I review one algorithm, the US Data Encryption Standard (DES), in great detail. As an example of potential changes to cryptographic implementations, I offer my own faster “Bitslice” implementation of DES designed for the Motorola G4 with AltiVec Vector Processing Unit – an implementation which completes some tests up to nine times faster than libdes (currently the fastest open source DES implementation for the G4). Then I examine two other cryptographic algorithms and discuss methods by which they too can be efficiently implemented on modern computers. Finally, I conclude with a brief discussion of very recent cryptographic algorithms (the AES candidates) and their potential success on tomorrow's parallel computers.

Contents

1	Cryptography: A Brief Introduction	1
1.1	The Future of Cryptography	3
1.2	Making a “Modern Cryptography”	6
2	Implementation	9
2.1	Secret Key Cryptography Overview	10
2.1.1	The US Data Encryption Standard (DES)	12
2.2	Understanding DES	13
2.2.1	Sub-Key Generation	13
2.2.2	Data Encryption	17
2.2.3	The f Function and S-Boxes	19
2.2.4	DES Modes & Decryption	23
2.2.5	3DES	26
2.3	Understanding Bitslice-DES	27
2.3.1	The Difference of Hardware DES	29
2.3.2	Bitslice Implementation Changes	30
2.4	The AltiVec Vector Processing Unit (VPU)	31
2.4.1	Swizzling on the AltiVec	38
2.5	Performance Testing	41
2.5.1	Swizzling Tests	43
2.5.2	Head-To-Head Tests	45
2.5.3	Swipe Size Tests	47
2.6	Assurance Testing	49
3	A Greater Context	51
4	Applying Parallelism To Other Crypto Algorithms	57

4.1	Parallel Cryptography, Today	58
4.2	Hashing Algorithms: MD5	59
4.2.1	Message Digest Algorithm Revision 5 (MD5)	60
4.3	Public-Key Cryptography: RSA	62
4.3.1	The Rivest-Shamir-Adleman (RSA) Method	63
5	Final Thoughts	65
A	Computer Science Background	71
A.1	Alternative Number Systems	71
A.2	Memory Storage	72
A.3	A Little Logic	73
B	Package Listing & Descriptions	76
C	Source Code	77
C.1	generate_bitslice_des.pl	77
C.2	Excerpts from bitslice.c	90
C.3	generate_swizzle_vpu_c.pl	97
C.4	swizzle_iu.c	102
C.5	swizzle_vpu.h	108
C.6	Excerpts from swizzle_vpu.c	112
C.7	main.c	116
C.8	generate_bs_speed_tests.pl	136
C.9	generate_swizzle_speed_tests.pl	139
C.10	swipe_tests.c	141
C.11	altivec_sboxes_c.pl	143
C.12	swap_endian_bitslice_c.pl	144
C.13	Excerpt from kwan.c	145

D Program Output	148
D.1 Usage Statement	148
D.2 Sample Output “-S”	148
D.3 Sample Output “-W”	149
D.4 Sample Output “-E”	149
D.5 Sample Output “-L”	150
D.6 Sample Output “-P”	151

List of Tables

1 High-Level Results Summary	10
2 Altivec Boolean Instructions	33
3 Altivec Permute Instructions	34
4 Altivec Data Stream Instructions	37
5 CHUD Performance Tools	43
6 Performance Testing Flags	44
7 Integer Unit vs. Vector Unit Swizzling	44
8 Results from DES - ECB Tests	46
9 Test Data from DES Swipe Size Tests	48
10 Assurance Testing Flags	51
11 Binary, Decimal and Hexadecimal Conversion Table	72
12 Logical operators	74

List of Figures

1 f Function Overview	20
2 S-Box #1 as a lookup table	22
3 Register Usage: DES vs. Bitslice DES	29

4	Swizzling eight 8-bit blocks on 8-bit registers	31
5	<code>vec_sll</code> Instruction Diagram	33
6	<code>vec_merge1</code> Instructions Diagram	33
7	<code>vec_sel</code> Instructions Diagram	35
8	<code>vec_perm</code> Instruction Diagram	35
9	<code>interleave128</code>	69
10	Demonstration of 8-bit Interleave	70
11	Bits and Bytes	73

Foreword

I began this project in fall term 2002-2003 as a method by which to further familiarize myself with modern cryptographic algorithms and using them in manners of high efficiency. My interest in parallelism grew out of my original research, as did the idea for my final project: my implementation of Bitslice DES. I have previously been fascinated with both computational performance and the mathematics of cryptography. This research satisfied both interests.

In this paper, I assume no background in cryptography, for I build the necessary context throughout. This work is laid out into five sections: Section 1 offers an introduction to cryptography and some context/justification for the work I have done here. Section 2 describes my implementation, its results, history, and the technical details of the algorithms DES and Bitslice DES and the hardware on which they are implemented. Section 3 offers some further technical and academic context for this work. Section 4 describes the research from which this project and paper began, the various methods I initially proposed for applying parallelism to legacy algorithms, and some applications of these methods to a few representative cryptographic algorithms from two areas of cryptography. Section 5 offers some closing comments regarding my work and some of the newest algorithms in cryptography. I have also attached some appendices containing helpful background information for those who are not computer scientists, the full source code of my implementation, some sample output from my implementation, and a full listing of the implementation's package contents.

My work brings together techniques from throughout recent cryptographic literature dealing with fast cryptography and demonstrates one example of that work. The following is written to be understandable by any educated person, and it does not require a prior knowledge of computers or their inner workings. Those not from a computer science discipline are encouraged to consult Appendix A: Computer Science Background, either before or throughout reading the paper.

1 Cryptography: A Brief Introduction

Cryptography (also referred to as “crypto”) is the science of keeping secrets. These secrets are not those kept behind locked doors or in secret passageways; rather, cryptography deals with keeping valuable information secret even when an “encrypted” form of that information is left in the open. Cryptography is a kind of secret-displacement: that which user keeps hidden is no longer the information itself, but instead the (much smaller) secret key with which to unlock that information. Cryptography as such is not a new science, but rather one which has been around for millennia – as long as humans have wanted to keep secrets from one another. Cryptography has changed much since its origins, particularly in the last 50 years, and even more so in the last five. It is now a modern, computer-aided cryptography with which we concern ourselves today.

To give you a little history: As far back as the Romans we have records of those such as Julius Caesar using cryptography. Caesar is famous for encoding the messages he sent to his generals by shifting the alphabet in which those messages were written. A simple example could be “BUUBDL OPX” translated “ATTACK NOW”.¹ It is fitting that this example deals with war, as cryptography, throughout history, seems particularly motivated by human conflict. World War II and the later US/USSR cold war are two great motivators from the last century. Machine-aided secret keeping came to the forefront during WWII with Germany’s Enigma machine.² Cold war spending and the advent of the computer saw the creation of modern computer-based ciphers, such as the United States’ Data Encryption Standard (DES) and the Soviet GOST algorithm[20, 6]. Cryptography in the recent years however, has taken a turn away from government, instead finding uses for businesses and consumers. With the advent of Public Key Cryptography and much more powerful³ con-

¹This is a single alphabetic rotation – A = B, B = C, etc.

²A mechanical device consisting of basically a typewriter, mechanical wheels and a set of lights. The user would consult a special code book, set the wheels to the starting position for the day and then type their message on the keyboard. The lights would light up with the corresponding translation of each letter. The Enigma code was eventually broken by allied forces late in the war.

³To give you some idea of the modern power of computers, my own year-old laptop is capable of a peak computational throughput of over four GigaFlops – four billion Floating Point Operations Per Second (Flops)

sumer computers, the consumer has found a new role in cryptography. This paper addresses a modern consumer-centered cryptography and discusses how computer scientists might go about making cryptography ready for efficient use on today's personal computers.

Before I begin general discussion, let me offer a brief list of domain specific terms:

Hashing & Hash functions: Hashing is the process of taking a large block of data (or a large number) and reducing it to a much smaller block of data (smaller number) representing the larger block. This is done in such a way that any small change in the original data results in a large change in the computed "hash." The net effect here is that the smaller "hash" can be used to uniquely identify that larger block of data, and also ensure the integrity of that data because any change in the original data should produce a different hash.

Encrypting & Ciphers: Encrypting is the process of converting a message (otherwise known as "plain-text") into a corresponding block of secret code (otherwise known as "cipher-text"). This encryption is accomplished through the use a specific "encryption algorithm" (also called a "cipher") and a special block of data called a "key" (or "key-text"). The key-text and plain-text are fed into the cipher and the appropriate cipher-text is returned. There are several different types of ciphers capable of performing such a conversion. Two types mentioned in this paper are "block" and "stream" ciphers, which correspondingly take plain-text data either divided into short blocks or as a long continuous stream. Both "encrypt" this plain-text data into corresponding cipher-text.

Secret (Symmetric) Key vs. Public (Asymmetric) Key Cryptography: There are two popular divisions of cryptography. Secret (Symmetric) Key cryptography in which a user has only one key which can both encrypt and decrypt a message, and Public (Asymmetric) Key cryptography in which a user has both a public key and a private key. A public key is used to encrypt messages and verify signed messages. A private key is used to decrypt and sign messages. The user can distribute the public key openly and keep the private key secret.

– a number four times the original "super computer."

1.1 The Future of Cryptography

From bank accounts to medical records and personal emails, every day more sensitive data are stored and transported digitally. With the continued growth of the Internet, more and more of these data reside on systems or are transferred over networks which themselves are neither physically nor digitally secure. To help solve these problems of digital data security, we have cryptography. Most cryptography has historically been used by governments, larger business, and computer geeks but not by the average consumer. Needs however, are now shifting, and consumers are using secure web connections, encrypted emails, encrypted file systems, and smart cards.⁴

Cryptography can already be done quite quickly on modern computers. My laptop⁵ can encrypt on average around 500,000 64-bit⁶ blocks per second. That's a cryptographic throughput of around 30 million bits (megabits) per second (Mbps) or 3.7 million bytes (megabytes) per second (MBps).⁷ The fact that libdes⁸ can average this throughput is a tribute to cryptography's speed. That said, the cryptography in use today by libdes and other implementations is not efficient and not near what should be expected of modern computers.⁹

⁴Smart cards are normal credit-cards or identification cards which carry a special micro-computer chip. Smart cards are physically secure devices designed to hold protected personal data in a cryptographically secure format. These devices generally will hold a unique public/private asymmetric cryptography key pair, used to uniquely identify the bearer. These cards are mentioned here as they have the potential to further bring cryptography into the mainstream. Every bearer will have a unique cryptographic key, allowing businesses to more easily support security for the consumer.

⁵Apple Titanium PowerBook, 550Mhz PowerPC G4, 756 Megabytes of RAM, 100Mhz system bus.

⁶The terms bit and byte are used here without explanation and refer to small quantities of computer storage. For a full description of meanings, turn to Appendix A.1.

⁷To get a sense of the speed here, a normal dial-up connection is 56 Kbps (kilobits per second) or 7 Kbps (kilobytes per second), broad-band internet is more on the order of 760 Kbps, a local area network more like 10 - 100 Mbps, and fast networks upwards of 1 Gbps or one billion bits per second transfer rates.

⁸Libdes, as mentioned in the abstract, is the fastest open source implementation of DES encryption on the Motorola PowerPC G4 processor. Libdes, pronounced "lib-dez", was originally written by Eric Young (now of RSA Security) back in 1993, and is generally regarded as one of the fastest implementations of DES available. Libdes is a library of functions which perform DES encryption in all commonly supported DES modes, as well as 3DES encryption in those same modes. Much more information regarding DES and its modes is available in Sections 2.1 and 2.1.1.

⁹I would at least expect that modern computers, should be able to encrypt data at at least half the speed at which they could write it out. This is however not nearly the case here. My laptop is capable of communicating over its network card at one billion bits per second – a speed over 15 times the current top

The consequences of this lack of efficiency are many. Foremost, inefficient implementations are expensive for high-end users such as large web sites who must purchase tens to hundreds if not thousands of computers to handle requests from millions of visitors. To them, supporting secure connections is a costly endeavor that requires proportionally much more computational power than a non-secure connection. Consumers too are affected by this lack of efficiency: especially as digital security becomes more prevalent, a customer's use of VPN¹⁰ software, encrypted disk images, or other security software should not negatively affect the rest of his or her computer usage, as it can today. They should not be sacrificing network transfer speeds, Quake III frame-rates, or other performance simply because of inefficiently implemented cryptography. It is the cryptographer's responsibility to correct this inefficiency.

These inefficiencies primarily stem from the fact that prior to the 1998 solicitation for the American Encryption Standard (AES), the cryptographic world was designed around computers with a single, 32-, 16-, or even 8-bit processor. Increasingly the computer world of today, and most definitely that of tomorrow, is not one of the 32-bit desktop, rather one of multi-cored chips,¹¹ multiple processor machines, and larger 64- or even 128-bit processors, many with a Vector Processing Unit (VPU).¹² This is a large change in the machinery of the consumer, and cryptography must be made ready for this change.

Making crypto ready for tomorrow's architectures not only solves current speed problems, but also opens cryptography to a whole new range of uses. Already we are seeing

speed of DES on the PowerPC.

¹⁰Virtual Private Networks (VPNs) – these are encrypted “virtual” networks built on top of physical networks such as the internet. These allow a group of computers to build a “virtual” network consisting solely of encrypted communications which only the computers on that virtual private network can read.

¹¹Placing multiple processor cores on the same piece of silicon. Manufacturers use this to reduce drastically the cost of having more than one processor. They reduce cost associated with the amount of silicon used and the cost of all the additional architecture (buses, memory, caches, etc.) associated with a completely separate processor. Itanium (Intel's new 64-bit processor) based, multi-cored chips are scheduled to ship by 2005 and IBM already ships a multi-core version of its high-end POWER4 processor.

¹²In contrast to scalar processing, a Vector Processing Unit (VPU) works on “vectors” of data, and performs the same operation (add, multiply, AND, OR, etc.) over a uniform set of data, just as would be performed on a unit, except now multiple units are worked on (and completed) all in a single span of time. This method of applying parallelism is commonly referred to as Single Instruction, Multiple Data (SIMD) computing.

interesting new applications based on fast implementations of AES such as Apple Computer's encrypted disk image technology: an encrypted virtual disk that can be read nearly as fast (less than 10% difference) as unencrypted disk access because of an efficient AES implementation on their hardware.¹³ Such technologies will soon become the norm, not the exception. With fast enough cryptographic algorithms, all data written out (to storage media, networks etc) could be done so in an encrypted fashion. The number of simultaneous secure connections the average consumer has open will continue to increase with things such as encrypted chat, secure video streaming, secure email, VPNs, and secure connections to handheld computers, cell phones, and other devices. Until cryptographic algorithms are ready to be performed at great speeds on the computers of today and of the future, many of these applications listed remain slow and difficult, if not impossible with current algorithms and implementations.

This leaves us then with an interesting problem. We have a world increasingly in need of greater crypto speed, one which is at the same time undergoing radical changes in computer hardware architecture, and yet one which still uses 28-year-old crypto algorithms.¹⁴ We are entering into a world in which parallel-ready software is a must, and it is thus time that our cryptographic software be brought into the 21st century. Some, perhaps much, of this shift to better, more flexible crypto has already begun through an influx of new algorithms via the AES solicitation.¹⁵ But recognizing that not all systems will be as quick to change, and that often new cryptographic algorithms take many years, if not decades to be accepted, it is important to explore what, if any, changes and amendments we can make to existing cryptographic standards to bring them into the future. I will discuss several ways of making these changes in this paper, as well as provide my own example of these changes to the DES

¹³I was fortunate enough at Apple's World Wide Developer Conference (WWDC) 2002 to see Steve Jobs demonstrate play-back of a high data-rate movie from such an encrypted disk image.

¹⁴Here I refer to DES, which was initially designed in 1974 and is still in use today.

¹⁵In 1998, the National Institute of Standards and Technologies (NIST), seeing that DES and 3DES encryption were no longer viable encryption solutions long term (due to a number of reasons – some discussed in this paper), made a public solicitation asking for candidates to become the new American Encryption Standard (AES) block cipher. Many candidates were entered, five were selected as finalists. How well those finalists fare on modern computers is discussed more in Section 5.

algorithm.

1.2 Making a “Modern Cryptography”

The least explored and yet the most lucrative target for reaping improvements in security speed is not changes to the operating system, nor to the security applications, but changes to the implementations of the algorithms themselves. Moving down to the lowest level of security software design allows us to exploit fully some of the growing technologies on the market today. Many CPUs, including Motorola’s G4 and Intel’s Pentium 4, already ship with VPUs (the AltiVec Engine, and the MMX/SSE/SSE2 units respectively), making vector processing power available to the consumer, yet few cryptographic implementations support these. In addition, Intel has promised to begin shipping a multi-cored version of its new Itanium processor by the year 2005, IBM already ships a multi-cored version of its POWER4 processor, and Apple and most other computer manufacturers ship multiprocessor machines in their desktop and server product lines. Cryptographic algorithms in general make no accommodations for this parallel processing, neglecting possible gains under these multiprocessor environments. In order to exploit these technologies fully, we can no longer depend on the flexibility of operating systems, or the seemingly unending megahertz climb. Rather we must redesign our cryptographic implementations to utilize these current and future computing architectures.

Embracing parallelism with modern implementations can allow better performance in a number of ways. I have listed three important ways below – ways which will be discussed in this paper.

1. By performing the *same calculation on a larger amount of data*. Performing the same calculation on large amounts of data concurrently is the technique most discussed in this paper and is the technique used by Vector Processing Units and SIMD architectures. Multi-cored chips and true multiple processor architectures can also use this type of parallelism by performing the same algorithm multiple times in parallel on

several processors. Utilizing the advantages of this type of computing is important for cryptography because it is these SIMD or VPU architectures which are the most common form of parallelism available on modern computers.

2. By performing *two distinct parts of a single algorithm at once*. This is only possible in true multiple processor environments, and is accomplished by allowing multiple individual processors to handle separate parts of an algorithm at the same time. A common technique of this type is pipelining: sending data from one processor to the next down an assembly chain of sorts. Pipelining can allow the computation of n sequential steps of the algorithm (in parallel) over a single clock cycle on n processors. An example of this is to let each processor do a single cryptographic round on data passed to it from a high data-rate network stream. If each processor is able to complete a single round of the cipher in time t , we can add n more rounds of encryption to our final cipher-text within the same time t by adding n processors to the pipeline[1]. By doubling the number of processors we can in effect double the security of the data stream with no effect on data-rate. Other techniques of this type often require specific algorithm design modifications and introduce processor scheduling concerns, and therefore remain less common.
3. By *making a single complex calculation faster* by distributing load over multiple processors or using parallel technologies such as VPUs or SIMD instructions. This is actually a layer below algorithm design, and depends on the implementations of the library¹⁶ from which the algorithm draws. This is useful in areas of cryptography where mathematically intensive operations are performed over large data sets. A good example of such an area is Public Key Cryptography (PKC). PKC requires the execution of extremely large mathematical operations. Math speed gains in PKC can be

¹⁶A library in this sense of the word is a collection of pre-packaged functions which a computer program can call to have the computer perform certain operations. These are generally common functions that programs use that are too complex (and not common enough) to warrant a direct implementation in hardware. Libdes is such a library containing functions which perform cryptographic operations. The implementation I describe in this paper could also be made into a library and distributed to other programmers.

exploited from any VPU or set of processors as long as one has the knowledge and/or the vendor-supplied math libraries to take advantage of the parallel processing power of those systems.

There are also a few common techniques and pitfalls for applying parallel processing to various cryptographic algorithms that deserve mention here prior to the discussion of the details of my implementation.

- *Hardware in Software* - Sometimes when moving from a system designed for a single smaller processor to an architecture including larger processors (or parallelism of any form) it is useful to look backward before proceeding forward. Such was the work of Eli Biham,¹⁷ when he noticed that speed gains could be achieved for DES by implementing the hardware (logic gate¹⁸) version of DES in software running on 1-bit or larger processors. Biham noticed that substantial speed could be gained by viewing a larger-than-one-bit processor as an array of 1-bit processors, and performing the DES algorithm according to the logic gate implementation in parallel over those 1-bit processors. This approach is commonly referred to as the “Bitslice” implementation and is described in much greater detail in the rest of this paper. Bitslice ideas can also have applications in a large range of cryptographic algorithms.
- *SIMD on any processor* - Another technique used when moving from a smaller processor (or single processor) to a larger (or multiple) processor(s), is to view the larger processor as an a processor designed for SIMD operations the size of the original smaller processor (even if the larger processor was not designed for such). This allows application of parallelism to an implementation at the packet-level (file-level), by computing two or more instances of the same algorithm at the same time across multiple packets or files all on the same processor. This implementation is only efficient under certain algorithmic

¹⁷The work I refer to here is Biham’s “A fast new DES Implementation in Software” [4] which is discussed at great length throughout this paper.

¹⁸Mathematical logic and logic gates are discussed in Appendix A.3

design constraints and fails in circumstances where parts of a single processor register must be treated differently based on their smaller internal values.¹⁹ This method of SIMD on any processor can be very effective but depends heavily on the processor on which (and the algorithm for which) it is implemented.

- *The problem of chaining* - Many cryptographic algorithms, in order to achieve increased security, or simply by their fundamental design constraints (e.g. hashing), involve chaining of information from one cipher-block to the next, introducing “recursive dependency”²⁰ into the algorithm. This dependency makes applying block level parallelism to the algorithm impossible and will be seen in many algorithms.

The implementation which I offer in this paper is a prime example of the “hardware in software” technique.

2 Implementation

As an example of applying some of these aforementioned principles of optimizations, particularly the usage of Vector Processing Units and the “hardware in software” idea, I have written my own implementation of DES. I have chosen to implement a variant of DES called Bitslice DES. My implementation of Bitslice DES runs some tests up to nine times as fast as the current fastest open source DES implementation on PowerPC hardware and faster than commercial hardware implementations of DES. Table 1 offers some high-level comparisons of my implementation’s performance.²¹

¹⁹There can be workarounds to these limitations, but those workarounds often lose much efficiency. Using lookups as an example, lookups could be translated into much larger entire-register lookups, but the tables required for such can be enormous. This implementation runs into difficulties with operations such as rotations, multiplication and addition. Rotates, multiplies or adds performed over groups of data stored on larger registers may require significant intra-register adjustments.

²⁰Lacking a discipline-standard word, I will refer to the round-to-round and block-to-block dependency of some functions in various algorithms as “recursive dependency” – hinting to the dependency introduced by applying the function to the same (or parts of the same) data in a recursive fashion.

²¹Here “random” data refers to simulated real-world data whereby each plain-text block is taken from a random data stream. Statistics using random data include the entire cost of running the implementation and represent real-world sustained throughput. “Static” data statistics, on the other hand, neglect certain

Architecture	Implementation	Processing Unit	Data	MB/s
G4 550Mhz	libdes	32-bit IU	random	3.2
G4 550Mhz	Bitslice	32-bit IU	random	3.1
IBM	logic gate	hardware	random	18.3
G4 550Mhz	Bitslice	128-bit VPU	random	11.8
G4 550Mhz	libdes	32-bit IU	static	3.2
G4 550Mhz	Bitslice	32-bit IU	static	11.8
P3 500Mhz	MMX Bitslice	64-bit VPU	static	12.0
Alpha 300Mhz	Alpha Bitslice	64-bit IU	static	17.1
G4 550Mhz	Bitslice	128-bit VPU	static	32.8

Table 1: High-Level Results Summary

Table 1 shows the much improved performance which my implementation offers over both DES running on other modern processors and over any previous implementation of Bitslice DES. What I offer here is perhaps the first implementation in which Bitslice DES finally moves out of the purely theoretical realm and enters as a day-to-day useful implementation with improved software encryption speeds. The following sections first give an in-depth description of the DES and Bitslice DES algorithms, followed by a description of some of the optimization and testing measure which I employed.

2.1 Secret Key Cryptography Overview

Before detailing the DES algorithm, it is useful to look at Block Ciphers, the broader category of algorithms to which DES belongs. Block ciphers are the most common type of cryptography, and are used for many general purpose tasks including encrypting large sets of data, generating large random numbers, and generating another class of ciphers called “stream ciphers.”²² Block ciphers are the truly the work-horse of cryptography.

hidden costs associated with processing real data. Static data statistics are provided for comparison with other research implementations such as MMX-Bitslice[15] and Biham’s Alpha Bitslice[4]. Static data is useful for showing the real peak performance of Bitslice, but neglects concerns present during real-world usage. Further discussion of the various performance measures of my implementation can be found in Section 2.5.

²²Stream ciphers are used for very long, continuous sets of data such as multimedia streams. Stream ciphers function by starting with a secret key, and an initial seed value. Encryption is performed repeatedly on the seed (the seed is used as the plain-text), each time feeding the cipher-text back into the algorithm as the new seed value. Every block of data generated by the cipher in this fashion can be used with an XOR

Secret key cryptography works at its most fundamental level by applying a non-linear mathematical formula to a block of data, XORing the result with some secret key, often permuting the bits around, and then repeating this process several more times. The reason why this does not just produce an (permanently) unintelligible jumble of data is that both the non-linear formula and the XOR operation have an inverse. In fact they are (normally) their own inverse. Thus beginning with a jumble of data, and the secret key, this process can be re-applied to the jumbled bits to reveal the original message. Someone who does not have access to the secret key will have no idea what data to use when applying this process (attempting decryption). It is this lack of knowledge (the secrecy of the key),²³ which makes ciphers such as DES secure. In the example of Bitslice, we will apply parallelism by performing the same encryption algorithm on several blocks of data at once.

When encountering the problem of parallelism in block ciphers, block ciphers such as DES have two key factors affecting the ability to answer the problem of applying parallelism. One of those factors is the mode in which one uses the cipher, and the other relates to the block size of the cipher itself. The mode question will be addressed at length in Section 2.2.4 when I discuss implementing the various modes. Block size affects one's ability to apply parallelism to an implementation because any time the block size is smaller than the registers of the computer with which we wish to compute the algorithm we must devise a way by which to process multiple blocks simultaneously in order to achieve full computational efficiency. Furthermore, which particular mathematical or other computational operations are involved in the algorithm affects our ability to add parallelism to its implementation. Bitwise operations²⁴ are particularly easy to do in parallel across multiple smaller blocks,

operation (see Appendix A.3) to “encrypt” a piece of the data stream. Readers interested in a more in-depth discussion of stream ciphers and their usage should consult Schneier[24].

²³The reader might be curious to know how the secrecy of the key-text is maintained in this process. The secrets of the key are kept safe both through the recursive application of this process onto the same data and through the fact that the original plain-text is not generally known. If either of these were not the case – we only used a few rounds of the algorithm, or we knew a whole list of cipher-text plain-text pairs – there are ways of discovering the secret key.

²⁴Operations which operate on individual bits. This is in contrast to other operations which manipulate byte or multi-byte values.

whereas multiplication and lookups are not as easy when performed as SIMD operations.

2.1.1 The US Data Encryption Standard (DES)

DES is by far the most common of the block ciphers, used for much of the encrypted communications and encrypted data storage throughout the world today. DES is a 64-bit block cipher which belongs to a class of block ciphers called Feistel networks. In Feistel networks, plain-text blocks are divided into equal sized high and low components; one component (for this example, the low component) has a non-linear function applied to it and the result of that application is exclusive OR'd (XOR'd)²⁵ with the other component (here, the high component). DES suffers on modern processors not only from a 32-bit dependency,²⁶ but also from inefficiencies in its standard 32-bit implementation which cause often only four or six bits of each 32-bit register to be used,²⁷ thus only running at 12-16% efficiency on 32-bit hardware, half that on 64-bit processors. DES was designed back in 1974 (well before the personal computer) and was originally intended for only a few years of use[11]. DES has survived over 28 years however, and is still regarded as cryptographically secure, even if it is limited by a short key length and small block size[30]. Many, many block ciphers which have followed DES also share many of DES's ideas; thus, DES is a supreme choice for discussion.

As DES is by far the most commonly used block cipher, there have been many attempts to make it faster. These have included several attempts at applying parallelism to DES implementations, including a most ingenious suggestion by Eli Biham, commonly referred to as Bitslice DES[4]. The Bitslice idea has also been employed in various other modern

²⁵For an explanation of boolean operations such as XOR (exclusive OR), please consult Appendix A.3.

²⁶The algorithm itself is 32-bit dependent because the half-blocks (half of the original 64-bit plain-text block) are always 32-bits in size. 32-bit dependency means here that although these 32-bit half-blocks can be stored efficiently on processors smaller than 32 bits wide, these 32-bit half-blocks can not be stored efficiently (without leaving a part of each register unused) on processors with registers larger than 32-bits. Thus the algorithm is dependent (or works best on) processors with registers 32 bits wide or smaller.

²⁷This inefficiency is due primarily the part of the DES algorithm referred to as the "S-Boxes." The S-Boxes are discussed in detail in section 2.2.3. In brief: the S-Boxes are special non-linear functions used in DES, which take six bits of input and return four bits of output. S-Box calls make up the majority of the 32-bit DES implementation, and thus most of the time the 32-bit processor registers only have four to six bits of data in them.

algorithms, including Serpent by Ross Anderson and Eli Biham[21]. This paper discusses Bitslice in great detail in Section 2.3.

2.2 Understanding DES

The following is a more in-depth, although still slightly abbreviated, explanation of the innards of the DES algorithm. Those interested in the full specification with a more detailed discussion should consult[11, 12, 24, 16, 31].

I will discuss DES in five parts. The first part deals with the sub-key²⁸ generation, the second gives a high-level perspective of the actual encryption of each data-block, the third details the crucial f function and its S-Box components, the fourth describes decryption and the implementation of the various modes of DES, and finally the fifth section covers 3DES – the most popular form of DES in existence today (and the only variation of DES still sanctioned by the US government). All of this information will be crucial for understanding how Bitslice DES is constructed and for a good understanding of the source code which I have provided.

The DES algorithm begins with the user supplying a 64-bit key and a stream of plain-text of arbitrary length. To begin processing this stream of plain-text, it is first divided into 64-bit blocks, each of which will each be encrypted separately. If the length of the plain-text is not exactly a multiple of 64 bits,²⁹ the final block of data is padded accordingly. After padding and division into 64-bit blocks, the algorithm continues with sub-key generation.

2.2.1 Sub-Key Generation

DES is performed in 16 rounds. Each of these 16 rounds requires a different “sub-key” (a smaller key built from a subset of the original key-text). Sub-key generation is the process

²⁸Sub-keys are sub-sections of the original key-text used in the internals of the DES algorithm. The details of sub-keys will be discussed at great length in Section 2.2.1.

²⁹Actually, because DES appends some final information to the end of the cipher-text the plain-text is padded to slightly less than an exact multiple of 64. For detailed discussion of DES padding, please consult Schneier[24].

of taking the 64-bit key-text and creating 16 48-bit sub-keys used for the 16 rounds of DES. These sub-keys are actually generated using only 56 of the 64 bits of the key, skipping every 8th bit. Due to this fact, often cryptographers speak of DES as providing only “56-bits of security,” as only 56-bits affect the security of the encrypted data.

To begin generation of the sub-keys a permutation vector $PC-1$ is first applied to the original key, which we will call K . This permutation when applied forms a permuted vector containing only 56 of the original 64 bits of the key-text K . We will call this permuted, smaller key $K+$.³⁰ Below is the $PC-1$ permutation table. Each entry in the table corresponds to the bit number from the original key (e.g. the first bit of $K+$ is actually the 57th bit of the original key and the eighth bit is actually the first bit of the original key). Bits are numbered in these examples from left to right, starting at one. For convenience I have also treated the bits memory for my Bitslice implementation in a left to right fashion which you will see in later sections and when browsing the source code.³¹ The vectors shown in this section are to be treated as if they were $1 \times n$ read left to right, reading across first and then down. (They are displayed in a more “square” fashion for easy reading.) For example, if

$$K = 0x133457799BBCDF1$$

$$K = 00010011\ 00110100\ 01010111\ 01111001\ 10011011\ 10111100\ 11011111\ 11110001$$

applying $PC-1$

$$PC-1 = \begin{pmatrix} 57 & 49 & 41 & 33 & 25 & 17 & 9 \\ 1 & 58 & 50 & 42 & 34 & 26 & 18 \\ 10 & 2 & 59 & 51 & 43 & 35 & 27 \\ 19 & 11 & 3 & 60 & 52 & 44 & 36 \\ 63 & 55 & 47 & 39 & 31 & 23 & 15 \\ 7 & 62 & 54 & 46 & 38 & 30 & 22 \\ 14 & 6 & 61 & 53 & 45 & 37 & 29 \\ 21 & 13 & 5 & 28 & 20 & 12 & 4 \end{pmatrix}$$

³⁰The following section is based largely the discussion of DES provided by Grabbe[12].

³¹The discussion of the basis for this choice, its effect on the code and its relation to common practice are discussed in Appendix A.2.

will form

$$K+ = 1111000\ 0110011\ 0010101\ 0101111\ 0101010\ 1011001\ 1001111\ 0001111$$

In the next step, $K+$ is split into two halves, a left C_0 , and right D_0 . It is from these C_0 , D_0 half-key pairs that we generate each of the 16 sub-keys. The half-key pairs C_n, D_n for $n = 1 \dots 16$ are generated by applying successive left rotations to the previous C_{n-1}, D_{n-1} pairs. Continuing our example from above, we have:

$$C_0 = 1111000\ 0110011\ 0010101\ 0101111, D_0 = 0101010\ 1011001\ 1001111\ 0001111$$

The number of single-bit left rotations applied to each C_n, D_n pair is given by Left-Rotate below. Left-Rotate, like all other named vectors ($PC-1$, $PC-2$, Left-Rotate, E , IP , IP^{-1}) given in this description is set in stone by the DES specification[11].³²

$$\text{Left-Rotate} = \{0, 1, 1, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 1\}$$

The pair C_n, D_n are to be rotated Left-Rotate[n] places to the left from the bit positions in C_{n-1}, D_{n-1} (e.g. for $n = 0$, we rotate 0, for $n = 1$ we rotate one from the 0 starting position, and for $n = 4$ we rotate two from the C_3, D_3 pair – a total of four positions from the C_0, D_0 pair). Applying successive rotations yields:

$$C_0 = 1111000\ 0110011\ 0010101\ 0101111, D_0 = 0101010\ 1011001\ 1001111\ 0001111$$

$$C_1 = 1110000\ 1100110\ 0101010\ 1011111, D_1 = 1010101\ 0110011\ 0011110\ 0011110$$

³²Left-Rotate is the only vector I list here which is 0-indexed, for all other vectors the indexing in general doesn't matter and can be assumed to begin with 1 or 0 as you please (in the source code by the design of C/Perl I am expected to use 0). A vector being 0-indexed means that the first value in the vector V is stored at the address 0, such that $V[0]$ returns the first value in the vector and $V[1]$ returns the second value in the vector.

$$C_2 = 1100001\ 1001100\ 1010101\ 0111111, D_2 = 0101010\ 1100110\ 0111100\ 0111101$$

⋮

$$C_{16} = 1111000\ 0110011\ 0010101\ 0101111, D_{16} = 0101010\ 1011001\ 1001111\ 0001111$$

The final shifted half-key pairs are then concatenated together to form 16 56-bit pre-keys, which we will call PK_n .

$$PK_1 = 11100001\ 10011001\ 01010101\ 11111010\ 10101100\ 11001111\ 00011110$$

$$PK_2 = 11000011\ 00110010\ 10101011\ 11110101\ 01011001\ 10011110\ 00111101$$

⋮

$$PK_{16} = 11110000\ 11001100\ 10101010\ 11110101\ 01010110\ 01100111\ 10001111$$

The final sub-keys are formed from 16 pre-keys ($PK_{1...16}$) using a final permutation vector $PC-2$, which selects only 48 bits from each of these 56-bit pre-keys.

$$PC-2 = \left\{ \begin{array}{cccccc} 14 & 17 & 11 & 24 & 1 & 5 \\ 3 & 28 & 15 & 6 & 21 & 10 \\ 23 & 19 & 12 & 4 & 26 & 8 \\ 16 & 7 & 27 & 20 & 13 & 2 \\ 41 & 52 & 31 & 37 & 47 & 55 \\ 30 & 40 & 51 & 45 & 33 & 48 \\ 44 & 49 & 39 & 56 & 34 & 53 \\ 46 & 42 & 50 & 36 & 29 & 32 \end{array} \right\}$$

Applying $PC-2$ to our pre-keys PK_n yields:

$$K_1 = 00011011\ 00000010\ 11101111\ 11111100\ 01110000\ 01110010$$

$$K_2 = 01111001\ 10101110\ 11011001\ 11011011\ 11001001\ 11100101$$

⋮

$$K_{16} = 11001011\ 00111101\ 10001011\ 00001110\ 00010111\ 11110101$$

We now have our 16 48-bit sub-keys which we will use below in our discussion of the actual data encryption.

2.2.2 Data Encryption

DES encryption begins with the 64-bit plain-text block (M). Like the first step in sub-key generation, data encryption begins by applying a permutation to the plain-text. The 64-bit initial permutation IP is applied to the plain-text M to form $M+$. Unlike the permutation $PC-1$ used for sub-key generation, IP is a full 64-bit permutation, thus no data are lost when permuted. For example, if

$$M = 0x123456789ABCDEF$$

$$M = 00000001\ 00100011\ 01000101\ 01100111\ 10001001\ 10101011\ 11001101\ 11101111$$

Applying IP

$$IP = \left(\begin{array}{cccccccc} 58 & 50 & 42 & 34 & 26 & 18 & 10 & 2 \\ 60 & 52 & 44 & 36 & 28 & 20 & 12 & 4 \\ 62 & 54 & 46 & 38 & 30 & 22 & 14 & 6 \\ 64 & 56 & 48 & 40 & 32 & 24 & 16 & 8 \\ 57 & 49 & 41 & 33 & 25 & 17 & 9 & 1 \\ 59 & 51 & 43 & 35 & 27 & 19 & 11 & 3 \\ 61 & 53 & 45 & 37 & 29 & 21 & 13 & 5 \\ 63 & 55 & 47 & 39 & 31 & 23 & 15 & 7 \end{array} \right)$$

to M yields:

$$M+ = 11001100\ 00000000\ 11001100\ 11111111\ 11110000\ 10101010\ 11110000\ 10101010$$

As in key generation, we take this permuted block and divide it into two (this time 32-bit) halves which we will call L_0 and R_0 . This step of block division is the first step in

any Feistel network (as discussed above in Section 2.1.1). We now have the two initial 32-bit half-blocks:

$$L_0 = 11001100\ 00000000\ 11001100\ 11111111, R_0 = 11110000\ 10101010\ 11110000\ 10101010$$

With all the setup complete, DES encryption consists simply of 16 applications of the following (standard Feistel network) formula:

$$\begin{aligned} L_n &= R_{n-1} \\ R_n &= L_{n-1} \oplus f(R_{n-1}, K_n) \end{aligned}$$

Notice how after each round the left block and right blocks are swapped, and just like any Feistel network, after the non-linear function f is applied to one half, that half is XORed with the other half.

After 16 iterations of this formula, we result in a final L_{16}, R_{16} . To form the final encrypted cipher-text block, we begin by concatenating the two halves *in reverse order* to form a pre-cipher-text block which I will call $C+$.

$$L_{16} = 01000011\ 01000010\ 00110010\ 00110100, R_{16} = 00001010\ 01001100\ 11011001\ 10010101$$

$$C+ = R_{16}L_{16}$$

$$C+ = 00001010\ 01001100\ 11011001\ 10010101\ 01000011\ 01000010\ 00110010\ 00110100$$

The final cipher-text C is formed from $C+$ by applying the inverse IP vector IP^{-1} :

$$IP^{-1} = \left\{ \begin{array}{cccccccc} 40 & 8 & 48 & 16 & 56 & 24 & 64 & 32 \\ 39 & 7 & 47 & 15 & 55 & 23 & 63 & 31 \\ 38 & 6 & 46 & 14 & 54 & 22 & 62 & 30 \\ 37 & 5 & 45 & 13 & 53 & 21 & 61 & 29 \\ 36 & 4 & 44 & 12 & 52 & 20 & 60 & 28 \\ 35 & 3 & 43 & 11 & 51 & 19 & 59 & 27 \\ 34 & 2 & 42 & 10 & 50 & 18 & 58 & 26 \\ 33 & 1 & 41 & 9 & 49 & 17 & 57 & 25 \end{array} \right\}$$

Giving our final cipher-text:

$$C = 10000101 \ 11101000 \ 00010011 \ 01010100 \ 00001111 \ 00001010 \ 10110100 \ 00000101$$

$$C = 0x85E813560F0AB405$$

The next section will cover the details of the 16 applications of the DES encryption formula (standard Feistel network formula) mentioned above.

2.2.3 The f Function and S-Boxes

The final piece missing from my explanation here is a description of the non-linear function f and the application of the DES encryption formula mentioned above. Like the rest of DES, the f function is specified in detail by the official NIST specification[11]. The f function is rather complex and will be broken down into several steps. I will first list here a brief overview of the individual steps of f . Also included is a similar pictorial explanation in Figure 1. Finally I provide a detailed example usage of f with the same sample data from the previous sections.

Application of $f(R_{n-1}, K_n)$ begins with the expansion (and permutation) of the incoming R_{n-1} block through the use of the expansion vector E . The resulting expanded block is then XORed with the provided round key K_n . This resulting block ($E(R_{n-1}) \oplus K_n$) is broken into eight sub-blocks ($B_{1...8}$). These sub-blocks are in turn fed into eight separate non-linear functions called S-Boxes ($S_{1...8}$). The result of those eight functions is re-combined to form a 32-bit block. This 32-bit block is then permuted (by P) and returned by f . A pictorial

overview of f is provided below in Figure 1, a detailed explanation of the f function follows.

$$f(R_{n-1}, K_n)$$

R_{n-1} is expanded:

$$R_{n-1} \rightarrow E(R_{n-1})$$

The expanded block $E(R_{n-1})$ is broken into eight smaller blocks:

$$E(R_{n-1}) \oplus K_n = (B_1)(B_2)(B_3)(B_4)(B_5)(B_6)(B_7)(B_8)$$

An S-Box is applied to each smaller block:

$$(B_1)(B_2)(B_3)(B_4)(B_5)(B_6)(B_7)(B_8) \rightarrow S_1(B_1)S_2(B_2)S_3(B_3)S_4(B_4)S_5(B_5)S_6(B_6)S_7(B_7)S_8(B_8)$$

The results from the S-Boxes are concatenated and permuted with P :

$$P(S_1(B_1)S_2(B_2)S_3(B_3)S_4(B_4)S_5(B_5)S_6(B_6)S_7(B_7)S_8(B_8)) = f(R_{n-1}, K_n)$$

Figure 1: f Function Overview

The f function begins by applying the expansion vector E to the 32-bit half block R_{n-1} to form $E(R_{n-1})$.

$$E = \begin{pmatrix} 32 & 1 & 2 & 3 & 4 & 5 \\ 4 & 5 & 6 & 7 & 8 & 9 \\ 8 & 9 & 10 & 11 & 12 & 13 \\ 12 & 13 & 14 & 15 & 16 & 17 \\ 16 & 17 & 18 & 19 & 20 & 21 \\ 20 & 21 & 22 & 23 & 24 & 25 \\ 24 & 25 & 26 & 27 & 28 & 29 \\ 28 & 29 & 30 & 31 & 32 & 1 \end{pmatrix}$$

This expanded block is then XORed with the provided sub-key to yield a 48-bit block $K_n \oplus E(R_{n-1})$. I will take for example $n = 1$, and compute R_1, L_1 using data from the previous section. We first expand R_0 :

$$R_0 = 11110000 \ 10101010 \ 11110000 \ 10101010$$

$$E(R_0) = 01111010 \ 00010101 \ 01010101 \ 01111010 \ 00010101 \ 01010101$$

Next we XOR the expanded block ($E(R_0)$) with the provided key-text K_1 :

$$\begin{aligned} K_1 &= 00011011\ 00000010\ 11101111\ 11111100\ 01110000\ 01110010 \\ K_1 \oplus E(R_0) &= 01100001\ 00010111\ 10111010\ 10000110\ 01100101\ 00100111 \end{aligned}$$

Now we break $K_1 \oplus E(R_0)$ into 8 6-bit sub-blocks which we will call $B_1 \dots B_8$.

$$\begin{aligned} K_1 \oplus E(R_0) &= (B_1)(B_2)(B_3)(B_4)(B_5)(B_6)(B_7)(B_8) \\ &= 011000\ 010001\ 011110\ 111010\ 100001\ 100110\ 010100\ 100111 \end{aligned}$$

Each of these 8-bit sub-blocks is then fed into one of eight S-Boxes. Before I continue with my example it is worth saying a few words about the S-Boxes.

S-Box stands for substitution-box, and the eight S-Boxes together form the heart of DES. Each S-Box is a non-linear mapping which takes six bits of input data and maps them to four bits of output data. In standard DES implementations S-Boxes are implemented as lookup tables, where two of the six bits determine the row, and four of the six bits determine the column for the lookup. S-Box #1 is shown in Figure 2 in its lookup table form. I have not included the rest of the S-Boxes here, but those interested can review their contents in numerous places including J. Orlin Grabbe's article[12] and the official DES specification[11]. I should also note here that S-Boxes can also be constructed with alternative table dimensions than the standard 4×16 or even without the use of tables (as they are in hardware implementations and for Bitslice DES). Appendix C.13 lists Matthew Kwan's reduced-gate-count logic-gate S-boxes, as were used in my Bitslice DES implementation. More in-depth discussions of other S-Box variations, as well as the specific mathematical properties of the S-Boxes are available from other sources including Schneier[24] and Menezes[16].

Returning to our example, we now apply the eight S-boxes to our example data. This

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7
1	0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8
2	4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0
3	15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13

Figure 2: S-Box #1 as a lookup table

application yields:

$$S_1(B_1)S_2(B_2)\dots S_7(B_7)S_8(B_8) = 0101\ 1100\ 1000\ 0010\ 1011\ 0101\ 1001\ 0111$$

Taking the concatenated results from these S-Box applications, for the final step in the f function we apply of the permutation vector P .

$$P = \begin{pmatrix} 16 & 7 & 20 & 21 \\ 29 & 12 & 28 & 17 \\ 1 & 15 & 23 & 26 \\ 5 & 18 & 31 & 10 \\ 2 & 8 & 24 & 14 \\ 32 & 27 & 3 & 9 \\ 19 & 13 & 30 & 6 \\ 22 & 11 & 4 & 25 \end{pmatrix}$$

$$\begin{aligned} P(S_1(B_1)\dots S_8(B_8)) &= 0010\ 0011\ 0100\ 1010\ 1010\ 1001\ 1011\ 1011 \\ &= f(R_{n-1}, K_n) \end{aligned}$$

This completes the discussion of the innards of the DES encryption algorithm. The next section offers decryption and mode information necessary for practical application of the algorithm.

2.2.4 DES Modes & Decryption

As alluded to at the beginning of this section, DES and other block ciphers have several different modes of operation. Generally the ability to answer the problem of applying parallelism depends directly on the mode in which one uses the cipher, thus the discussion of these modes has direct bearing on my study here. The three most commonly used block-cipher modes are:

1. ECB (Electric Code Book) - In ECB mode each block of the message is encrypted separately. This is the most common block cipher mode, but is less secure than any of the others described here. ECB is vulnerable to attacks under which plain-texts, or partial plain-texts (and their associated cipher-texts) are known[4]. For modern algorithms with large (128-bit or larger) block sizes, the number of known plain-texts required for an attack is extremely large ($> 2^{43}$ plain-texts for DES).³³ Regardless, it is a good idea when using block ciphers in this mode to change keys often (at least every $n/2$ blocks where n is the smallest number of required plain-texts for a known attack). This mode allows very easy application of parallelism to a block cipher implementation as you will see in my discussion of Bitslice DES below.
2. CBC (Cipher Block Chaining) - In CBC mode each block is encrypted after first XORing the plain-text of this message block (block_n) with the cipher-text from the previous block (block_{n-1}). This introduces block-to-block data dependency and assures that two identical cipher blocks have no relation in their plain-texts. Single packet (or single file), block-level parallelism is impossible when performing encryption in this mode.³⁴ I should note here that although it is impossible to apply block-level

³³One of a number of sources discussing known plain-text attacks on small block-size ciphers such as DES is RSA's own website: <http://www.rsasecurity.com/rsalabs/faq/3-2-2.html> Schneier also offers information on the subject of plain-text attacks[24].

³⁴An example of block-level parallelism would be reading four blocks from a file at once and then encrypting them all in parallel. This is different from conventional non-parallel implementations such as libdes, which may read multiple blocks at once, but still encrypt them all sequentially instead of in parallel. This block-level parallelism is impossible in CBC mode due to the block-to-block data dependence inherent in CBC mode encryption.

parallelism to ciphers while encrypting in CBC mode, this limitation is *not* present during decryption. Since CBC mode functions by XORing the block_{*n*-1} cipher-text with the block_{*n*} plain-text *before* encryption, decrypting any CBC block_{*n*} will yield the XOR product of the original block_{*n*} plain-text and the block_{*n*-1} cipher-text. One could decrypt all CBC cipher-text blocks in parallel and XOR them with the appropriate cipher-text blocks as needed. Since all encrypted blocks are necessarily known at decryption time, all blocks of the message can be decrypted simultaneously.³⁵ This allows full use of block level parallelism when running decryption under CBC mode.

3. CFB (Cipher FeedBack) - In CFB mode each block of cipher-text is computed by first encrypting the previous block's cipher-text (again) and then XORing at least part of that result (re-encrypted cipher-text) with a sub-block of this round's plain-text. CFB mode can be used with plain-text sub-blocks of various lengths ranging from one to the original full block size. Readers interested in understanding the particulars of CFB mode should consult Schneier[24]. For our concerns here, CFB mode also introduces block-to-block data dependency and thus shows similar difficulties to applying parallelism to ciphers used in CBC mode.

Having now discussed the various block-cipher modes, it is also important in this section to discuss the particulars of DES decryption. DES decryption is nearly identical to DES encryption, but its small differences from encryption are useful to review here both for better understanding of the attached project source, and for the benefit anyone wishing to implement their own Bitslice DES.

Decryption in DES is relatively simple due to the circular nature of both the XOR operation and the f function.³⁶ To implement DES decryption, a programmer need only apply DES as normal to the block of cipher-text but change the order in which she applies the

³⁵This is unlike encryption where the previous cipher-text for each block is *not* yet known. Each cipher-text must be computed sequentially in CBC encryption.

³⁶This means that if you apply $f(f(x)) = x$, also $((x \oplus a) \oplus a) = x$.

sub-keys. For decryption one generates the normal sub-keys, but reverses the key schedule³⁷ (e.g. K_1 now becomes K_{16} and K_{16} now becomes K_1 , etc.). When examining the perl-script listed in Appendix C.1 used for DES decryption code generation, you will see that I have done exactly that. To allow for best understanding of DES decryption, I give a step-by-step explanation below.³⁸

The first step in DES encryption/decryption is to apply the initial permutation IP . When applying IP to a cipher-text block, this cancels the previous application of IP^{-1} (the final stage of encryption) leaving us with the concatenated pair (R_{16}, L_{16}) . For decryption, we will call these (L_0, R_0) respectively. Now consider the DES encryption formula:

$$\begin{aligned} L_n &= R_{n-1} \\ R_n &= L_{n-1} \oplus f(R_{n-1}, K_n) \end{aligned}$$

Applying this in the first round encryption context of $n = 1$ is effectively,

$$\begin{aligned} L_1 &= R_0 \\ R_1 &= L_0 \oplus f(R_0, K_1) \end{aligned}$$

but in terms of decryption we really have: (where (R_{16}, L_{16}) are the half-blocks as they were named during encryption)

$$\begin{aligned} L_1 &= R_0 = L_{16} \\ R_1 &= L_0 \oplus f(R_0, K_1) = R_{16} \oplus f(L_{16}, K_{16}) \end{aligned}$$

³⁷The key schedule mentioned here refers to the specified order in which the sub-keys are applied. The phrase “key scheduling” is often used as a synonym for “generating sub-keys” when discussing block cipher implementations.

³⁸The decryption example which I describe here, draws from the discussion found in Menezes[16].

If we remember from encryption (or simply consult the DES encryption formula above), we can make substitutions for $L_{16} = R_{15}$ and $R_{16} = L_{15} \oplus f(R_{15}, K_{16})$. Rewriting:

$$\begin{aligned} L_1 &= R_0 &= L_{16} &= R_{15} \\ R_1 &= L_0 \oplus f(R_0, K_1) &= R_{16} \oplus f(L_{16}, K_{16}) &= L_{15} \oplus f(R_{15}, K_{16}) \oplus f(R_{15}, K_{16}) \end{aligned}$$

Noting the circular property of XOR ($((x \oplus a) \oplus a) = x$), we simplify:

$$\begin{aligned} L_1 &= R_{15} \\ R_1 &= L_{15} \end{aligned}$$

Thus with a little logical deduction, we have shown that decryption round one, yields $(L_1, R_1) = (R_{15}, L_{15})$, inverting round 16 of encryption. Repeating this over 15 more rounds yields $(L_{16}, R_{16}) = (R_0, L_0)$. The final steps of decryption are the same as encryption. First we concatenate the two halves (L_{16}, R_{16}) *in reverse order* to form the block $R_{16}L_{16} = L_0R_0$. We then apply the final IP^{-1} to this concatenated block. The application of IP^{-1} cancels the original IP (as was applied to the plain-text in the first step of encryption) and results in the original plain-text. The interested reader, can apply all 16 rounds by hand for further proof, or alternatively run my Bitslice implementation with the “-P” or “-T” flags (see Section 2.5) to confirm the correctness of my decryption.

2.2.5 3DES

3DES (pronounced “triple-dez” or “three-dez”) is the application of the DES cipher three times over a each message block, using two (or three) different keys[11]. I mention 3DES because it is by far the most common form of DES in use today. DES is no longer considered secure for general use by the federal government as the short 56-bit DES keys can be discovered (via brute force computation) in a matter of hours using powerful enough computers. 3DES was developed as a DES replacement, and, although it has now been superseded by

the new AES, it is still regarded as secure and is in widespread use. 3DES encryption is accomplished by chaining DES encryption-decryption-encryption together,³⁹ in any of the modes mentioned above.⁴¹ The 3DES variant on DES effectively triples the number of rounds of the DES algorithm, and doubles (or triples) the secret key length. Just like DES, when 3DES is used in ECB mode it is easily parallelized by distributing packets among various processors, or over a vector and using a VPU. The block-level parallelism possible in ECB mode can be exploited well with a Bitslice DES implementation.

3DES used in CBC or CFB modes does not allow direct block-level pipelining⁴² due to the block-to-block data dependence introduced by CBC and CFB modes during encryption. One can, however, still get a speed boost in 3DES CBC mode by decrypting all blocks in parallel using the decryption trick mentioned above for CBC mode. I currently know of no library which exploits this decryption trick when using a parallel 3DES implementation (such as Bitslice 3DES).

2.3 Understanding Bitslice-DES

Having now covered the basic DES algorithm, we can speak more in depth about an optimized version of DES called Bitslice DES. Bitslice DES is a faster DES implementation originally proposed by Eli Biham in the 1997 presentation of his paper “A fast new DES implementation in software”[4]. The name “Bitslice” was coined by Matthew Kwan shortly following Biham’s presentation and has been used since to describe this implementation[29]. Bitslice DES has

³⁹Chaining here refers to how the output of encryption is fed directly into decryption.⁴⁰ The decryption is performed with a different key from the first original encryption, thus the message is not returned to plain-text, but rather scrambled further. When DES is performed with two keys as opposed to three, the encryption (first and third) operations share the same key, while the decryption (second) operation uses a separate key.

⁴¹3DES decryption is accomplished by chaining decryption-encryption-decryption together using the same two or three keys used for 3DES encryption. See Schneier[24], Menezes[16] or Welschenbach[28] for further discussion of 3DES and the details of its implementation.

⁴²Pipelining is when in output from one function/process is fed directly into another function/process. This is a technique for exploiting parallelism whereby one process will compute stage one of the algorithm for block one, feed that directly into a second processor which will compute stage two for block one while the first processor computes stage one of block two, etc. Such pipelining is not possible in 3DES used in CBC or CFB mode due to the block-to-block data dependence introduced by those modes.

since that presentation attained rather limited fame, being used primarily for key-searching during the RSA DES challenge⁴³ and password cracking programs such as John the Ripper.⁴⁴ What I discuss in this paper is a modern version of the Bitslice DES algorithm, one optimized for processors with Vector Processing Units (particularly the AltiVec) and capable not only of key-searching but also key encryption and decryption.

Bitslice gains its speed by solving the problem of DES’s inefficient register usage. As mentioned above, during the majority of its execution a plain-vanilla DES implementation uses only four to six bits of any register – a highly inefficient practice on modern 32-bit or larger processors. Bitslice in contrast will use every bit it is provided and scales from a 1-bit processor on up to as many bits as we may some day dream of. Bitslice accomplishes this efficiency by changing the way in which we store the data in these registers.

Normal DES implementations work on a single block of data at a time, and within that block work on four to six bits at any given time. Bitslice in contrast will work on n blocks of data at a time, where n is the bit-width of the registers of the processor on which it is implemented. Bitslice transforms the “heterogeneous” data blocks⁴⁵ consisting of some four- or six-bit subset of the 32-bit half-block,⁴⁶ into “homogeneous” data blocks consisting of 32 first-bits (or second- or third-bits) from 32 different data blocks[10]. Figure 3 shows a comparison between normal DES register usage and Bitslice DES register usage.⁴⁷ Where normal DES would operate on four bits of a single block, Bitslice DES operates on four registers full of 32 copies of those same four bits from 32 different blocks. DES regards each n bit processor available to the system as an $n \times 1$ -bit SIMD processor (capable of performing

⁴³<http://www.rsasecurity.com/rsalabs/challenges/des3/>

An implementation of Bitslice was actually used in the cracking program used by the winning team.

⁴⁴<http://www.openwall.com/john/>

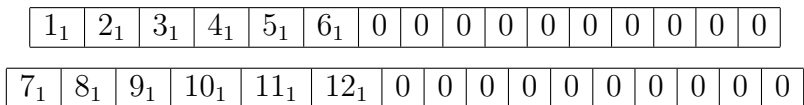
⁴⁵This is done via a process called “swizzling” which is discussed in great detail in Sections 2.3.2 and 2.4.1.

⁴⁶Commonly bits are referred to as 0 through 31, and all arrays (in common programming languages) are 0-based, i.e. the first value is stored at the index 0. For clarity to all readers however, (including those not from a computer science background) I have chosen to use 1-based arrays and begin counting bits starting with one.

⁴⁷ n_m refers to bit n from block m . The normal DES registers are two registers used to hold 6-bit S-Box inputs from a single block. The Bitslice DES registers are the six registers needed to hold the 32 copies of six S-Box input bits from 32 blocks.

simple logic calculations on each bit) upon which it performs the hardware implementations of DES. A Bitslice implementation can efficiently compute up to x blocks in parallel on an x -bit processor[4]. This implementation turns out to be significantly faster than normal DES (despite some hidden costs we will discuss below).

Normal DES 16-bit Registers



Bitslice DES 16-bit Registers

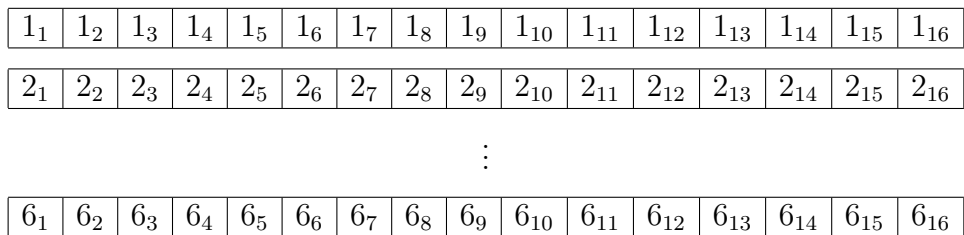


Figure 3: Register Usage: DES vs. Bitslice DES

2.3.1 The Difference of Hardware DES

Bitslice DES functions with the principle of using the hardware version of DES in software. Hardware implementations of DES have several subtle differences from software implementations, and it is from those differences that we both gain and lose efficiency with Bitslice. Those differences and how they affect Bitslice DES are discussed below.

One gain we receive in hardware is that the permutation operations used throughout DES are completely free in hardware. The electrons leaving one logic gate can be routed into any other at uniform cost, achieving permutation of the data at zero cost. The permutation matrices dictate at circuit design time where to connect each wire to. In a similar fashion when implementing Bitslice DES all permutation decisions are made at source code generation time, saving the implementation from executing permutation computations at runtime.

Another change to DES, when implementing the algorithm in hardware, is the S-Boxes. Because hardware is expensive, the large lookup-table-based S-Boxes commonly used in software DES are replaced by equivalent logic-gate S-Box implementations in hardware DES. These logic-gate S-Boxes are both more complex to understand and more complex to design than simple lookup tables. However, even extremely inefficient logic-gate S-Boxes save substantial circuit board space over lookup-table S-Boxes in hardware. The efficient design of various logic gate implementations are outlined in papers both from Biham[4] and Kwan[13] and will not be discussed here. The question of how to design the most efficient logic gate S-Boxes is still open.

Logic gate implementations in hardware can be implemented as multi-input, multi-output gates. Using several logic gates chained together one can replace an S-Box lookup-table. For logic-gate S-Boxes to be useful for Bitslice, however, we require exclusively two-input, single-output gates. This limitation is because in software we only have two-input one-output boolean logic operations (the simple logic operations described in Appendix A.3 – AND (&), OR (|), XOR (\oplus), ANDC, NOR, NAND). The specific design and two-input conversion of these gates is outside the scope of this paper. Those interested can again consult Kwan[13] and Biham[4] for various gate generation algorithms. For my Bitslice implementation I have used slightly modified versions of Kwan’s generated S-Boxes which he offers at his website[29] in source form.

2.3.2 Bitslice Implementation Changes

So what do these changes mean? For one, the change from addressing heterogeneous data to homogenous data means that we have to somehow transform the heterogeneous data which we receive 99% of the time, into the homogeneous data which we need.⁴⁸ This is done via a complex process called swizzling. Swizzling is necessary in order to change the data that

⁴⁸The use of the words heterogeneous data and homogenous data were explained in Section 2.3.

we receive in from the rest of the world to a format which Bitslice can process efficiently.⁴⁹ The swizzling process is the most expensive part of any current Bitslice implementation. Swizzling requires changing the orientation of all data in the desired section of memory; this is not a trivial operation. Figure 4 shows the effect of swizzling in eight 8-bit blocks.⁵⁰ The swizzling we use throughout Bitslice is of 32-, 64-, or 128-bit blocks on a 32-bit processor (or 128-bit VPU).⁵¹

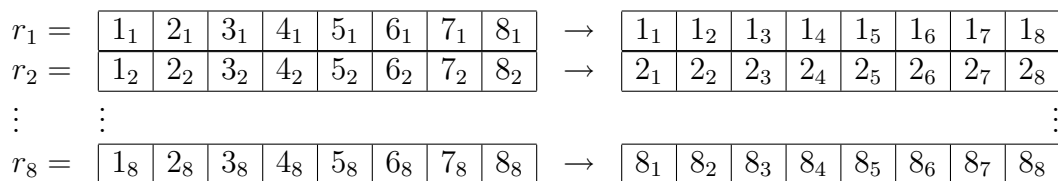


Figure 4: Swizzling eight 8-bit blocks on 8-bit registers

With the data swizzled into homogenous register groupings, we can now modify our code (make it Bitslice DES instead of normal DES) to operate on these vectors instead of the individual bits as it had before.

2.4 The AltiVec Vector Processing Unit (VPU)

Important to understanding my implementation of Bitslice is some understanding of the hardware on which it was implemented. Part of what allows my implementation to perform as well as it does is the architecture on which it is designed, specifically the vector processing

⁴⁹Swizzling can essentially be thought of as a bit-level matrix transpose. The swizzling algorithm is given a group of n blocks of k bits, and is expected to return k blocks of n bits. There are two problems which make this simple sounding problem complex. The first is that computers don't organize bits in nice arrays in memory. Everything is stored in long continuous streams. We can't then just say to a computer, "I want to look at that square of memory, just read it to me down, first then across, instead of across first then down." There is no concept of "down" in memory – only across. The second is that computers work with byte-addressing, and we are performing bit level operations. So we can't just ask for the first bit, we have to take byte chunks at a time, and treat each bit within those bytes differently. Byte addressing is explained more in Appendix A.2.

⁵⁰Notice I have numbered the bits on this processor in reverse of what is "common." I have done this throughout my source code as well, and made this decision for two reasons. The first reason is that this is the numbering used in the DES description which I used most heavily[12]. The second reason is that I felt this numbering system left to right, would appeal as more logical to the reader as we are not treating these individual bits with any numerical meaning.

⁵¹Again here, as in previous figures, I use n_m to signify the n th bit from the m th block.

unit which it so heavily uses. The vector processing unit featured in my implementation is the Motorola AltiVec™ Vector Processing Unit. The AltiVec was designed particularly for multimedia and scientific applications in which large sets of data undergo similar transformations at the same time. AltiVec instructions achieve as much as a $4\times$ speedup over integer unit instructions by executing the same instruction on a block of data four times as wide.⁵²

For my implementation I focused on three aspects of the AltiVec: bitwise logical operators, permute operations, and data stream operations. In this section I describe each type of operation, list the common operations I used, and provide diagrams to explain the actual memory manipulations each operation performs.

To begin my discussion of AltiVec instructions, I take the simplest instructions: boolean logic instructions. The AltiVec architecture includes a total of 160 new instructions for vector processing[2]. Five of those instructions are bitwise boolean logic operations and are listed in Table 2 by their C language names. I used these boolean logic instructions throughout the AltiVec versions of my code to replace the corresponding C language built-in boolean operators (and (&), or (|) and xor (^) and not (!)⁵³). For those not familiar with Boolean logic, a brief overview is given in Appendix A.3. The functions listed in Table 2 are used extensively in my AltiVec translation of Kwan’s S-Boxes. `vec_xor` in particular is used commonly throughout my generated Bitslice encrypt/decrypt code. All of the instructions listed in Table 2 expect two 128-bit input vectors and return a 128-bit result.

One of the AltiVec’s most useful features – the one which has made my efficient swizzling algorithm possible – is the AltiVec’s suite of permute operations. These include operations to reorder bytes within a vector, shift bits within a vector and build new vectors from other

⁵²The majority of the information in this section comes from (partial) reading of both the AltiVec Technology Programming Interface Manual[2] and AltiVec Technology Programming Environment Manual[3] supplied by Motorola. Additional information, especially related to proper usage of data stream instructions was found in Ollmann’s AltiVec tutorial[19]. Readers interested in learning more about the AltiVec processing unit are encouraged to consult those three technical papers as well as Apples Developer documentation: <http://developer.apple.com/hardware/ve/>

⁵³The NOT operator is not covered in Appendix A.3 as it is not otherwise used throughout this paper. Any NOT operator can equivalently be rewritten as an XOR operator of a value with itself. Otherwise written: $\text{NOT } a = a \text{ XOR } a$.

`vec_and` takes two vectors and returns their 128-bit boolean AND
`vec_or` takes two vectors and returns their 128-bit boolean OR
`vec_xor` takes two vectors and returns their 128-bit boolean XOR
`vec_nor` takes two vectors and returns the complement of their 128-bit boolean OR
`vec_andc` takes two vectors and returns the 128-bit boolean AND of the first vector with the complement of the second vector.

Table 2: AltiVec Boolean Instructions

vectors. All of the AltiVec permute operations used in my code are listed in Table 3.

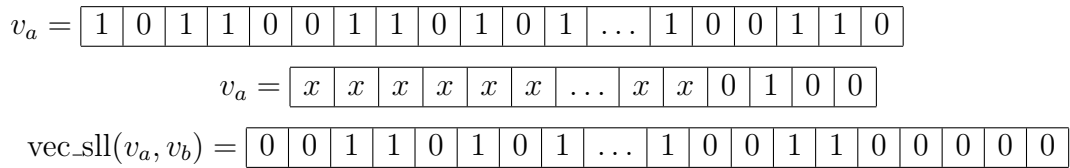


Figure 5: `vec_sll` Instruction Diagram

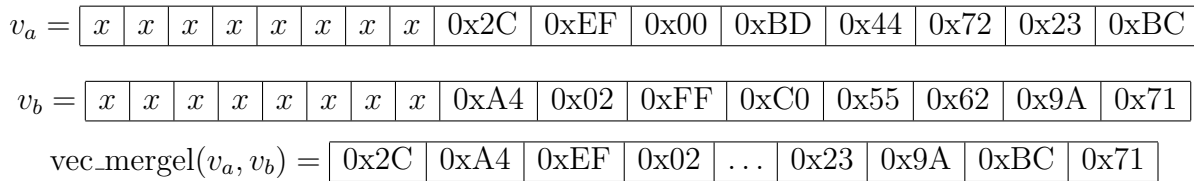


Figure 6: `vec_merge1` Instructions Diagram

Most unique of the AltiVec’s permute instructions is the `vec_perm` instruction. This instruction, when used creatively, allows the efficient swizzling demonstrated in my implementation. A high-level overview of my AltiVec swizzling algorithm is covered in Section 2.4.1. In this section as an example of the power of these permute operations, I will examine the details of the `interleave128` (or `interleave128c`) function used throughout my AltiVec swizzling code.⁵⁴ Figure 9 contains an abbreviated listing the `interleave128` function: the kernel of the AltiVec swizzling code.

Given two vectors, `interleave128` returns the 256-bit product of a bit-by-bit interleave

⁵⁴A quick scan of my `swizzle_vpu.h` source file reveals that `interleave128c` used throughout my source code is actually only a convenience wrapper around the real `interleave128` function shown in Figure 9 and described in this section.

<code>vec_sll</code>	Vector Shift Left takes two vectors (v_a, v_b). <code>vec_sll</code> shifts the first vector n bits to the left where n is the number specified by the last 4 bits of the second vector. See Figure 5 for an example of <code>vec_sll</code> in use.
<code>vec_mergel</code>	Vector Merge Low bytes takes two vectors (v_a, v_b). From these two vectors <code>vec_merge</code> selects the high or low 64-bit halves and from them forms the byte-wise interlace, storing this in a 128-bit result vector. See Figure 6 for an example of <code>vec_mergel</code> .
<code>vec_sel</code>	Vector Select takes three vectors. The first two vectors passed to <code>vec_sel</code> are data vectors (v_a, v_b), and the third vector is the control vector (v_c). <code>vec_sel</code> uses the control vector to build a result vector. Every bit for which the control vector is 0 the result contains the corresponding bit found in v_a . Every bit for which the control vector is 1 the result contains the corresponding bit found in v_b . Figure 7 shows an example of <code>vec_sel</code> .
<code>vec_perm</code>	Vector Permute takes three vectors. The first two vectors passed <code>vec_perm</code> are data vectors (v_a, v_b), and the third vector is the control vector (v_c). <code>vec_perm</code> regards each of the vectors as 16 groups of 8-bits. <code>vec_perm</code> uses the lower 5 bits of each byte in the control vector to represent a number 0-32 (the highest 3 bits are ignored). The bytes in v_a are regarded by <code>vec_perm</code> as numbered 0-15, and the bytes in v_b as numbered 16-31. <code>vec_perm</code> replaces each byte in the result vector with the corresponding byte from either v_a or v_b based on the lookup using the lower 5-bits of each byte in the control vector. See Figure 8 for an example of this operation.

Table 3: AltiVec Permute Instructions

of the original two vectors. This result split is over two⁵⁵ 128-bit vectors: high and low halves of the larger 256-bit vector.

The algorithm shown in `interleave128` can be broken down into five steps, each of which are performed twice, once to form the high half of the 256-bit vector, and once to form the low half. `interleave128` accomplishes the entire interleave of a full 256-bits in a total of 20 instructions – far fewer than any corresponding code on currently available for an integer unit.

Step 1 of `interleave128` constructs “doubled” copies of one (for this example lower) half of the two original 128-bit vectors. This doubling is accomplished by performing a byte-level

⁵⁵Although `interleave128` allows specifying a separate two vectors into which to place the resulting 256-bit product, the convenience function `interleave128c` returns the result in place of the original vectors.

$$\begin{aligned}
v_a &= \boxed{0\ 0\ 1\ 1\ 0\ 1\ 0\ 0\ 1\ 1\ 0\ 0\ \dots\ 1\ 0\ 0\ 0\ 1\ 0} \\
v_b &= \boxed{1\ 0\ 1\ 0\ 0\ 0\ 0\ 1\ 0\ 0\ 1\ 1\ \dots\ 0\ 0\ 0\ 1\ 0\ 0} \\
v_c &= \boxed{0\ 0\ 1\ 0\ 1\ 1\ 1\ 1\ 0\ 1\ 0\ 1\ \dots\ 1\ 1\ 0\ 1\ 0\ 0} \\
\text{vec_sel}(v_a, v_b, v_c) &= \boxed{0\ 0\ 1\ 1\ 0\ 0\ 0\ 1\ 1\ 0\ 0\ 1\ \dots\ 0\ 0\ 0\ 1\ 1\ 0}
\end{aligned}$$

Vector v_c specifies for each bit whether to place a bit from $v_a(0)$ or $v_b(1)$ in the result.

Figure 7: `vec_sel` Instructions Diagram

$$\begin{aligned}
v_a &= \boxed{2C_{00}\ EF_{01}\ 78_{02}\ FF_{03}\ 35_{04}\ 72_{05}\ 41_{06}\ \dots\ 87_{0A}\ 45_{0B}\ 28_{0C}\ AB_{0D}\ 23_{0E}\ BC_{0F}} \\
v_b &= \boxed{A4_{10}\ 02_{11}\ FF_{12}\ C0_{13}\ 55_{14}\ 62_{15}\ 9A_{16}\ \dots\ 23_{1A}\ C0_{1B}\ 55_{1C}\ 62_{1D}\ 9A_{1E}\ 71_{1F}}
\end{aligned}$$

The control vector v_c specifies which byte from v_a or v_b to place in each byte of the result.

$$\begin{aligned}
v_c &= \boxed{00\ 0F\ 14\ 13\ 13\ 16\ 03\ 1D\ 16\ 04\ 0A\ 1B\ 05\ 10\ 1E\ 1F} \\
\text{vec_perm}(v_a, v_b, v_c) &= \boxed{2C\ BC\ 55\ C0\ C0\ 9A\ FF\ 62\ 9A\ 35\ 87\ C0\ 72\ A4\ 9A\ 71}
\end{aligned}$$

Figure 8: `vec_perm` Instruction Diagram

merge of the vector with itself. This constructs a 128-bit vector consisting of identical two byte pairs, in the order of the original bytes.⁵⁶ Figure 6 shows an example of the `vec_mergel` instruction. Further example data is shown below:

$$\begin{aligned}
v_a &= \boxed{x\ x\ x\ x\ x\ x\ x\ x\ 2C\ EF\ 00\ BD\ 44\ 72\ 23\ BC} \\
\text{vec_mergel}(v_a, v_a) &= \boxed{2C\ 2C\ EF\ EF\ 00\ 00\ \dots\ 72\ 72\ 23\ 23\ BC\ BC}
\end{aligned}$$

Step 2 of `interleave128` calls a four-bit left shift operation with the vector resulting from Step 1 and a special vector ($v30$) of which the last four bits are the binary value representing the number “4.” This left shift operation shifts the entire vector from Step 1 so that each byte (with the exception of the far-right byte) now contains swapped 4 bit pairs consisting of the right four bits of the original byte, followed by the left four bits of the

⁵⁶For example, the first two bytes are both the left-most byte from the lower half of the source vector and the last two bytes of the result are both the right-most byte from the lower half of the source vector.

original byte. Figure 5 shows an example of the `vec_sll` instruction.

Step 3 of `interleave128` uses a vector select operation to build new groupings of these doubled bytes from Steps 1 and 2. This vector select instruction is called with the original vector (with which we began Step 1), the now shifted “doubled” vector result from Step 2, and a special vector (`v31` in the source) for which the bytes alternate `0xFF`, `0x00` (all 1s or all 0s). Vector `v31` is listed as part of Appendix C.5. This instruction constructs a vector consisting of the first byte from the second vector, the second byte from the first vector, etc. Thanks to the shift in Step 2, these resulting bytes are constructed exactly such that the last four bits of each byte are successively four bits from the original vector. We have in essence interleaved one half of the original vector with itself at the 4-bit level. Figure 7 shows an example of the `vec_sel` instruction.

Step 4 of `interleave128` now applies the special permute operation using the vector from Step 3 as control vectors. The data vectors passed to this `vec_perm` operation are special lookup tables containing the 8-bit representation of the 4-bit numbers 0-16 interleave with `0x0`.⁵⁷ These lookup tables (`table1`, `table2`) are listed as part of Appendix C.5. The two lookup tables `table1` and `table2` are actually just 8-bit representations of the 8-bit values 0-16, padded accordingly with 0 bits. For example in `table1`, the bits are padded to the right and `0 = 0000 0000`, but `1 = 0000 0010` and `7 = 0010 1010`. Likewise in `table2`, the bits are padded to the left, thus `0 = 0000 0000`, `1 = 0000 001` and `7 = 0001 0101`. Using a `vec_perm` operation with these lookup tables and our resulting vector from Step 3, results in a vector consisting of lower 64 bits of the original vector interlaced with 64-bits of 0.

Steps 1-4 are performed twice in parallel, once for each of the input vectors. The results from both input vectors are passed into Step 5. Steps 1-3 are identical for both input vectors. Step 4, uses a different lookup table for each original vector – `table1` for the first vector and `table2` for the second.

⁵⁷The careful reader will remember that `vec_perm` looks at the lower *five* bits of every byte, not just the lower *four* as we would like it to now. This is not a problem however, because we simply pass the same table for both data vectors v_a and v_b , thus making the fifth bit irrelevant. The highest bit selects which table to look from, but since both tables are the same, it could just as well be ignored.

Step 5 of `interleave128` now takes these spaced copies of the original vectors and XORs them together, to form a single interleaved copy of the lower 64-bits of each original vector. The code for all five of these steps is listed in Appendix C.5.

Steps 1-5 are then repeated to process the high bits of originally supplied vectors.

After the complete execution of `interleave128` the original first vector now contains the low bits of the resulting 256-bit interleaved vector, and the original second vector now contains the high bits of the interleaved vector. From this basic `interleave128` operation I build all of the AltiVec swizzle code using an algorithm which I detail in Section 2.4.1.

Finally the AltiVec has a series of data stream controlling functions which are used to control how data is pre-fetched for the processor. Issuing a Data Stream Touch instruction (`vec_dst`), signals to the processor that the memory specified by that instruction is about to be used. The processor can then make an informed decision about what data it should be requesting from memory. By pre-warning the processor, data can be loaded into the Level 2 cache before it is needed with better accuracy than the processor would otherwise accomplish on its own. Pre-loading data into the Level 2 cache avoids very expensive processor stalls due to “cache misses” (when an instruction asks to load data from memory which has not yet been pulled into to the *much faster* Level 2 or Level 3 processor caches). Data stream instructions are particularly useful in loops in which the compiler might not be able to determine the next section of data needed from memory[19]. I have coded my implementation of Bitslice, both the AltiVec swizzling code and encryption code, to take advantage of these instructions. The AltiVec data stream instructions for controlling data pre-fetch are listed in Table 4.

<code>vec_dst</code>	Vector Data Stream Touch – instructs the hardware to open a “data stream” with which to begin fetching the specified data into cache.
<code>vec_dss</code>	Vector Data Stream Stop – instructs the hardware to close an open “data stream” and discontinue fetching the specified data into cache.

Table 4: AltiVec Data Stream Instructions

The code containing all of these various AltiVec instructions is attached in Appendix C; further discussion of the performance increase offered by these functions can be found in

Section 2.5, and the next section on performing Swizzling operations on the AltiVec.

2.4.1 Swizzling on the AltiVec

One of the key features to my Bitslice implementation is its real world usability. This usability comes from addressing the concern of data swizzling – a concern that prior to my implementation had rendered Bitslice impractical for normal encryption/decryption. This swizzling speed jump is based both on the power of the AltiVec vector processing unit and the flexible design of its instruction set. The special `vec_perm` instruction described in Section 2.4 makes extremely low-cost bit-level interleaves possible.

The swizzling algorithm used in this paper is based on the interleave function `interleave128` described in detail in the previous section. Figure 10 shows an example of a bit-level interleave. The source code for the actual 128-bit version of this function is that given in Figure 9. The full listing of the file containing `interleave128` and supporting functions is given in Appendix C.5.

From this single interleave function, I have built each of various the AltiVec-based swizzling functions (listed in Section 2.5.1). Each of these functions is built using a simple algorithm – very similar to one you might use on paper – designed for sorting from n blocks of k length to k blocks of n length. This algorithm is described in detail below.

Begin with a stream of n blocks, each of k length.⁵⁸ We wish to sort these into a stream of k blocks each of n length. For an example I will use a stream of four blocks all of two length. Such a stream would appear as follows:

$$\{1, 1, 2, 2, 3, 3, 4, 4\}$$

⁵⁸For this example I assume k is the exact length of the vectors expected by our interlace operation. I comment on changes caused by the algorithm by the failure of this assumption at the end of this section.

If we break this stream into four 2-unit vectors, we have:

$$\text{vectors: } v_1 = \{1, 1\}, v_2 = \{2, 2\}, v_3 = \{3, 3\}, v_4 = \{4, 4\}$$

Step 1, we take our list of vectors and arrange them on the paper in two equal length lines. These two lines will each contain half the total number of vectors. We will call the top line the “high” list and the bottom line the “low” list.

$$v_1 = \{1, 1\}, v_2 = \{2, 2\}$$

$$v_3 = \{3, 3\}, v_4 = \{4, 4\}$$

Step 2, we apply an interleave operation to each of the columns in order from left to right. Each time we perform an interleave operation on a column of vectors we append the two resulting vectors to a list “new vectors.” This list of “new vectors” will be used for the next round of the algorithm. Appending to this list of new vectors is the equivalent to writing the resulting vectors in left to right order across the sheet of paper. When writing them on a piece of paper, we would do so in a fashion matching the two lists made above (filling the first line with half the vectors, then beginning the second line). In the computer, we construct a full list, of all the newly interlaced vectors, which we will in the next round divide again into two lists – two lines on paper. The results of these interleave operations and the new list of vectors are shown below:

$$\text{interleave}(v_1, v_3) = \{1, 3, 1, 3\} = (v_1 = \{1, 3\}, v_3 = \{1, 3\})$$

$$\text{interleave}(v_2, v_4) = \{2, 4, 2, 4\} = (v_2 = \{2, 4\}, v_4 = \{2, 4\})$$

$$\text{new vectors: } v_1 = \{1, 3\}, v_3 = \{1, 3\}, v_2 = \{2, 4\}, v_4 = \{2, 4\}$$

Step 3, we need only set our original list of vectors equal to our new list of vectors, and

repeat starting with Step 1.⁵⁹

This simple three step algorithm is repeated $\log_2 n$ times where n is the number of blocks of length k that we began with.

The code to implement this swizzling algorithm is equally straight-forward. The only additional concern is the number of blocks affected by each swizzling operation.⁶⁰ For this example, we made the assumption that each interleave operation would affect exactly two blocks of data each of the original block-size. However, this is not always the case as is demonstrated my AltiVec swizzling code. Bitslice swizzling requires swizzling of sets of 32- or 64-bit blocks using interleave operations based off 128-bit vectors. Each interleave operation affects then four or eight blocks (two vectors, two or four blocks per vector), not just two (two vectors, one block per vector). The number of blocks which fit inside a single vector affects the total number of vectors necessary to contain all the blocks. This in turn affects the total number of interleaves which we must do each round. This change is ignored in the paper algorithm presented above as the number of interleaves necessary per round is determined by the length of our two lines. For convenience the code which I have written to perform AltiVec swizzling code generation uses “lists” to store the vectors, and only executes enough interleave operations to exhaust both lists, thus also remaining agnostic regarding the block-length/total number of vectors in use. The full source listing of my perl code used to generate AltiVec swizzling code can be found in Appendix C.3.

Having now discussed the DES algorithm behind Bitslice, the implementation changes necessary for Bitslice, and the hardware on which I have gone about making those changes, it is now time to discuss the results. The next two sections deal with the extensive suite of tests which I designed for my implementation, and the resulting data gathered from those tests.

⁵⁹When performing this algorithm on paper, Step 3 essentially becomes part of Step 2 as we can write the vectors into two lines directly without needing to list them first.

⁶⁰Otherwise stated, this is the relation between the size of our vectors used for interleaving and the size of our blocks being interleaved.

2.5 Performance Testing

The premise for this entire project has been that current cryptography is substantially slower on modern hardware than it should be. I set out to not only find, but demonstrate ways by which cryptography could be implemented more efficiently on modern architectures. As such, testing of the performance of my implementation was of utmost concern, and, as you will see in these three sections on performance, I have designed a large number of tests to assure accurate measure of the performance of my implementation.

Before I discuss the tests I implemented however, let me briefly mention my choice to use empirical instead of analytical methods for performance analysis in this paper. It is common practice for computer scientists to use various analytical methods with which to gauge algorithm performance. I found, however, that the performance of my implementation could be better gauged (particularly real-world performance) not by mathematical analysis but by real data collection. Computer scientists most commonly speak of performance of an algorithm in terms of the mathematical relationship between the size of the data set and the time the entire execution takes in worst (or average) case. This worst case runtime analysis is not something which can be easily compared between Bitslice and conventional DES implementations due to a couple factors. For one, DES and Bitslice DES are essentially the same algorithm. DES is essentially an $O(n)$ ⁶¹ algorithm with a number of large constants.⁶² Bitslice DES is actually the exact same algorithm with generally smaller constants⁶³ and the additional cost of swizzling.

The second reason that Big-O comparison doesn't really apply here is the cost of Swizzling in Bitslice DES. Swizzling is an entirely separate algorithm from Bitslice DES, but it is still an important part of the real cost of Bitslice. Previous researchers have largely ignored

⁶¹Big-O notation is a common way of describing algorithms in computer science. $O(n)$ says that the worst case runtime for the algorithm in question has a linear relationship with the size of the data set.

⁶²Conventional DES implementations also have the additional cost of key-scheduling (a cost which is only one-time in Bitslice and is paid at code generation time). Key scheduling is also approximately $O(n)$.

⁶³This is because the computed run time can be divided by some constant s where s is the number of blocks processed per round of Bitslice.

the cost of swizzling, disregarding it as a separate cost. However, if the goal is to produce an implementation with hopes of being practical, that cost can not be ignored. Swizzling as I have implemented it on the integer unit is in worst case $O(n * k)$ where n is the final number of blocks and k is the length of those blocks. This is essentially an $O(n^2)$ algorithm. Swizzling as I have implemented it on the AltiVec Unit is $O(\log_2(n) * k)$ or $O(n * \log(n))$ – a significantly faster algorithm (at least for large value of n). Both integer unit and vector unit swizzling algorithms have rather small constants. This runtime analysis is unfortunately does not really explain the differing performance of the two algorithms, and thus for my implementation, I have adopted a more empirical approach towards gauging the performance of my implementation.

To gather performance data on my algorithm I used a set of three main performance tests on my code. These include: swizzling tests, head-to-head encryption comparisons, and swipe size⁶⁴ comparisons. Swizzling tests were used to help me grade the performance of my swizzling algorithms (both for IU and AltiVec). Head-to-head encryption tests were employed to demonstrate how my implementation competes against others. Head-to-head tests have a rather obvious need, as I was looking to develop an implementation which offered an efficiency improvement over any on the PowerPC platform. Finally, swipe tests are focused again in terms of real-world usage. Swipe tests allowed head-to-head tests of libdes and Bitslice when limiting Bitslice’s swipe size to sub-optimal levels. This allowed me to simulate “real world” uses in which the program using the Bitslice algorithm might not be able to keep it fed to full capacity. Swipe test data could help a future implementer decide at which point they might choose to use libdes vs. my Bitslice in their application.

As no useful optimization is ever accomplished without first really understanding what is happening inside the program to be optimized, I used a variety of tools to see what was

⁶⁴*Swipe size* refers to the number of blocks processed per round of Bitslice execution. For example, if you were to execute a round of Bitslice, providing it with only 18 of a possible 32 blocks (on a 32-bit integer unit), I would define this round as having a swipe size of 18. In swipe size tests, I capped the maximum number of blocks passed into my Bitslice implementation at a number lower than what Bitslice is capable of processing per-round. I then watched the resulting execution times (and throughputs) for comparison with libdes and other standard DES implementations.

“really going on.” Apple Computer provides a host of performance tools for use on their PowerPC systems titled the Computer Hardware UnDerstanding toolkit (CHUD Tools). It was these CHUD tools which, in addition to my own tests, I used most extensively for understanding my implementation. Table 5 shows a list of the CHUD tools that I used in testing Bitslice.

Shikari.app	sampled various parts of my code and various internal performance counters.
Sampler.app	took samples of running code and then allowed browsing of the execution tree.
amber	produced execution traces of my code
simg4	analyzed trace files for cycle counts, pipeline stalls, etc.
Reggie.app	monitored performance counters on the G4 while running my code.
acid	displayed trace file statistical summaries.

Table 5: CHUD Performance Tools

With the CHUD tools in addition to my own internal performance routines, I was able to profile my code to learn where it was spending all of its execution time, how many times I was causing cache misses, and where I might better optimize my algorithms. Without the use of these tools I would have just been guessing where to optimize. In addition to the CHUD tools, I wrote my own series of performance tests, the results of which you see listed in this and other sections.⁶⁵ Table 6 outlines the use of the three most useful performance testing flags which I built into my implementation.

2.5.1 Swizzling Tests

As I have stated in previous sections, one of the unique features of my Bitslice implementation is my concentration on the real world problem of real data throughput. One of the ways by which I achieved success in this area was to use a fast swizzling algorithm. I wrote what I believe was a relatively efficient integer unit swizzling algorithm, and then replaced it with an optimized Altivec algorithm (described in detail in Section 2.4.1). Table 7 shows some performance results comparing my integer unit and vector unit swizzling code. The results

⁶⁵Sample output information from my own performance tests is provided in Appendix D.

- S Runs libdes and Bitslice encryption speed test for head-to-head comparison. Runs tests including both static and random data, using integer unit, and vector unit encryption, and integer unit and vector unit swizzling. Supplying an optional n argument, where $0 < n \leq 9$ will cause the program to perform each test n times in a row assure good data averages.
- W Runs swizzle tests comparing integer unit swizzling to vector unit swizzling of both 32 and 128 blocks at a time. Supplying an optional n argument where $0 < n \leq 9$ will cause the program to perform each test n times in a row to assure good data averages.
- E Runs swipe size tests for both integer unit and vector unit encryption. Supplying an optional n argument where $0 < n \leq 9$ will cause the program to perform each test n times in a row to assure good data averages.

Table 6: Performance Testing Flags

shown in Table 7 can be generated from the bitslice executable through the use of the “-W” execution flag.

Algorithm	64-bit Blocks/Second	Mbps	MBps
fill_data32_iu	950,000	58.0	7.2
extract_data32_iu	1,180,000	71.8	9.0
fill_data32_vpu	7,320,000	447.0	55.6
extract_data32_vpu	6,700,000	409.0	51.1
fill_data128_iu	570,000	34.6	4.3
extract_data128_iu	560,000	34.0	4.3
fill_data128_vpu	6,210,000	381.0	47.7
extract_data128_vpu	6,770,000	410.4	51.3

Table 7: Integer Unit vs. Vector Unit Swizzling

When looking at the individual performance data in Table 8 one can see that moving to the AltiVec for swizzling offered a very large gain in swizzling performance. This performance gain can make Bitslice implementation a practical encryption/decryption tool. Previous studies of the efficiency of Bitslice, such as that of Lauren May 2000[15], neglect swizzling costs and don’t offer computer scientists a practical algorithm. May’s study mentions the necessary data transformation (swizzling) necessary to use Bitslice, however offers no solution towards doing such efficiently. Although Bitslice looks fast on paper in previous studies, if one were to consider the additional cost of swizzling the incoming and outgoing data,

Bitslice would have been much slower than the best normal DES implementations of the time. With the swizzling (and encryption) speeds I have presented in this paper however, my implementation of Bitslice encrypts/decrypts in real world situations much more rapidly than other DES implementations. The individual encryption speed results for Bitslice in comparison with those of other algorithms, are shown in the following section covering Head-To-Head encryption tests.

2.5.2 Head-To-Head Tests

The second, and likely most interesting, performance metric shows how Bitslice performs when pitted against libdes on the G4 and DES implementations on other platforms. I have performed a number of head-to-head encryption speed comparison tests and built a speed testing facility directly into my Bitslice implementation. The “-S” flag passed to the executable will activate built-in comparison tests for Bitslice at various different swizzling/encryption algorithm combinations. This flag will also cause Bitslice to run the exact same test data through a libdes speed test. All results listed for the PowerPC processors were generated using these built-in speed testing capabilities of my Bitslice implementation. Test results were gathered running on otherwise completely idle systems. The results included in Table 8⁶⁶ were generated using an “-S4” flag, performing four rounds of each test for good data averages. Sample output from an actual “-S” run can be found listed in Appendix D.

Bitslice performance comparisons listed in Table 8 were executed with random input data, fed in continuously from memory. In order to allow comparison with both previous Bitslice implementations (which did not perform swizzling) as well as real-world applications such as IBM’s dedicated crypto PCI card and the PowerPC libdes implementation, I tested Bitslice ECB encryption with all combinations of integer unit and vector unit swizzling and encryption code. The particular data you see listed in Table 8 is generated using a test space of 6553600 blocks of data. Entries listed with swizzling turned on (for either the integer

⁶⁶The results included in Table 8 taken from other paper were each given in Mbps form. I have from that extrapolated the MBps, and an approximate blocks/second.

Architecture	Program	Swizzle	S-Boxes	Blocks/Second	Mbps	MBps
G4 550Mhz	libdes	N/A	IU	500,000	30.5	3.8
G4 550Mhz	Bitslice	IU	IU	410,000	24.8	3.1
G4 550Mhz	Bitslice	AltiVec	IU	1,180,000	71.6	9.0
G4 550Mhz	Bitslice	None	IU	1,550,000	94.8	11.8
G4 550Mhz	Bitslice	IU	AltiVec	270,000	16.7	2.1
G4 550Mhz	Bitslice	AltiVec	AltiVec	1,950,000	118.8	14.8
G4 550Mhz	Bitslice	None	AltiVec	4,260,000	261.2	32.4
G4 1.25Ghz	libdes	N/A	IU	930,000	56.5	7.1
G4 1.25Ghz	Bitslice	AltiVec	AltiVec	3,740,000	227.3	28.4
G4 1.25Ghz	Bitslice	None	AltiVec	8,350,000	510.7	63.5
P3 500Mhz	Bitslice-MMX	None	MMX (64-bit)	1,440,000	92.0	11.5
Alpha 300Mhz	Bitslice-Alpha	None	IU (64-bit)	2,140,000	137.0	17.1
IBM	Hardware	N/A	Hardware	2,290,000	146.4	18.3

Table 8: Results from DES - ECB Tests

or vector unit) include both fill and extract swizzling operations – these are to simulate normal operation involving both reading data from a heterogeneous stream, processing it, and returning it a heterogeneous form. Entries listed with swizzling set to “none” are useful for comparison with previous implementations such as those found in May[15] or Biham[4] but do not represent real world performance as they do not consider the cost of swizzling the data.

As one can see from the results, my mid-range laptop computer executing my Bitslice implementation competes well against both previous Bitslice implementations and other DES implementations in either in hardware and software. The peak performance of my algorithm using “real-world” data is approximately four times faster than libdes – the current fastest open source DES implementation on the PowerPC. When considering the raw encryption speed (neglecting swizzling costs, as previous Bitslice implementations have done) my Bitslice implementation sustains a throughput around nine times faster than libdes when running on the same hardware with the same data. For comparison, I also tried testing my implementation on PowerPC hardware slightly over twice the speed of my own. When running my implementation on that hardware against libdes also run that faster hardware,

my implementation again faster than libdes – a difference of a similar ratio to that seen on my slower hardware. Not surprisingly, when comparing performance of my Bitslice implementation on 1.25Ghz processor with that of the hardware crypto module from IBM, the 1.25Ghz G4 performs better. What is encouraging however, is how my now year-old, mid-range laptop running my Bitslice DES achieves encryption speeds only 10% shy that of dedicated crypto hardware.

I should also note that although I believe my implementation to be well optimized, I will not pretend that my optimized code is even in the same league as those those who spend their days counting clock cycles and watching instructions flow through CPU execution pipeline simulations. A promising result of this research was that with the application of only a few relatively straightforward parallelization techniques, real-world speed improvements were achieved. Specifically, a four fold speed improvement was achieved over an implementation (libdes) which has undergone optimization for over 10 years. These results show much promise for future cryptographic implementations designed to exploit parallelism, and further serve to highlight the discrepancy between the small percentage of processing power our current cryptographic algorithms are designed to use and the vast amounts of power currently available on modern hardware.

2.5.3 Swipe Size Tests

A final test important for my implementation was the swipe size test. As not every application programmer may wish to use the easily parallized ECB encryption mode, and as not every application will wish to pay the latency penalty to run Bitslice DES at its full throughput, I have done testing to help identify critical tradeoff points. These are points below which it is better to use libdes or some other standard DES implementation instead of my Bitslice DES. These points arise because small enough sets of input data will not fill the entire Bitslice array, and thus leave Bitslice running with the same (or worse) register inefficiency as normal DES. Swipe size testing can be activated by running my implementa-

tion with the “-E” option. “-E” will cause my implementation to display results of a swipe tests against both IU and VPU Bitslice variants. Table 9 shows abbreviated results from running one such test.

Algorithm	Swipe Size	Swizzle	Encryption	Blocks/Second	% Efficiency
libdes	1	N/A	IU	500,000	100%
Bitslice	4	vpu	vpu	70,000	14%
Bitslice	8	vpu	vpu	140,000	28%
Bitslice	16	vpu	vpu	280,000	56%
Bitslice	28	vpu	vpu	480,000	96%
Bitslice	29	vpu	vpu	500,000	100%
Bitslice	30	vpu	vpu	520,000	104%
Bitslice	32	vpu	vpu	550,000	110%
Bitslice	40	vpu	vpu	690,000	138%
Bitslice	60	vpu	vpu	1,000,000	200%
Bitslice	100	vpu	vpu	1,700,000	340%
Bitslice	128	vpu	vpu	2,100,000	420%

Table 9: Test Data from DES Swipe Size Tests

To find this point of most efficiency, the swipe test runs several individual encryption speed tests each time applying an artificial limit to the swipe size accepted by my Bitslice implementation. These numbers when compared with those in the Section 2.5.2 show Bitslice taking a slight performance hit. In order to support the necessary flexibility for these tests, I made small modifications to my encryption code which may explain this performance hit. These tests show an approximate trade-off point in my algorithm at around 29 blocks per swipe. Programmers wishing to encrypt fewer than 29 blocks at a time are suffering a performance hit to use this Bitslice code. Knowing this tradeoff point, an application programmer can decide whether or not his cryptographic throughput is great enough to warrant the use of Bitslice. It is worth noting that this tradeoff point is quite high (about double what I had hoped/expected it to be). The height of this trade-off point offers some indication of why Bitslice has seen little adoption; programmers need to be able to provide over 30 parallel data sets for Bitslice to be practical. Providing at least 30 parallel data blocks is very easy to do when performing ECB encryption over large files, but is not as easy

to do when encrypting only a few small data sets all with CBC encryption.

One condition, which I do not account for, but which should be accounted for in a production system, is ignoring extra empty blocks. If passed an array containing fewer than the expected number of blocks to be swizzled, my AltiVec swizzling code will not ignore any extra empty blocks. The integer unit code will successfully ignore the extra blocks; however, the AltiVec code as written does not perform such optimizations. Ignoring extra blocks during swizzling would certainly affect the swipe size number presented above. This is just one of likely many optimizations which could still be applied to my code to allow better performance in the swipe tests.

With the performance tests listed in these previous three sections, along my extensive testing under the CHUD tools (not discussed here), I have produced an implementation which is at least faster than any other open source DES implementation on PowerPC hardware. This implementation is also one which competes well against (and out most cases out-performs) previous Bitslice implementations and normal DES implementations in both software and hardware. The final remaining piece to test in my implementation is the most crucial aspect of security: implementation correctness. The entire web of confidence and trust surrounding the security of my implementation is based on my implementation's compliance with the DES specification[11] and DES's own proven security. The next section describes the testing measures I developed to assure proper correctness and specification compliance of my implementation.

2.6 Assurance Testing

Data Security, or Information Assurance, as it is often called, is based not only on the idea of keeping data secret, but also on the concept of instilling trust in those data. Trust in information means that 1. one knows the information is from whom it are claimed to be from. 2. the information is exactly that which it is claimed to be. 3. the information has not been altered since the time at which it was last "trusted." Trust in turn is based on the idea of

testing. Trust in the data requires not only testing of the information itself (through various independent measures), but also testing of the tools which generate and provide security for that information. The tools of security, including the Bitslice implementation I propose in this paper must be rigorously tested. My Bitslice implementation underwent a large set of tests, including both assurance or performance testing.⁶⁷ Those tests assured that the implementation I had created functioned properly (according to the DES specification[11]) with all varieties of data and circumstances.

The official DES algorithm is accepted as secure and all trust in DES implementations stems from that belief. Thus to instill trust in my implementation it is only necessary that I prove it conforms to the algorithm as stated in the specification[11]. For a standard with which to compare, I have chosen Eric Young's libdes implementation. Libdes has been in use for over 10 years and ships as part of many other cryptographic libraries including Apple Computer's own Security.framework.⁶⁸ I have extremely high confidence in the correctness of libdes.

I designed a large series of tests, to test my implementation both against libdes as well as against itself.⁶⁹ These tests, like the performance tests from the previous section are built into my Bitslice implementation. In Table 10 I list the most important of Bitslice's built-in assurance tests, along with their basic descriptions. For a sample of output from each of these tests, consult Appendix D.

My Bitslice implementation has successfully passed all ECB encryption tests included with libdes. I have left my implementation running overnight in “-T” mode, during which

⁶⁷In fact, the majority of my work implementing Bitslice DES was in writing testing code. Although it receives a smaller section of the paper, the assurance testing piece of Bitslice DES was by far the more time-consuming (and difficult) of the two testing suites to write (or at least write correctly).

⁶⁸The “.” here between Security and framework is intentional, and is part of the naming convention used under Apple's OS X.

⁶⁹To test an implementation against itself, refers to the common practice in cryptography of testing an implementation against built-in test data, or by performing circular operations such as encryption followed by decryption and confirming that the original and final plain-texts agree. In Bitslice, it was particularly important to design internal swizzling tests to confirm that that data would both swizzle in and swizzle out of Bitslice arrays correctly.

- T “-T” tests my implementation against libdes, using random data. Causes Bitslice to enter an infinite test loop only producing output upon test failures. The user can end the test loop using ^C, at which time he or she is presented with statistics regarding how many blocks were successfully processed and confirmed correct.
- L “-L” tests my implementation’s swizzling code – that which is necessary to pull data into and out from the Bitslice encryption arrays. “-L” tests the Integer Unit swizzling code. Supplying “-La” flag performs additional tests using AltiVec swizzling code. Each test prints “OK” if correct or failure information upon encountering an error condition.
- P “-P” performs a practice run (searching, encryption and decryption) against known libdes practice data. Each test prints “OK” if correct or failure information upon encountering an error condition.

Table 10: Assurance Testing Flags

time it successfully encrypted several billion random blocks in exact agreement with lides.⁷⁰ My implementation has also successfully passed each of the suite of tests which I designed specifically for Bitslice. With the positive results from all of these tests, I am very confident that my implementation is both fully compatible with libdes, and correct and secure in accordance with the official National Institute of Standard’s and Technologies (NIST) DES specification[11].

3 A Greater Context

This section discusses the academic and industry context for my implementation. I discuss here such factors as the viability of DES, the history of Bitslice (including previous implementations), the roots of my implementation, the vast improvements my implementation offers, and some future uses for Bitslice. This section complements Section 1 by providing some academic and technical background to my research.

To begin, I will first discuss the viability of DES; DES is an older, proven algorithm, and DES has undergone over 20 years of rigorous crypto-analysis. DES, because of its age, has

⁷⁰The exact number of blocks which it successfully encrypted is unknown, as the 32-bit integer I used to count each block overflowed (probably several times).

several shortcomings: DES's 56-bit key length is too short by today's standards; its original design is extremely inefficient on today's modern computers; and its block length is shorter than desirable for the larger data sets of today. Regardless of these shortcomings, DES, at least in its 3DES form, still remains secure and still enjoys wide-spread use. DES/3DES are in use every second of every day throughout the world, and will continue to be used so for many years. What this Bitslice implementation provides is better a way of using this older technology efficiently on modern computers. Although this particular implementation does not have the same life expectancy as parallel implementation of AES would, it still serves to demonstrate the concepts of parallelism so central to this paper, and is in that sense very useful.

Bitslice DES is also by no means a new DES implementation. As I mentioned above, Bitslice was originally proposed by Eli Biham in 1997. Bitslice has seen rather limited attention since then. It is common to find Bitslice in cracking programs and key searching programs such as John the Ripper, or distributed.net's client for the RSA DES challenge. But aside from those key-cracking programs I have seen few programs which support Bitslice, and have not yet seen an encryption library which uses Bitslice. Bitslice has received relatively little attention since its initial introduction, a fact I attribute to two factors.

The first factor is a mindset difference. Programmers have been taught for years to find speed gains through the use of "low latency" algorithms.⁷¹ As we have continued to move towards a world of increasingly massive parallel computational power, the programmer's mindset has been slow to make the necessary change from "low-latency" to "high-throughput" computing. Bitslice is *not* a low-latency algorithm. One iteration of my implementation of Bitslice DES takes approximately 29 times as long as a single iteration of libdes DES. That means that doing one block at a time with Bitslice is 29 times slower than with libdes. Bitslice is instead a *high-throughput* algorithm. Bitslice is not limited to one block per round

⁷¹Low latency algorithms return data as quickly as possible. This is in contrast to high-throughput algorithms which attempt to process as much data as possible in the shortest amount of time. High-throughput algorithms will generally process more data than a low latency algorithm over a given span of time, but each individual piece of data may not be available as quickly.

of execution, and instead can compute up to 32 blocks per round on an integer unit, and 128 blocks per round on the AltiVec VPU. As the common programming mindset shifts from low-latency to high-throughput, I believe Bitslice DES and other parallel algorithms will be used more.

The second factor that has limited the attention given to Bitslice has been that Bitslice until now has had poor consumer hardware to run on. As I stated previously, Bitslice gets its big speed gain from really wide parallelism. Bitslice has been implemented on (non-consumer) 64-bit processors (the Alpha), 32-bit processors (x86), and even register starved 64-bit VPUs such as the MMX. However, it isn't until one takes Bitslice to a fully developed 128-bit vector processing unit such as the AltiVec that one really begins to see Bitslice's full potential.

Even though Bitslice has had these two factors limiting its adoption, Bitslice has been implemented a few times on various platforms. Biham originally wrote the Alpha implementation of Bitslice[4], and Kwan wrote the original x86 implementation[29]. A later implementation was developed for the MMX instruction set on the Pentium III[15]. As far as I know the Bitslice implementation in this paper is the first implementation of Bitslice written to the AltiVec instruction set and the first Bitslice implementation to include a truly fast algorithm for data swizzling.

My implementation draws its roots from previous work done by Kwan in 1998[29]. Kwan was interested in the efficient implementation of logic-gate-based S-Boxes, and has published at least one paper on the topic[13]. Kwan provides source code to a key-searching Bitslice implementation at his website free of charge[29]. My code originally stemmed from Kwan's free Bitslice key-searching code. My current source code, however, no longer shares any resemblance to Kwan's code, with one exception: the S-Box implementations I use remain Kwan's.⁷² In order to provide the kind of encryption/decryption, swizzling, and AltiVec support which I desired for this implementation, I not only rewrote the majority of Kwan's

⁷²However, you will notice when browsing the source, that I do provide a perl script with which to generate AltiVec ready versions of his S-Boxes from his original S-Box code.

main source file (Appendix C.7) but I also wrote several additional perl scripts (Appendix C) including one (Appendix C.1) to generate encryption/decryption functions. Nearly all the code which Kwan provides at his website is generated by unpublished scripts. It is likely that my encryption generation script would show many similarities to his unpublished key-searching generation script.

Although my implementation stems from Kwan's work, there were several choices Kwan made that I did not choose to repeat. The first of these choices was his use of 56-bit keys instead of the standard 64 bit keys.⁷³ Although 56-bit keys are economical in terms of memory, I found them to be more of a hassle than a help. The DES specification[11] dictates that all keys must have an odd parity to each byte (the number of 1s in each byte are odd⁷⁴), but I found that this restriction is often not upheld in real world keys. Keys which do not follow this specification, and are reduced to 56 bits, no longer report the true value key and thus operations such as fill/extract tests fail for these keys. I also found that when implementing the original versions of my Altivec swizzling functions 56-bit keys were inconvenient (as they are not a power of 2). In a future revision one could optimize my swizzling and encryption implementations to also support 56-bit keys, but for the time being I have not done so.⁷⁵

A second choice Kwan made that I chose not to repeat, was to organize bits in memory in a right-to-left (high-to-low, or little-endian) fashion, even though the DES specification[11] is written from a left-to-right (low-to-high, or big-endian) perspective.⁷⁶ The data come in

⁷³As previously mentioned, 56-bit keys are really all that is necessary as DES only uses 56 bits of the original key.

⁷⁴If there are an even number of 1s in bits 1-7, then bit 8 is a 1. If there are an odd number of 1s in bits 1-7 then bit 8 is set to 0.

⁷⁵Actually, my encryption code already contains support for generating code capable of using 56-bit keys. I originally wrote in this support for compatibility with Kwan's implementation. The integer unit swizzling code I provide (see Appendix C.4) could also be trivially modified to support 56-bit keys. My Altivec swizzling code has also undergone recent modifications which should make it simple to add 56-bit key support. However, such support has at time of writing not been added to the `generate_swizzle_vpu.c.pl` code and is left to the reader.

⁷⁶This is actually a slightly "loose" use of the terms "big-" and "little-endian" as those technically refer to the order in which the bytes are stored, not the bits. Bits are always stored within a byte with the high bit to the "left" and the low bit to the "right." What Kwan has done in his code, however, is to store the 64-bit blocks of DES in a right-to-left fashion, which some would refer to as "little-endian" as I have done

ordered in a left-to-right fashion and must leave in a similar left-to-right fashion according to the specification; thus, it does not make sense to convert it to a right-to-left ordering in the meanwhile. For my own ease in programming, if nothing else, I have chosen to switch my code to be left-to-right and I provide a set of tools in the tools subdirectory to facilitate the switch on Kwan's original code as well.

In addition to those two small changes I made to Kwan's design, there are also a host of improvements I made in my code. Most notable of these improvements are AltiVec processor support and support for both 32-bit and 128-bit swizzling and encryption. This is the first implementation of Bitslice I know of to support a full 128 bits of parallelism. Sections 2.5.1 and 2.5.2 have already discussed the speed gains that my implementation realizes from this addition. A final feature, new with my implementation is my emphasis on real-world performance. Mentioned in Section 2.5, this emphasis has for the first time produced an implementation of Bitslice which can and should prove useful to the security application programmer. Previous implementations have concentrated on the encryption side alone, neglecting the essential swizzling code. My implementation takes both into account, and, with the power of the AltiVec processor, demonstrates a full Bitslice implementation of superior performance.

As to the future of Bitslice? Although Bitslice has received little attention in the past, I believe that that may change in the near future, if not for Bitslice DES, then for Bitslice-like algorithm implementations. Not only are programmers (slowly) becoming more aware of the benefits of high-throughput programming (as Bitslice embodies), but there are also already modern day security applications which could benefit by using an efficient Bitslice implementation. The most obvious of these is the IPSec⁷⁷ VPN technology which has been gaining popularity in the last couple years. IPSec can be used to encrypt all network packets leaving your computer. When one is dealing with a network connection of high activity, that

here.

⁷⁷IPSec: Internet Protocol Security, is a set of extensions to Internet Protocol version 4 (IPv4, also part of the Internet Protocol version 6, IPv6, specification) which allow all traffic between any two hosts to be encrypted.

can be thousands, if not tens of thousands, of packets per second. Bitslice, as I have already demonstrated, excels in this kind of situation in which there are large sets of similar sized data (even if they all have unique keys). This makes using Bitslice DES (or any Bitslice-like cryptographic implementation) an attractive option for IPSec.

I should restate here that Bitslice is also very useful for most other normal DES encryption/decryption, particularly for encrypting/decrypting large files using ECB encryption. When encrypting large files my Bitslice implementation can read up to 128 blocks at a time, and encrypt them all in the same time period. Libdes, in contrast, could read all 128 at once, but would have to encrypt each of them sequentially and would only finish the first 10-20 in the time it took Bitslice to finish all 128.

A final, still developing, factor, which may make Bitslice even more useful in the future, is the ability to run Bitslice quickly far below its optimal efficiency. In my implementation of Bitslice, it is possible to run Bitslice at 25% efficiency (using VPU code) and still maintain a speed gain over libdes. This could allow Bitslice to be useful even on much slower network connections, or in situations where few data are available and a low-latency algorithm would otherwise be preferred. Further optimization in this area would be required for Bitslice to be truly useful with small data sets.

One final future application for Bitslice is an in-application packet-level parallel implementation of Bitslice DES. Such a packet-level implementation could be used in applications such as an http server module supporting SSL or TLS encryption.⁷⁸ Instead of encrypting each http data packet separately, the http server could write all data needing encryption to a buffer. The SSL module would then use an x -bit Bitslice implementation to process up to x different packets in parallel one block from each at a time. This packet-level parallelism can function regardless of the mode (or modes) of the cipher used.⁷⁹ To the best of my knowledge

⁷⁸SSL stands for Secure Socket Layer encryption and is the method most web browsers use to connect to secure websites. TLS stands for Transport Layer Security. TLS 1.0 is the successor to SSL 2.0 and 3.0.

⁷⁹It should be noted here that it is not even required that all packets computed in parallel in a Bitslice array use the same DES mode. With minor optimizations to my current implementation, it would be possible to compute blocks of CBC, CFB, and ECB modes, both for encryption and for decryption, all in the same array.

no such system has been attempted in software to date, but clearly from Biham's work and from the results of my own work such an implementation is possible. Such a system could offer busy servers such as Amazon.com or ATM controllers a boost in their software crypto processing power – possibly allowing the use of fewer servers, each requiring no additional hardware crypto devices.

My implementation of Bitslice draws on much information found in previous work, but also clearly raises the bar for future efficient DES implementations. Furthermore, with the AltiVec optimizations I have presented, I believe that Bitslice DES is finally in a position to be useful to the application programmer, much more-so than ever before. I see applications such as IPSec as perfectly suited for the type of massively parallel cryptography as Bitslice DES provides. My implementation stems from a history of research in parallel cryptography, and adds to that research further contributions, that will help in future efficient use of existing technology.

My work on this project, regarding applying parallelism in cryptography, extends well beyond my up-until-now limited focus on Bitslice DES. As was mentioned in my introduction, I will now in Section 4 give a brief overview of how future researchers might go about applying these same techniques of parallelism to other cryptographic algorithms.

4 Applying Parallelism To Other Crypto Algorithms

The following section discusses some of my research that focused on applying parallelism to two other types of cryptography not examined in this paper: hashing and public (asymmetric) key cryptography. In previous research I examined the changes necessary to apply parallelism to any of a larger range of cryptographic algorithms. I have selected for this paper only to discuss these two other key types of algorithms and their potential parallel implementations. I refer interested readers to my previous research[26] for a full discussion of implementing parallelism in a larger range of algorithms.

4.1 Parallel Cryptography, Today

I will first look at some of the approaches which have already been used to adapt our aging cryptographic algorithms to newer hardware[17]. I will classify methods of parallelization based upon to what “level” of the computer system parallelism is applied. I will be referring specifically to network cryptography, but analogous examples exist in single user (local) cryptography. The levels of classification I will use are: per-connection (per-process), per-packet (per-file), and inter-packet (data level) parallelism[18].

A *per-connection parallelism* method is one whereby each connection to the server, or from the client, is given its own thread or process that runs exclusively on one processor. This is the most common method of parallelization, and requires no modification to the existing algorithm and often no modification to the existing server software. By running multiple instances of the server process, one can often achieve this level of parallelism. This method is limited in its ability to speed up a single connection, offering high-speed, single-connection cryptography no benefits (e.g. encrypting a single file). This level of parallelism also does not address the question of running older algorithms on newer processors. By simply running a single algorithm instance per processor, this method makes no attempt to make the old algorithm run any more efficiently on newer processors than the algorithm did on the previous architecture. Per-connection parallelism is the most common method of parallelism found in cryptography today, and the one which is most often the selling angle for multi-processor machines which run cryptographic applications. The per-connection parallelization method makes no attempt to fully utilize modern architectures I will not discuss it further in this paper..

Per-packet parallelism is a method in which connections disperse their packet processing load over multiple processors, wherein each packet is treated individually. One example is a group of processors (or threads) handling the actual logic for all connections, then placing each prepared packet in a buffer, where it is handled (encrypted) by a group of processors. In this design, a single processor might handle packets from various connections, or a sin-

gle connection might use packets encrypted by various processors. Per-packet parallelism is similar to the design of specialized cryptographic hardware, where the cryptography portion of an application is offloaded to a specialized processor. Many current algorithms lend themselves well to this kind of parallelization, but, surprisingly, I encountered no cryptographic software implementing this this per-packet parallelism.

Intra-packet parallelization is the most difficult type of parallelism to introduce post-facto, since it depends heavily on algorithm design. This paper has focused on this type of parallel processing, but it is one which has historically not been addressed as much as other methods. An example of this method is a block-cipher such as DES running in ECB (Electronic Code Book) mode, whereby each block of the message is computed independently of the others and could be computed in parallel. This level of parallelization requires changes to the implementation of the cryptographic algorithm itself, depending no longer on the flexibility of the hardware or operating system upon which it is run.

The following two sections deal with two cryptographic algorithms from different branches of cryptography than DES. In each I describe the algorithm in brief detail and offer an overview as to how one might approach the problem of applying parallelism to it.

4.2 Hashing Algorithms: MD5

Hashing algorithms take in a normally large block of data and compute a unique “hash” value, much shorter than the original block. This “hash” can be then passed around with, or independently from, the original block of data, and can be used to verify the integrity of the data. On common current use for hashes is for verifying the integrity of files downloaded over the internet. Often web servers, along with providing a file, will also provide a webpage listing the cryptographic hash for that file. A user can then verify the file they received against the hash provided by the server, to assure they got the correct unaltered file. Hashing is also useful for assuring that the files on your hard-drive where not tampered with. Hashing of all files, or at least all executables, on servers with multiple users is a relatively common

security practice. Administrators can quickly confirm by checking hash values that important files have not changed. Hashing functions also used often in conjunction with Public Key Cryptography (described in detail in section 4.3) to produce “signed hashes” – short secure representations of the larger data. The hash is first computed and then “signed” to prevent a man-in-the-middle⁸⁰ from simply re-computing a hash for the altered data. Signing only the hashes, and not the entire message, saves both parties from the enormous expense of signing or verifying a signature over a large block of data, but still offers similar integrity and authenticity verification due to the uniqueness of the hash.⁸¹

In general, hash functions are already quite fast when compared to block ciphers or public key cryptography. However, the majority of hash function implementations do not support modern architectures well, and waste many unnecessary CPU cycles. Speeding-up hash functions can make current applications of hashing technology both more transparent and convenient. Applications benefiting from this speed would include the hashing and signing each network packet, the hashing of extremely large chunks of data (i.e. a downloadable file from web server), and the hashing of very large sets of chunks of data (e.g. storing and verifying hashes for all executables on a public server). Making hashing even faster on modern processors could also open the doors to potentially new uses for hash functions.

4.2.1 Message Digest Algorithm Revision 5 (MD5)

MD5 is the most common hashing algorithm; an implementation of MD5 is distributed as part of all standard Linux/Unix operating systems. MD5 is a 128-bit hash function designed

⁸⁰Man-in-the-middle attacks are network based attacks in which the malicious party sits on a network between two communicating individuals and alters communication between the two with the purpose of learning protected information from one or both parties.

⁸¹A good hash function has a very low chance of collision, and so signing a hash – thus proving that it came from you, as is verifiable with your public key – can be as good as signing the entire block of data. Given enough time, someone could find another set of data which would hash to the same hash value, and replace the original data with this new data – which would verify against the signed hash and could fool the recipient. However this type of hashing and signing is used in situations where speed is desired over security, for situations in which one worries of vulnerability to such time consuming attacks, one can alternatively perform the more time intensive operation of encrypting or signing the entire block of data and prevent such an attack.

for operation on 32-bit registers. MD5 was designed in 1991 by Ronald R. Rivest, and it is the successor to MD4 (also designed by Rivest, 1990). MD4, however, is no longer considered useful for security after several successful collision attacks in years past.⁸² Some wonder if MD5 is not reaching the end of its useful days[30]. MD5 involves a sequence of bitwise XORs and 32-bit additions over a large 512-bit data block divided into 32-bit sub-blocks. These operations result each round in four 32-bit chaining variables which are XORed with the chaining variables from the previous round. MD5 carries these chaining variables from one round to the next. Following the computation of all rounds, MD5 concatenates the four chaining variables to form the final 128-bit signature. MD5, like all hash functions, involves these chaining variables which introduce recursive dependency and prevent any direct block-level parallel implementation.

MD5, however, due to its reliance solely on XOR operations and 32-bit additions will allow a clear and simple transformation to an implementation on an SIMD instruction set such as the AltiVec. An SIMD enabled implementation of MD5 would allow packet-level (file-level) parallelism. Computing MD5 over multiple buffers in parallel on an SIMD architecture supporting 32-bit adds should theoretically cost nearly the same as computing a single buffer on the same $n \cdot 32$ bit chip, and would thus offer an n fold speedup over a normal implementation on that same chip. This could be very useful in situations in which one had to compute MD5 hashes on large sets of data – such as is found in computing hashes for all executables on public servers or computing signed hashes for all network traffic in trusted communications. An “SIMD on any processor” implementation of MD5 is unlikely, due to MD5’s heavy use of 32-bit adds in overflow conditions. Some modern processors larger than 32-bits may do support parallel 32-bit additions on a single n -bit register, thus allowing an SIMD like implementation.

I have not yet encountered any parallel-ready implementations of MD5, or more modern

⁸²A collision attack a method, which given an initial hash value and data block, can find data block which is similar and hashes to the same hash value. Attacks are known for MD4 which provide collisions on modern personal computers in under a minute. Similar attacks are available for MD5 data but take hours to days[30].

hash functions such as SHA-1 or others described in my previous paper[26]. Unlike block-ciphers, since most hash functions do not depend on lookups, but rather simple 32-bit or larger addition and boolean logic operations, it is slightly surprising that parallel implementations are not more prevalent.⁸³

4.3 Public-Key Cryptography: RSA

Public Key Cryptography (PKC) – also known as asymmetric key cryptography – is a variant of cryptography whereby there exists not only a secret key, but also a public key. With the public key, any individual can encrypt data such that it is then only decipherable using the original secret key. An individual who knows another’s public key, can also verify data signatures made using the secret key. The individual with the secret key can produce data signatures, verifiable with the public key, as well decrypt data blocks that were encrypted by the public key. An essential feature of any PKC system is that encrypting data with the public key (as well as general knowledge of the public key) lends no information about the secret key.

PKC differs greatly from “secret key” cryptography (a.k.a. block and stream ciphers). Unlike secret key cryptography, whereby the entire key-text is kept secret, PKC allows half of the key-text to be publicly known. PKC works from the assumption that there exist functions for which the application of the function is comparatively simple, yet the computing the inverse operation of such a function is very difficult (without insider knowledge). This property allows for the creation of separate encrypting key and decrypting keys. The encrypting (public) key is used to perform the non-invertible function, and the decrypting (secret) key provides the necessary clues with which to quickly perform the functions inverse and learn the original plain-text. The existence of these clues (the secret-key) saves the secret-key holder from needing to compute the function’s inverse directly. Those who do not hold the secret key, however, do not have such clues. Attackers are only able to find the

⁸³The lack of parallel implementations of hashing functions is part of a more general trend in software; most software today does not use the full parallel processing power of modern computers.

original plain-text by computing the inverse to public key system's function directly – an operation normally requiring days, years or millennia in computation time.

There are a few different functions which are currently believed to have this non-invertible property and thus are used in PKC. The most common function exhibiting this non-invertible property is the multiplication of large primes. It is comparatively simple to generate large prime numbers, and equally simple to multiply large primes; however, it is computationally very difficult to factor large numbers into unique primes. The difficulty of this computation is shown by such real world problems as the Great Internet Mersenne Prime Search (GIMPS).⁸⁴ Based on this non-invertible property of multiplying large primes, cryptographers have constructed one of many PKC systems[30]. Discussion of other functions exhibiting this non-invertible property (and the PKC systems based on those functions) can be found in my previous paper[26]. Of the various PKC algorithms, I will only discuss here the most popular system: the Rivest-Shamir-Adleman method (RSA).

4.3.1 The Rivest-Shamir-Adleman (RSA) Method

The RSA system is clearly most common PKC system currently in use. RSA keys form the heart of popular PKC infrastructures such as Pretty Good Privacy (PGP)⁸⁵ (or the compatible GNU Privacy Guard (GPG)⁸⁶). RSA keys are also used for in other systems including Secure Shell login (SSH).⁸⁷ RSA was developed by Rivest, Shamir and Adleman, and depends on our assumed inability to factor large composite numbers into large primes quickly. RSAs uses very large amounts of BigNum⁸⁸ math during both encryption and decryption. This heavy dependence on BigNum math, lends the RSA system well to the

⁸⁴GIMPS is a distributed computation project which runs software on thousands of volunteered personal computers throughout the world. GIMPS software performs calculations to determine the primality of *very* large numbers, looking for ever larger Mersenne Primes – primes having the property $M_n = 2^n - 1$ where n is itself also a prime number. <http://www.mersenne.org/prime.htm>

⁸⁵<http://www.pgp.com/>

⁸⁶<http://www.gnupg.org/>

⁸⁷<http://www.ssh.com/>, <http://www.openssh.com/>

⁸⁸Big Number, or large number, math is doing mathematical operations which require the use of more than a single register to be completed – in this case exponentiation of integers at least 1024 bits in size.

application of parallelism via a parallel enabled math library.

The efficiency the BigNum library on which RSA is implemented directly affects RSA's ability to work well on parallel architectures. Current VPUs generally ship with accompanying math libraries which can handle BigNum math at high optimization for various integer sizes[27]. Conversion of any RSA implementation from using standard BigNum libraries to using any one of these parallel-ready math libraries could provide large optimizations on architectures containing VPUs or parallel processors.⁸⁹ Likewise, since the math involved in BigNum computations uses integers larger than several registers, BigNum libraries could also be tuned utilize larger CPUs (64-bit, 128-bit) and achieve a clear speed gain.⁹⁰ Many basic mathematical functions also have optimized parallel implementations, which can allow efficient BigNum optimizations on platforms with Symmetric Multiprocessing (SMP)⁹¹ architectures[14]. PKC based on prime factorization shows clear promise of faster computation on parallel architectures over traditional architectures. That improvement is directly based on efficient BigNum computation on each platform.

PKC, like secret key cryptography and hashing, also shows promise for practical, efficient implementations on modern parallel architectures. Optimizations for these architectures, however, have not yet been made. This brief overview, and more the discussion found in my previous research[26], demonstrates the clear possibility for *much* more efficient implementations of PKC and other cryptography on modern hardware.

⁸⁹This would be another interesting example to have included with this project, having found no BigNum libraries which utilize vendor supplied vector libraries to do more efficient BigNum computation. Such a task should be very straightforward and might be of interest to someone wishing to better understand BigNum math and vector processing.

⁹⁰I know at least the BigNum library shipping as part of the OpenSSH already supports efficient operation on 64-bit machines. This BigNum library does not however support efficient operation on Vector Processing Units.

⁹¹Multiprocessing is the processing of data over multiple processors in parallel. Symmetric Multiprocessing refers specifically to system on which the hardware is designed for "tightly coupled" interaction between multiple processors. SMP is the most common multiprocessing architecture available today.

5 Final Thoughts

Further analysis is certainly required on the subject of parallelization in order to assure proper utilization of future architectures by both our legacy cryptographic algorithms and our more modern algorithms. What I have offered here, both in my implementation of Bitslice DES and in my brief analysis of other algorithms, clearly demonstrates that great gains are possible from effort towards such parallelization.

Section 3 already discussed at length the relation of my implementation to current and past computer science research. In this section I will instead discuss current research and its promise for efficiency on modern hardware. The major focus in current cryptographic research is the batch of algorithms submitted to the 1998 American Encryption Standard (AES) solicitation/competition. The AES competition produced a raft of new, high-quality algorithms of which one, Rijndael, was selected as the new American Encryption Standard. Rijndael is, as of 2000, now recommended by the United States government for use in all new security applications. As there has been so much focus on these new algorithms as of late, it is appropriate that I at least briefly comment here on how well these algorithms embrace modern architectures.

- Rijndael, the winner of the AES solicitation, shows some promise for intra-round parallelization, but implementations have not yet been developed to take advantage of all types of parallel architectures, including VPUs. Like all other AES submissions, Rijndael was required to work well on small 8-bit embedded-systems processors, but was not required to excel on VPUs or parallel architectures. It is likely, regardless of Rijndael's design, that due to its official status as the new AES, very efficient parallel-enabled implementations of Rijndael will surface within the near future[8, 7, 9].
- Serpent, regarded by some as the AES “runner up,” seems particularly well designed to the task of parallelization. This is likely due to the fact that one of Serpent's main contributors was Eli Biham, the father of Bitslice. Serpent shows promise of efficient

implementations on architectures with much larger words than those which we use today. Part of Serpent’s design is that it allows for both a “normal” mode and a “bitslice” mode. In its bitslice mode Serpent is scaleable to as many bits as available from the processor – thus it can function well on VPU’s in a similar fashion to Bitslice DES[21].

- RC6 shows promising performance on modern architectures for at least the short term. RC6, from my understanding of the specification, seems with a few straightforward modifications to allow for at least an efficient 64-bit implementation.⁹² RC6 however may not be easy to implement efficiently on (non SIMD) processors with word sizes any larger than 64 bits[22].
- TwoFish is yet another Feistel network and (at least from my reading of the specification) showed no more readiness for parallelization than DES or any other Feistel networks. As a Feistel network, however, much of the previous analysis applies, and TwoFish likely even has an efficient Bitslice implementation, and may even support parallel processing during decryption just like DES does[25, 23].
- MARS, the fifth finalist, is also a Feistel network and shows potential for parallelization similar to that of TwoFish[5].

From my research of the AES candidates, it is my conclusion that the AES finalists all at least show promise of efficient implementations on future hardware. This is regardless of the fact that most do not appear designed for implementation on future parallel architectures. It is likely that only once the consumer finally moves to the 64-bit multi-processor desktop computer (or there is a dramatic shift in the programmer’s mindset) that we will see very efficient implementations of these algorithms for modern parallel architectures.

⁹²In fact, distributed.net already provides a key-cracking client for RC6 which uses the AltiVec processor. I have not examined distributed.net’s code in great enough detail to comment on the relative efficiency of this SIMD implementation. I have also not yet seen an encryption library for RC6 supporting AltiVec.

Perhaps even more important than modern block ciphers is modern public key cryptography. Although PKC is still today not widely used by the consumer, I see this quickly changing in the near future. The Department of Defense is currently beginning the process of issuing the Common Access Card (a Java-based smart card⁹³) containing unique Public and Private keys for each recipient. Once enough people have their own unique public and secret key pair, public key cryptography becomes not only possible but practical. The future for Public Key Cryptography in terms of parallelization also looks bright, especially due to its complete dependence on mathematical functions for which many parallel systems were initially designed. This allows the same PKC systems to run efficiently on any platform dependent only on a lower level, but well studied, problem of how to do large number math fast.

My implementation of Bitslice was designed to show that applying minor modifications to our cryptographic algorithms can reap great rewards in terms of speed on modern parallel architectures. This study was designed initially to answer the question of how one might go about applying parallelism to current cryptography and if such is even possible. My Bitslice implementation, as an example of such parallelism in cryptography, gives both a demonstration of how and a proof of success. The results of my testing have shown that large speed gains can be accomplished through the application of parallelism in cryptography. If the reader were to take anything from the implementation offered here they should remember Bitslice's success as an example of how the power of parallelism can be exploited to give the world a faster, more modern cryptography.

Many of our present day algorithms, although not specifically designed for the hardware of tomorrow, have been found to work efficiently on current and future platforms with a few intelligent modifications. I would caution, however, that the majority of these implementations, although theoretically possible, remain only on paper and have not yet been

⁹³Smart cards were described in a footnote in Section 1. Smart cards are basically normal plastic identity cards which carry a an 8-bit microprocessor, and a small amount of very low power random access memory. Smart cards are also designed to be physically secure devices, such that the information which they contain should only be extractable with knowledge of the PIN number for that card.

implemented in software on many architectures. There is still much work for the implementors of cryptography: to move cryptographic libraries from 32-bit-dependent versions running on newer hardware to truly optimized versions for that hardware. Although we saw some awareness of architectures with the AES submissions, four of the five algorithms were not designed specifically for the computers of the future, focusing instead on the computers of today, and smaller processors such as smart cards. There is still much work to be done towards the parallelization of cryptography, but we have a promising start towards “modern” cryptography.

```

inline void
interleave128 (
    vector unsigned char stream1, vector unsigned char stream2,
    vector unsigned char table1, vector unsigned char table2,
    vector unsigned char *high, vector unsigned char *low )
{
    vector unsigned char v1, v2, v3, v4, v10, v11, v30, v31;

    v30 = ( vector unsigned char ) ( 0x04 );
    v31 = ( vector unsigned char ) ( 0xff, 0x00, 0xff, 0x00,
        0xff, 0x00, 0xff, 0x00, 0xff, 0x00, 0xff, 0x00 );

    v1 = vec_mergel ( stream1, stream1 ); // byte interleave
    v2 = vec_mergel ( stream2, stream2 ); // byte interleave
    v3 = vec_sll    ( v1, v30 ); // shift half a byte
    v4 = vec_sll    ( v2, v30 ); // shift half a byte
    v1 = vec_sel    ( v1, v3, v31 ); // the top half of each byte
    v2 = vec_sel    ( v2, v4, v31 ); // the top half of each byte
    v10 = vec_perm ( table1, table1, v1 ); // magic table lookups
    v11 = vec_perm ( table2, table2, v2 ); // magic table lookups
    *low = vec_vor ( v10, v11 ); // compose the two into a whole

    v1 = vec_sld    ( stream1, stream1, 8 );
    v2 = vec_sld    ( stream2, stream2, 8 );
    v1 = vec_mergel ( v1, v1 );
    v2 = vec_mergel ( v2, v2 );
    v3 = vec_sll    ( v1, v30 );
    v4 = vec_sll    ( v2, v30 );
    v1 = vec_sel    ( v1, v3, v31 );
    v2 = vec_sel    ( v2, v4, v31 );
    v10 = vec_perm ( table1, table1, v1 );
    v11 = vec_perm ( table2, table2, v2 );
    *high = vec_vor ( v10, v11 );
}

```

Figure 9: interleave128

$$\begin{array}{l}
v_1 = \boxed{1 \mid 0 \mid 1 \mid 1 \mid 0 \mid 0 \mid 0 \mid 1} \\
v_2 = \boxed{0 \mid 0 \mid 1 \mid 1 \mid 0 \mid 1 \mid 0 \mid 0} \\
\text{interleave8}(v_1, v_2) = \boxed{1 \mid 0 \mid 0 \mid 0 \mid 1 \mid 1 \mid 1 \mid 1 \mid 0 \mid 0 \mid 0 \mid 1 \mid 0 \mid 0 \mid 1 \mid 0} \\
v_1 = \boxed{1 \mid 0 \mid 0 \mid 0 \mid 1 \mid 1 \mid 1 \mid 1} \\
v_2 = \boxed{0 \mid 0 \mid 0 \mid 1 \mid 0 \mid 0 \mid 1 \mid 0}
\end{array}$$

Figure 10: Demonstration of 8-bit Interleave

A Computer Science Background

A.1 Alternative Number Systems

Necessary for review of this paper is the understanding of two numbering systems used heavily in computing: binary (base 2) and hexadecimal (base 16). I give a brief description of the properties of both to assure best understanding throughout the paper.

The majority of our world is based on a base 10 number system. In such a system the numbers “roll over” to the next digit-place after ten items: 00, 01, 02, 03, 04, 05, 06, 07, 08, 09 \rightarrow 10. Binary numbers are numbers in a base 2 number system and roll over after two items: 000, 001 \rightarrow 010, 011 \rightarrow 100. Binary numbers are the primary numbers for computing, since they can be recorded easily by computers as the physical presence or absence of something. In the case of the computer memory, 1s and 0s are recorded by the presence or absence of an electrical charge.

Another number system used frequently in this paper is the hexadecimal number system. This (as its name suggests) is a number system in base 16. When counting, numbers roll over to the next place after 16 items. Since we do not have 16 unique conventional digits, we use the first seven letters (A - F) of the alphabet to represent the additional digits. An example of counting to 16 (base 10) in hexadecimal numbers is the following: 00, 01, 02, 03, 04, 05, 06, 07, 08, 09, 0A, 0B, 0C, 0D, 0E, 0F \rightarrow 10. Hexadecimal numbers can be represented using either lower or upper case characters; both mean the same in this context. Hexadecimal numbers are commonly prefixed by a 0x (i.e 0x1A = 26) to identify them as such. Both of these number systems follow the same rules of addition, subtraction, and multiplication as are found in our conventional base 10 system. The difference here is that each place can represent more or fewer items before “spilling over” to the next higher place. Table 11 shows examples of counting in each system and offers conversions between the three.

Decimal	Hexadecimal	Binary	Decimal	Hexadecimal	Binary
0	0x00	00000000	9	0x09	00001001
1	0x01	00000001	10	0x0A	00001010
2	0x02	00000010	11	0x0B	00001011
3	0x03	00000011	12	0x0C	00001100
4	0x04	00000100	13	0x0D	00001101
5	0x05	00000101	14	0x0E	00001110
6	0x06	00000110	15	0x0F	00001111
7	0x07	00000111	16	0x10	00010000
8	0x08	00001000	17	0x11	00010001

Table 11: Binary, Decimal and Hexadecimal Conversion Table

A.2 Memory Storage

Important to discussions within the paper is an overview of at least some types of possible computer memory storage. Such knowledge is useful in understanding the internals of each DES and Bitslice DES as well as some of the differences between the two. I will focus here on storage in memory, as all data these algorithms will manipulate will be in memory, leaving the underlying operating system to worry about getting data from other places (such as hard drives) into memory.

When we discuss storing data, we discuss sizes of storage, and we do so in term of bits and bytes. A bit is the representation of one place in a binary number: a bit is equal to one 1 or one 0. A byte is a collection of eight of these bits (one can of course talk then of kilobytes and megabytes, but for our purposes we only care about 64 bits, or eight bytes at a time).⁹⁴ Figure 11 gives a graphical overview of bits and bytes.⁹⁵

⁹⁴I should note here that in computer science there exists a frustrating dual-usage of the words megabyte and kilobyte or megabit and kilobit. As the words were originally defined, kilo- referred to 1024 units, and mega- referred to 1024*1024 units. As these quantities were neither inline with the standard metric notation of the prefixes kilo- and mega-, or convenient for mathematical computations, common usage has modified kilobit (or -byte) and megabit (or -byte) to refer to 1000, or 1000 * 1000 units respectively. This can cause confusion when reading statistics like those you will find in this paper. For my usage here, I have remained with the original definitions (kilo 1024 units, mega = 1024*1024 units), which may make my mega- and kilo-based numbers smaller than if the exact same results were presented in other papers.

⁹⁵Regarding the numbering used in Figure 11: the “low bit” is the bit representing 2^0 . In this table the common numbering scheme (which is opposite the one used throughout much my paper) begins by numbering the low-bit on the far right at 0. I have numbered the bits here starting with 1, but it is equally (if not more) common to number them starting with 0 to correspond to the powers of two which they represent. I have

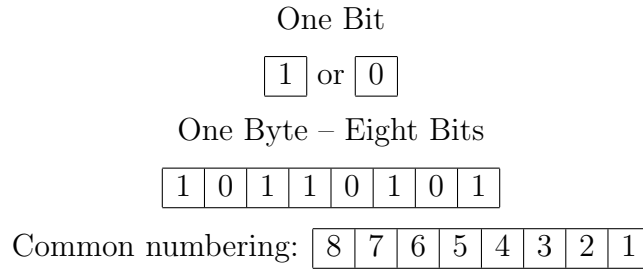


Figure 11: Bits and Bytes

To access this storage of bits and bytes the computer assigns addresses to each location in memory; specifically, the computer assigns a unique physical address to every eight bits, or one address per every byte of memory. This memory addressing scheme is called byte-addressing storage and is used in all modern computers. This addressing scheme can become an issue for algorithms like those described in this paper where the majority of the work is done at the bit level. In order to access the individual bits of any byte, implementers have to do special manipulating.

A.3 A Little Logic

In order to completely understand the discussions of cryptographic algorithms and especially the implementation of Bitslice DES, it is essential that the reader know at least a little about the discipline of mathematical logic. Particularly, in this paper we care about Boolean logic – logical operations performed on the set of numbers 0 and 1.

In mathematical logic, we commonly ask the question “is that statement true?” To answer that question perform the operations listed in that statement and arrive at a result representing either truth or falsehood of that original statement. In Boolean logic we represent truth by the value 1, and falsehood by the value 0. In computers we also use a single bit with a value of 1 to represent truth and a 0 to represent falsehood.⁹⁶

made both of these choices for the ease of reading, deviating with what would be a more direct correlation with the computer hardware.

⁹⁶Since, as described in Appendix A.2 computers use byte-addressing, and do not normally operate with data on a bit-level, commonly blocks of data larger than one bit are said to represent falsehood only if all

The language we will use to express these statements of mathematical logic will consist only of 1's (TRUE) or 0's (FALSE), a few other simple operators, OR, AND, XOR, and a few more complex operators NOT, ANDC, NAND and NOR. We will define these statements in terms of their “truth tables.” Truth tables define operations by telling us that given which two inputs, the function returns what value. The truth tables we give here can be used by taking a given statement (e.g. “1 OR 0”) and finding the first value in the left most column and the second value in the first row; the cell intersected by the row of the first value and the column of the second is the answer. The truth tables are given in Table 12.

AND (&)	<table style="border-collapse: collapse; text-align: center;"> <tr><td style="border: 1px solid black; width: 15px; height: 15px;"></td><td style="border: 1px solid black; width: 15px; height: 15px;">1</td><td style="border: 1px solid black; width: 15px; height: 15px;">0</td></tr> <tr><td style="border: 1px solid black; width: 15px; height: 15px;">1</td><td style="border: 1px solid black; width: 15px; height: 15px;">1</td><td style="border: 1px solid black; width: 15px; height: 15px;">0</td></tr> <tr><td style="border: 1px solid black; width: 15px; height: 15px;">0</td><td style="border: 1px solid black; width: 15px; height: 15px;">0</td><td style="border: 1px solid black; width: 15px; height: 15px;">0</td></tr> </table>		1	0	1	1	0	0	0	0	OR ()	<table style="border-collapse: collapse; text-align: center;"> <tr><td style="border: 1px solid black; width: 15px; height: 15px;"></td><td style="border: 1px solid black; width: 15px; height: 15px;">1</td><td style="border: 1px solid black; width: 15px; height: 15px;">0</td></tr> <tr><td style="border: 1px solid black; width: 15px; height: 15px;">1</td><td style="border: 1px solid black; width: 15px; height: 15px;">1</td><td style="border: 1px solid black; width: 15px; height: 15px;">1</td></tr> <tr><td style="border: 1px solid black; width: 15px; height: 15px;">0</td><td style="border: 1px solid black; width: 15px; height: 15px;">1</td><td style="border: 1px solid black; width: 15px; height: 15px;">0</td></tr> </table>		1	0	1	1	1	0	1	0	XOR (\oplus)	<table style="border-collapse: collapse; text-align: center;"> <tr><td style="border: 1px solid black; width: 15px; height: 15px;"></td><td style="border: 1px solid black; width: 15px; height: 15px;">1</td><td style="border: 1px solid black; width: 15px; height: 15px;">0</td></tr> <tr><td style="border: 1px solid black; width: 15px; height: 15px;">1</td><td style="border: 1px solid black; width: 15px; height: 15px;">0</td><td style="border: 1px solid black; width: 15px; height: 15px;">1</td></tr> <tr><td style="border: 1px solid black; width: 15px; height: 15px;">0</td><td style="border: 1px solid black; width: 15px; height: 15px;">1</td><td style="border: 1px solid black; width: 15px; height: 15px;">0</td></tr> </table>		1	0	1	0	1	0	1	0
	1	0																														
1	1	0																														
0	0	0																														
	1	0																														
1	1	1																														
0	1	0																														
	1	0																														
1	0	1																														
0	1	0																														
ANDC	<table style="border-collapse: collapse; text-align: center;"> <tr><td style="border: 1px solid black; width: 15px; height: 15px;"></td><td style="border: 1px solid black; width: 15px; height: 15px;">1</td><td style="border: 1px solid black; width: 15px; height: 15px;">0</td></tr> <tr><td style="border: 1px solid black; width: 15px; height: 15px;">1</td><td style="border: 1px solid black; width: 15px; height: 15px;">0</td><td style="border: 1px solid black; width: 15px; height: 15px;">1</td></tr> <tr><td style="border: 1px solid black; width: 15px; height: 15px;">0</td><td style="border: 1px solid black; width: 15px; height: 15px;">0</td><td style="border: 1px solid black; width: 15px; height: 15px;">0</td></tr> </table>		1	0	1	0	1	0	0	0	NOR	<table style="border-collapse: collapse; text-align: center;"> <tr><td style="border: 1px solid black; width: 15px; height: 15px;"></td><td style="border: 1px solid black; width: 15px; height: 15px;">1</td><td style="border: 1px solid black; width: 15px; height: 15px;">0</td></tr> <tr><td style="border: 1px solid black; width: 15px; height: 15px;">1</td><td style="border: 1px solid black; width: 15px; height: 15px;">0</td><td style="border: 1px solid black; width: 15px; height: 15px;">0</td></tr> <tr><td style="border: 1px solid black; width: 15px; height: 15px;">0</td><td style="border: 1px solid black; width: 15px; height: 15px;">0</td><td style="border: 1px solid black; width: 15px; height: 15px;">1</td></tr> </table>		1	0	1	0	0	0	0	1	NAND	<table style="border-collapse: collapse; text-align: center;"> <tr><td style="border: 1px solid black; width: 15px; height: 15px;"></td><td style="border: 1px solid black; width: 15px; height: 15px;">1</td><td style="border: 1px solid black; width: 15px; height: 15px;">0</td></tr> <tr><td style="border: 1px solid black; width: 15px; height: 15px;">1</td><td style="border: 1px solid black; width: 15px; height: 15px;">0</td><td style="border: 1px solid black; width: 15px; height: 15px;">1</td></tr> <tr><td style="border: 1px solid black; width: 15px; height: 15px;">0</td><td style="border: 1px solid black; width: 15px; height: 15px;">1</td><td style="border: 1px solid black; width: 15px; height: 15px;">1</td></tr> </table>		1	0	1	0	1	0	1	1
	1	0																														
1	0	1																														
0	0	0																														
	1	0																														
1	0	0																														
0	0	1																														
	1	0																														
1	0	1																														
0	1	1																														

Table 12: Logical operators

If we wished to use one of these truth tables, to find the value of the statement “1 OR 0” we would find the value the column which represented the value for 1, and the row which represented the value for 0. Finding then the intersection of these two, we would see that “1 OR 0” is defined as being “1” or TRUE. Thus the statement “1 OR 0” is said to be equivalent to the statement “1” under boolean logic. It will be necessary throughout the reading of this paper that you at least be familiar with the operations listed in Table 12 at least enough to know that they are boolean logic operators, however you need not know what exactly they mean or have memorized their truth tables.

One additional example which will be useful in understanding the paper, is to know that in terms of computers operations such as “OR” can apply to any equal-sized groups of bits. For example, the statement “1001 OR 0010” would under boolean logic resolve to bits in that block are equal to 0, otherwise the block is to represent truth. In practical terms this means that the number 143 represents truth, as does the number 34,522 and any other number not equal to 0.

the bit group “1011.” This is in essence applying the same logical operator in parallel across multiple bit pairs.

A final note regarding logic as used in electronic devices is also necessary here. The truth tables given here are implemented in electronic devices in what are called “logic gates.” Logic gates most often work exactly like the tables given here. An OR logic gate, having two inputs, when the first input receives an electrical charge (TRUE) and the second does not (FALSE), the output of the gate is charged (TRUE). Logic gates however, can differ from the simplified operations presented above by taking inputs from more than two sources. For example a five input logical OR gate might take $\{0,0,1,0,1\}$ and output 1. In the technique described for Bitslice DES, one takes the “logic gate” implementation of DES (which may use logic gate accepting more than just two inputs) and first converts this logic gate DES to a version consisting only of two-input gates. After that, then one can implement this hardware version of DES as Bitslice DES in software using the the bitwise logic functions already implemented on modern computers.

B Package Listing & Descriptions

A short listing of the most important files in my Bitslice implementation. Those essential to the function of the algorithm (and to the understanding of what I have done) are listed in source form in Appendix C.

top level:

README	brief description regarding usage of this source, etc.
bitslice	generated Mach-O executable file for the PowerPC G4
bitslice.c	generated encryption/decryption code
bitslice.h	testing data and function definitions
kwan.c	Matthew Kwan's original S-Boxes
libdes.a	binary version of the libdes encryption library
main.c	main source file for Bitslice
sboxes.h	s-box definitions
swizzle_iu.c	integer unit swizzling code
swizzle_vpu.c	vector unit swizzling code
test_bs_speed.c	speed test functions for Bitslice execution
times.txt	various recorded notes and execution times for Bitslice

./tools:

altivec_sboxes.c.pl	generates altivec enabled s-boxes
eval2encrypt.pl	converts kwan's des_eval code into basic encryption code
generate_bitslice_des.pl	generates Bitslice encryption/decryption code
generate_bs_speed_tests.pl	generates Bitslice speed testing code
generate_sboxes_h.pl	generates s-boxes header file
generate_swizzle_speed_tests.pl	generates swizzle speed testing code
generate_swizzle_vpu.c.pl	generates vpu swizzling code
swap_endian_bitslice.c.pl	reverses bit order for Kwan's evaluation code

C Source Code

In an effort to provide Lawrence with a lasting record of my Bitslice work, I have provided the following appendix containing at least the most important files of my Bitslice source code. In order to compile and run the actual Bitslice application would require generation of a substantial number of additional C source files using the perl scripts provided here. The appendix consists primarily of the hand written sources for my project and the scripts used to produce the auto-generated source files. There are however a few excerpts from generated files provided for clarity.

C.1 generate_bitslice_des.pl

This is a perl script which I wrote to generate the necessary bs_encrypt and bs_decrypt functions for both integer and AltiVec units.

```
#!/usr/bin/perl
# $id$

# generate_bitslice_des.pl
#
# This file is used to generate all the encryption/decryption
# code used in Eric Seidel's AltiVec Bitslice implmentation
#
# This script was originally designed to produce code similar
# to the generated bitslice code destributed by Matthew Kwan
#
# Eric Seidel, 2003

# To generate code compatible with Matthew Kwan's bitslice
# you should set options to the following:
# $keyreduce = 1;
# $little = 1;

# To generate code compatible with the rest of my implmentation
# it is suggested you use the following options:
# $keyreduce = 0;
# $little = 0;
```

```

# set this value to 1 to generate code expecting 56 bit
# key matrices instead of 64 bits.
$keyreduce = 0;

# set this value to 1 to generate little-endian code.
# this is NOT necessary for little endian machines,
# just makes little endian people feel better.
$little = 0;

# set this value to 1 to generate code for both
# the integer unit (iu) and the vector processing unit (vpu)
# otherwise referred to as the altivec unit (G4)
# a value of 0 will generate ONLY integer unit code.
$vpu_enabled = 1;

$header_printed = 0;

foreach $altivec (0..$vpu_enabled) {

foreach $encrypt (1,0) {

if (!$header_printed) {
    # only print the header once.
    $header_printed++;

    # Print out the Header info for each Encrypt/Decrypt file.
    print "/*\n";
    print " * Auto-Generated bitslice code.\n";
    print " * This code was modeled after code generated by ",
        "Matthew Kwan, 1998.\n";
    print " * The perl script generate_bitslice_des.pl was ",
        "written by\n";
    print " * Eric Seidel, 2002.\n";
    print " */\n\n";

    print "/*\n";
    print " * Generating Options:\n";
    print " * Key Size: ", ($keyreduce?"56 (reduced)":"64 (normal)", "\n";
    print " * Endian Choice: ",
        ($little?"LITTLE-ENDIAN":"BIG-ENDIAN"), "\n";
    #print " * Processing Unit: ",
    #    ($altivec?"AltiVec 128-bit Vector Unit":
    #    "32-bit Integer Unit"), "\n";
    print " */\n\n";
}
}
}

```

```

print "#include \"kwan.c\"\\n";

if ($vpu_enabled) {
    print "#include \"sboxes_std_altivec.c\"\\n";
}
print "\\n\\n";
}

# Set the "vector" datatype based on AltiVec or not.
# also set the call suffix at this time.
if ($altivec) {
    $datatype = "vector unsigned int";
    $suf = "_vpu";
} else {
    $datatype = "unsigned long";
    $suf = "_iu";
}

#####
# Sub-Routines #
#####

# prints pretty matrices
sub pretty {
    @_ [0] =~ s/((([^\n]+,){7})\s*/$1\n\t/g;
    return("\t" . @_ [0]);
}

# prints pretty sbox calls
sub spretty {
    @_ [0] =~ s/((([^\n]+,){4})\s*/$1\n\t\t/g;
    return("\t" . @_ [0]);
}

#####
# Key-Scheduling #
#####

# initial key permutation
@pPC1=( 57, 49, 41, 33, 25, 17, 9,
        1, 58, 50, 42, 34, 26, 18,
        10, 2, 59, 51, 43, 35, 27,
        19, 11, 3, 60, 52, 44, 36,
        63, 55, 47, 39, 31, 23, 15,

```

```

    7, 62, 54, 46, 38, 30, 22,
    14, 6, 61, 53, 45, 37, 29,
    21, 13, 5, 28, 20, 12, 4);

#final subkey permutation
@PC2=( 14, 17, 11, 24, 1, 5,
      3, 28, 15, 6, 21, 10,
      23, 19, 12, 4, 26, 8,
      16, 7, 27, 20, 13, 2,
      41, 52, 31, 37, 47, 55,
      30, 40, 51, 45, 33, 48,
      44, 49, 39, 56, 34, 53,
      46, 42, 50, 36, 29, 32);

if ($keyreduce)
{
    #shrink matrix, shrinks the 64 bit array to a 56 bit array.
    # used for compatibility with
    # Matthew Kwan's bitslice implementation.
    @shr = ( 0, 1, 2, 3, 4, 5, 6, 0,
            7, 8, 9, 10, 11, 12, 13, 0,
            14, 15, 16, 17, 18, 19, 20, 0,
            21, 22, 23, 24, 25, 26, 27, 0,
            28, 29, 30, 31, 32, 33, 34, 0,
            35, 36, 37, 38, 39, 40, 41, 0,
            42, 43, 44, 45, 46, 47, 48, 0,
            49, 50, 51, 52, 53, 54, 55, 0);

    # apply the shrink matrix to shrink it to 56 bit keys
    # for easy comparison with kwan's code
    @PC1 = ();
    while(@pPC1) {
        $i = shift(@pPC1);
        #print "i:\t", $i, "\t\t$joined:\t",$joined[$i-1], "\n";
        push(@PC1, $shr[$i-1]);}
} else {
    # no reduction necessary.
    @PC1 = @pPC1;
}

@lastC = @PC1;
#print "Just checking: ", ($#lastC + 1) / 2, "\n\n";
@lastD = splice(@lastC, ($#lastC + 1) / 2);

```

```

# Sub-Key Shifts
# These are used as left shift counts when generating
# the individual sub-key halves Cn, Dn
# 0 is ignored
# 1, 2, 9, 16 = 1, all others = 2
@shifts = ( 0, 1, 1, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 1 );

# Let the 0th subkey be empty.
$K[0] = "";

# For printing debug information.
#print "\@K0= {" ,pretty(join(' ', @PC1)),"};\n\n";
#print "\@PC2= {" ,pretty(join(' ', @PC2)),"};\n";

#print "\@C0= {" ,pretty(join(' ', @lastC)),"};\n";
#print "\@D0= {" ,pretty(join(' ', @lastD)),"};\n";

# Now we iterate through 1..16 generating each of the sub-keys

foreach (1..16) {
    # To do so, we copy the bits of the previous subkeys halves
    # and then apply a left shift as designated by the
    # shift matrix.
    # e.g. for C1, D1, we apply a $shifts[1] = 1 bit shift from C0, D0

    @C = @lastC; # copy previous bits
    foreach $i (1..$shifts[$_]){
        push(@C, shift(@C)); # shift them left
    }
    @lastC = @C;
    #print "\@C",$_,"= {" ,pretty(join(' ', @C)),"};\n";

    @D = @lastD; # copy previous bits
    foreach $i (1..$shifts[$_]){
        push(@D, shift(@D)); # shift them left
    }
    @lastD = @D;
    #print "\@D",$_,"= {" ,pretty(join(' ', @D)),"};\n";

    # Now we re-join Cn, Dn and apply the final permutation PC2
    @temp = @PC2;
    @joined = (@C,@D);
    @KN = ();
    while(@temp) {
        $i = shift(@temp);

```

```

    #print "i:\t", $i, "\t\t$joined:\t",$joined[$i-1], "\n";
    if ($little) {
        if ($keyreduce) {
            push(@KN, 55 - $joined[$i-1]);
        } else {
            push(@KN, 63 - $joined[$i-1]);
        }
    } else {
        push(@KN, $joined[$i-1] - 1);
    }
}

# Store the sub-key away for later.
@K[$_] = join(' ', @KN);

# Print a nice copy of the subkey -- Debug info
#print "\@K",$_,"= {" ,pretty(join(' ', @KN)),"};\n\n"
    #if ($_ == 1); # only print the first subkey to save space.
}

# Done with subkeys... on to the message itself

#####
# Message Processing #
#####

# initial message perumation:
@IP=( 58, 50, 42, 34, 26, 18, 10, 2,
      60, 52, 44, 36, 28, 20, 12, 4,
      62, 54, 46, 38, 30, 22, 14, 6,
      64, 56, 48, 40, 32, 24, 16, 8,
      57, 49, 41, 33, 25, 17, 9, 1,
      59, 51, 43, 35, 27, 19, 11, 3,
      61, 53, 45, 37, 29, 21, 13, 5,
      63, 55, 47, 39, 31, 23, 15, 7);

# selection table, expands Rn to S blocks
@E=( 32, 1, 2, 3, 4, 5,
     4, 5, 6, 7, 8, 9,
     8, 9, 10, 11, 12, 13,
     12, 13, 14, 15, 16, 17,
     16, 17, 18, 19, 20, 21,
     20, 21, 22, 23, 24, 25,
     24, 25, 26, 27, 28, 29,
     28, 29, 30, 31, 32, 1);

```

```

# final S-Box permutation matrix
# applied to the conjoined results of
# the 8 s-box applications.
# NOTE: this is 1 based!
@P=( 16, 7, 20, 21,
      29, 12, 28, 17,
      1, 15, 23, 26,
      5, 18, 31, 10,
      2, 8, 24, 14,
      32, 27, 3, 9,
      19, 13, 30, 6,
      22, 11, 4, 25);

# but since we wish to use P to determine the OUTPUTS
# of the S-Box function and not on the actual bits
# returned from the function, we need to use P's inverse:
# NOTE: this one is 0 based!
@Pinv=( 8, 16, 22, 30,
        12, 27, 1, 17,
        23, 15, 29, 5,
        25, 19, 9, 0,
        7, 13, 24, 2,
        3, 28, 10, 18,
        31, 11, 21, 6,
        4, 26, 14, 20);

# To generate (a 0-based) Pinv use:
#@NP = ();
#foreach $i (0 .. $#P) {
#    @NP[$P[$i]-1] = $i;
#}
#print spretty(join(', ', @NP));

# chop the message into two 32-bit blocks

# First we need to reverse the message if we are little-endian
if ($little) {
    @lastL = reverse(@IP);
} else {
    @lastL = @IP;
}
# Then we chop it in half.
@lastR = splice(@lastL, 32);

```

```

# Print out the function header information
# (for easy reading of the generated source).

print "/*\n";
print " * This function runs one round of ",
      $encrypt?"encryption":"decryption", " over\n";
if ($encrypt) {
    print " * the plain-text bits p[0] .. p[63] to produce\n";
    print " * the cypher-text bits c[0] .. c[63] using the \n";
} else {
    print " * the cypher-text bits c[0] .. p[63] to produce\n";
    print " * the plain-text bits c[0] .. c[63] using the \n";
}
if ($keyreduce) {
print " * the key bits k[0] .. k[55].\n";
} else {
print " * the key bits k[0] .. k[63].\n";
}
print " */\n";

# Print encrypt/decrypt function name.
if ($encrypt) {
    print "void bs_des_encrypt$suf(\n";
    print "\tconst ", $datatype, " *p,\n";
    print "\t", $datatype, " *c,\n";
} else {
    print "void bs_des_decrypt$suf(\n";
    print "\tconst ", $datatype, " *c,\n";
    print "\t", $datatype, " *p,\n";
}
print "\tconst ", $datatype, " *k\n";
print ") {\n";

# If we are either key-reducing or little-endian, we need to warn
# the user.
if ($keyreduce || $little)
{
    print "\t// ****NOTE****\n";
}

# Warn them about little endian code.
if ($little)

```

```

{
    print "\t// If you are comparing this to the DES ",
          "specification you will find\n";
    print "\t// that our arrays organize the bits in the ",
          "reverse order to what you\n";
    print "\t// assume in a \"little endian\" fashion with ",
          "the first bit being far right.\n";
    print "\t// What the DES specification refers to as the ",
          "1st bit is\n";
    print "\t// actually bit 63 (or 55) in our arrays. This ",
          "was a design decision made\n";
    print "\t// in order to better allow the author to compare ",
          "his bitslice DES\n";
    print "\t// implementation to the bitslice implementation ",
          "provided by Matthew Kwan.\n";
    print "\t// There are options in the perl script which ",
          "allow you to generate\n";
    print "\t// a more \"specification friendly\" version of ",
          "DES if you so desire.\n\n";
}

# Warn them about 56-bit key code.
if ($keyreduce)
{
    print "\t// YOU SHOULD ALSO NOTE that currently keys are ",
          "assumed to have 56 bits\n";
    print "\t// using options in the perl script you can also ",
          "change that to 64 bits if desired.\n\n";
}

# More code comments.
print "\n\t// First we load all our plain-text into local ",
      "variables (hopefully registers).\n";
print "\t// We know however that the AltiVec unit on modern ",
      "G4s only has 32 registers thus\n";
print "\t// we naively hope that the compiler can optimize away ",
      "at least 32 of these variables.\n";
print "\t// We may have to add many load hints at a later time ",
      "to make this run efficiently.\n";
print "\t// Notice that we are applying the Initial ",
      "Permutation (IP) matrix as we do this.\n";

if ($encrypt) {
    $a = "p";
} else {

```

```

    $a = "c";
}

# Print out the actual C assignment statements.
foreach (0 .. $#lastL) {
    # print out the C variable assignment
    print "\t", $datatype, "\t1", $_, " = ", $a,
          "[", ($lastL[$_]-1), "];\n";
    # and change the internal perl assignment
    @lastL[$_] = "l$_";
}

foreach (0 .. $#lastR) {
    # print out the C variable assignment
    print "\t", $datatype, "\tr", $_, " = ", $a,
          "[", ($lastR[$_]-1), "];\n";
    # and change the internal perl assignment
    @lastR[$_] = "r$_";
}

print "\n";

print "\t// Now we go through 16 rounds of S-box ",
      "applications, XORing the correct subkey\n";
print "\t// with the text at each stage, and then storing ",
      "the results in vector for the next S-box.\n";
print "\t// Notice how with bitslice (because we effectively ",
      "have direct bit-addressing)\n";
print "\t// we can completely ignore key scheduling, as it is ",
      "done for us.\n";
print "\t// If you read the generation script, you will see ",
      "that it pre-calculates the\n";
print "\t// necessary sub-key lookups so we don't have to ",
      "bother with subkeys here.\n";
print "\n";

# Ah... finally we print out the actual 16 rounds of S-Box calls.
foreach (1..16) {

    # We use two arrays here to do the generation:
    # pre-in and pre-out.
    # pre-in are the variables which will be used for
    # INPUT into the s-boxes
    # pre-out are the variables which will be used for
    # OUTPUT from the s-boxes

```

```

# both undergo some processing before being used directly.

print "\t// S-Box round #$_\n";

# Load up this round's sub-key.
if ($encrypt) { # normal key schedule
    @subkey = split(' ', $K[$_]);
} else { # reverse key schedule.
    @subkey = split(' ', $K[17 - $_]);
}

# Is this a R = R or L = R round? (i.e. odd or even)
# load accordingly into the pre-in and pre-out vectors
if ( $_ % 2) { # odd round
    @prein = @lastR;
    @preout = @lastL;
} else { # even round
    @prein = @lastL;
    @preout = @lastR;
}

# Apply the selection/expansion table @E
# to the pre-in array to form the in array
@in = ();
@temp = @E;
while(@temp) {
    $j = shift(@temp);
    # in is the expanded R(n-1)
    push(@in, $prein[$j-1]);}

#print "\@in",$_,"= {" ,pretty(join(' ', @in)),"};\n\n"
#    if ($_ == 1);

# Apply the inverse of the permutation matrix @P (@Pinv)
# to the preout array to form the out array
@out = ();
@temp = @Pinv;
while(@temp) {
    $j = shift(@temp);
    # out is the permuted bits of preout
    # as will be used for the assignments
    # of the s-box results.
    # NOTE how this is 0-based!
    push(@out, $preout[$j]);}

```

```

#print "@out",$_,"= {" ,pretty(join(', ', @out)),"};\n\n"
# if ($_ == 1);

foreach $i (0..7) {
    # take the first six bits of in as the next block
    @B = splice(@in, 0, 6);

    # generate the list of in variables
    @ins = ();
    foreach $q (0..5) {
        if ($altivec) {
            @ins[$q] =
                "vec_xor($B[$q], k[" .
                $subkey[6 * $i + $q] . "])";
        } else {
            @ins[$q] =
                "$B[$q] ^ k[" . $subkey[6 * $i + $q] . "];"
        }
    }

    # generate the list of out variables
    @outs = ();
    foreach $q (0..3) {
        @outs[$q] = "&" . $out[4*$i + $q];}

    # Finally, print the entire S-box function call
    print "\ts", $i+1, "$suf( ",
        spretty(join(", ", @ins, @outs)), ");\n";
}
print "\n";
}

# now we finally print out the cypher-text assignments
# we combine L(16)R(16) as RL
# and perform the final permutation FP

# Final message permutation:
# also commonly referred to as IP(-1).
@FP=( 40, 8, 48, 16, 56, 24, 64, 32,
      39, 7, 47, 15, 55, 23, 63, 31,
      38, 6, 46, 14, 54, 22, 62, 30,
      37, 5, 45, 13, 53, 21, 61, 29,
      36, 4, 44, 12, 52, 20, 60, 28,
      35, 3, 43, 11, 51, 19, 59, 27,
      34, 2, 42, 10, 50, 18, 58, 26,

```

```

        33, 1, 41, 9, 49, 17, 57, 25);

print "\t// Now we perform the final permutation IP-1 as ",
      "we assign the\n";
print "\t// computed vectors to their appropriate cypher ",
      "text slots.\n";

# join the two halves
if ($little) {
    @joined = reverse(@lastR, @lastL);
} else {
    @joined = (@lastR, @lastL);
}
#print "joined =", pretty(join(' ', @joined));

# apply the final permutation
# and print out the final permuted assignments.
@temp = @FP;
if ($encrypt) {
    $a = "c";
} else {
    $a = "p";
}
foreach (0..$#FP) {
    print "\t", $a, "[$_]\t= ", $joined[shift(@temp)-1] ,";\n"
}

print "}\n\n\n";

} # foreach $encrypt

} # foreach $altivec

```

C.2 Excerpts from bitslice.c

This is an excerpt from the source-code generated by generate_bitslice_des.pl for AltiVec enabled encryption. Decryption looks almost identical, only with a reversed key-schedule. IU Bitslice DES code is also nearly identical, with different logical operators and data types.

```
/*
 * Auto-Generated bitslice code.
 * This code was modeled after code generated by Matthew Kwan, 1998.
 * The perl script generate_bitslice_des.pl was written by
 * Eric Seidel, 2002.
 */

/*
 * Generating Options:
 * Key Size: 64 (normal)
 * Endian Choice: BIG-ENDIAN
 */

#include "kwan.c"
#include "sboxes_std_altivec.c"

#define USE_PREFETCH 1

/*
 * This function runs one round of encryption over
 * the plain-text bits p[0] .. p[63] to produce
 * the cypher-text bits c[0] .. c[63] using the
 * the key bits k[0] .. k[63].
 */
void bs_des_encrypt_vpu(
const vector unsigned int *p,
vector unsigned int *c,
const vector unsigned int *k
) {

#if USE_PREFETCH
    // mark the incoming data is about to be read from.
    // address, special value, identifier
    // 1 vector per unit 00001
    // 64 units total 01000000
    // spaced at 16 bytes each. 00000000000010000
    // with 3 leading bits and spacers:

```

```

        // 0000 0001 0100 0000 0000 0000 0001 0000
        // in hex: 0x01100010
        vec_dst(p, 0x01800010, 0);
#endif

// First we load all our plain-text into local
// variables (hopefully registers).
// We know however that the AltiVec unit on modern
// G4s only has 32 registers thus
// we naively hope that the compiler can optimize away
// at least 32 of these variables.
// We may have to add many load hints at a later time
// to make this run efficiently.
// Notice that we are applying the Initial Permutation
// (IP) matrix as we do this.
vector unsigned int l0 = p[57];
vector unsigned int l1 = p[49];
vector unsigned int l2 = p[41];
vector unsigned int l3 = p[33];
vector unsigned int l4 = p[25];
vector unsigned int l5 = p[17];
vector unsigned int l6 = p[9];
vector unsigned int l7 = p[1];
vector unsigned int l8 = p[59];
vector unsigned int l9 = p[51];
vector unsigned int l10 = p[43];
vector unsigned int l11 = p[35];
vector unsigned int l12 = p[27];
vector unsigned int l13 = p[19];
vector unsigned int l14 = p[11];
vector unsigned int l15 = p[3];
vector unsigned int l16 = p[61];
vector unsigned int l17 = p[53];
vector unsigned int l18 = p[45];
vector unsigned int l19 = p[37];
vector unsigned int l20 = p[29];
vector unsigned int l21 = p[21];
vector unsigned int l22 = p[13];
vector unsigned int l23 = p[5];
vector unsigned int l24 = p[63];
vector unsigned int l25 = p[55];
vector unsigned int l26 = p[47];
vector unsigned int l27 = p[39];
vector unsigned int l28 = p[31];
vector unsigned int l29 = p[23];

```

```

vector unsigned int l30 = p[15];
vector unsigned int l31 = p[7];
vector unsigned int r0 = p[56];
vector unsigned int r1 = p[48];
vector unsigned int r2 = p[40];
vector unsigned int r3 = p[32];
vector unsigned int r4 = p[24];
vector unsigned int r5 = p[16];
vector unsigned int r6 = p[8];
vector unsigned int r7 = p[0];
vector unsigned int r8 = p[58];
vector unsigned int r9 = p[50];
vector unsigned int r10 = p[42];
vector unsigned int r11 = p[34];
vector unsigned int r12 = p[26];
vector unsigned int r13 = p[18];
vector unsigned int r14 = p[10];
vector unsigned int r15 = p[2];
vector unsigned int r16 = p[60];
vector unsigned int r17 = p[52];
vector unsigned int r18 = p[44];
vector unsigned int r19 = p[36];
vector unsigned int r20 = p[28];
vector unsigned int r21 = p[20];
vector unsigned int r22 = p[12];
vector unsigned int r23 = p[4];
vector unsigned int r24 = p[62];
vector unsigned int r25 = p[54];
vector unsigned int r26 = p[46];
vector unsigned int r27 = p[38];
vector unsigned int r28 = p[30];
vector unsigned int r29 = p[22];
vector unsigned int r30 = p[14];
vector unsigned int r31 = p[6];

#if USE_PREFETCH
    // mark the incoming data is about to be read from.
    // address, special value, identifier
    // 1 vector per unit 00001
    // 64 units total 01000000
    // spaced at 16 bytes each. 0000000000010000
    // with 3 leading bits and spacers:
    // 0000 0001 0100 0000 0000 0000 0001 0000
    // in hex: 0x01100010
    vec_dst(k, 0x01800010, 1);

```

```

#endif

// Now we go through 16 rounds of S-box applications,
// XORing the correct subkey with the text at each stage,
// and then storing the results in vector for the next S-box.
// Notice how with bitslice (because we effectively have
// direct bit-addressing) we can completely ignore key
// scheduling, as it is done for us.
// If you read the generation script, you will see that
// it pre-calculates the necessary sub-key lookups so we
// don't have to bother with subkeys here.

// S-Box round #1
s1_vpu( vec_xor(r31, k[9]), vec_xor(r0, k[50]),
vec_xor(r1, k[33]), vec_xor(r2, k[59]),
vec_xor(r3, k[48]), vec_xor(r4, k[16]),
&l18, &l116, &l122, &l130);
s2_vpu( vec_xor(r3, k[32]), vec_xor(r4, k[56]),
vec_xor(r5, k[1]), vec_xor(r6, k[8]),
vec_xor(r7, k[18]), vec_xor(r8, k[41]),
&l112, &l127, &l11, &l117);
s3_vpu( vec_xor(r7, k[2]), vec_xor(r8, k[34]),
vec_xor(r9, k[25]), vec_xor(r10, k[24]),
vec_xor(r11, k[43]), vec_xor(r12, k[57]),
&l123, &l115, &l129, &l15);
s4_vpu( vec_xor(r11, k[58]), vec_xor(r12, k[0]),
vec_xor(r13, k[35]), vec_xor(r14, k[26]),
vec_xor(r15, k[17]), vec_xor(r16, k[40]),
&l125, &l119, &l19, &l10);
s5_vpu( vec_xor(r15, k[21]), vec_xor(r16, k[27]),
vec_xor(r17, k[38]), vec_xor(r18, k[53]),
vec_xor(r19, k[36]), vec_xor(r20, k[3]),
&l17, &l113, &l124, &l12);
s6_vpu( vec_xor(r19, k[46]), vec_xor(r20, k[29]),
vec_xor(r21, k[4]), vec_xor(r22, k[52]),
vec_xor(r23, k[22]), vec_xor(r24, k[28]),
&l13, &l128, &l110, &l118);
s7_vpu( vec_xor(r23, k[60]), vec_xor(r24, k[20]),
vec_xor(r25, k[37]), vec_xor(r26, k[62]),
vec_xor(r27, k[14]), vec_xor(r28, k[19]),
&l131, &l111, &l121, &l16);
s8_vpu( vec_xor(r27, k[44]), vec_xor(r28, k[13]),
vec_xor(r29, k[12]), vec_xor(r30, k[61]),
vec_xor(r31, k[54]), vec_xor(r0, k[30]),
&l14, &l126, &l114, &l120);

```

```

.
.
.

// S-Box round #16
s1_vpu( vec_xor(131, k[17]), vec_xor(10, k[58]),
vec_xor(11, k[41]), vec_xor(12, k[2]),
vec_xor(13, k[56]), vec_xor(14, k[24]),
&r8, &r16, &r22, &r30);
s2_vpu( vec_xor(13, k[40]), vec_xor(14, k[35]),
vec_xor(15, k[9]), vec_xor(16, k[16]),
vec_xor(17, k[26]), vec_xor(18, k[49]),
&r12, &r27, &r1, &r17);
s3_vpu( vec_xor(17, k[10]), vec_xor(18, k[42]),
vec_xor(19, k[33]), vec_xor(110, k[32]),
vec_xor(111, k[51]), vec_xor(112, k[0]),
&r23, &r15, &r29, &r5);
s4_vpu( vec_xor(111, k[1]), vec_xor(112, k[8]),
vec_xor(113, k[43]), vec_xor(114, k[34]),
vec_xor(115, k[25]), vec_xor(116, k[48]),
&r25, &r19, &r9, &r0);
s5_vpu( vec_xor(115, k[29]), vec_xor(116, k[4]),
vec_xor(117, k[46]), vec_xor(118, k[61]),
vec_xor(119, k[44]), vec_xor(120, k[11]),
&r7, &r13, &r24, &r2);
s6_vpu( vec_xor(119, k[54]), vec_xor(120, k[37]),
vec_xor(121, k[12]), vec_xor(122, k[60]),
vec_xor(123, k[30]), vec_xor(124, k[36]),
&r3, &r28, &r10, &r18);
s7_vpu( vec_xor(123, k[5]), vec_xor(124, k[28]),
vec_xor(125, k[45]), vec_xor(126, k[3]),
vec_xor(127, k[22]), vec_xor(128, k[27]),
&r31, &r11, &r21, &r6);
s8_vpu( vec_xor(127, k[52]), vec_xor(128, k[21]),
vec_xor(129, k[20]), vec_xor(130, k[6]),
vec_xor(131, k[62]), vec_xor(10, k[38]),
&r4, &r26, &r14, &r20);

#if USE_PREFETCH
    // mark the incoming data is about to be read from.
    // address, special value, identifier
    // 1 vector per unit 00001
    // 64 units total 01000000
    // spaced at 16 bytes each. 00000000000010000

```

```

    // with 3 leading bits and spacers:
    // 0000 0001 0100 0000 0000 0000 0001 0000
    // in hex: 0x01100010
    vec_dst(c, 0x01800010, 2);
#endif

// Now we perform the final permutation IP-1 as we assign the
// computed vectors to their appropriate cypher text slots.
c[0] = 17;
c[1] = r7;
c[2] = 115;
c[3] = r15;
c[4] = 123;
c[5] = r23;
c[6] = 131;
c[7] = r31;
c[8] = 16;
c[9] = r6;
c[10] = 114;
c[11] = r14;
c[12] = 122;
c[13] = r22;
c[14] = 130;
c[15] = r30;
c[16] = 15;
c[17] = r5;
c[18] = 113;
c[19] = r13;
c[20] = 121;
c[21] = r21;
c[22] = 129;
c[23] = r29;
c[24] = 14;
c[25] = r4;
c[26] = 112;
c[27] = r12;
c[28] = 120;
c[29] = r20;
c[30] = 128;
c[31] = r28;
c[32] = 13;
c[33] = r3;
c[34] = 111;
c[35] = r11;
c[36] = 119;

```

```
c[37] = r19;  
c[38] = 127;  
c[39] = r27;  
c[40] = 12;  
c[41] = r2;  
c[42] = 110;  
c[43] = r10;  
c[44] = 118;  
c[45] = r18;  
c[46] = 126;  
c[47] = r26;  
c[48] = 11;  
c[49] = r1;  
c[50] = 19;  
c[51] = r9;  
c[52] = 117;  
c[53] = r17;  
c[54] = 125;  
c[55] = r25;  
c[56] = 10;  
c[57] = r0;  
c[58] = 18;  
c[59] = r8;  
c[60] = 116;  
c[61] = r16;  
c[62] = 124;  
c[63] = r24;  
}
```

C.3 generate_swizzle_vpu.c.pl

This is a perl script which I wrote to generate the necessary 128-bit fill and extract functions for the AltiVec unit.

```
#!/usr/bin/perl -w

# $id$
# generate_swizzle_vpu.c.pl
# used for generating

print<<__END__;
// \ $id\ $

/*
 * swizzle_vpu.c
 * This file was generated by generate_swizzle_vpu.c.pl
 * Contains functions used for bit swizzling on the
 * AltiVec Vector Processing Unit
 * Eric Seidel, 2003.
 */

#include "bitslice.h"
#include <stdio.h>
#include <stdlib.h>
#include "swizzle_vpu.h"

__END__

foreach $size (32, 128) {
    foreach $action ("fill", "extract") {
        $half = $size/2;

        if ($action eq "fill") {
            $blocks = $size;
        } else {
            $blocks = 64;
        }

        if ($size == 128) {
            $vectype = "VEC";
        } else {
            $vectype = "unsigned long";
        }
    }
}
```

```

print<<__END__;
```

```

/*
 * Swizzles data in or out of bitslice arrays.
 * Uses the altivec instructions for SPEED!
 * Autogenerated from perl code
 * written by Eric Seidel, 2003.
 */

__END__
    print "void ", $action , "_data", $size, "_vpu (\n";
    if ($action eq "fill") {
        print "\tconst unsigned char *in,\n";
        print "\tunsigned incount,\n";
        print "\t$t$vectype *out\n";
    } else {
        print "\tconst $vectype *in,\n";
        print "\tunsigned incount,\n";
        print "\tunsigned char *out\n";
    }
    print ") {\n";

print <<__END__;
```

```

    // Since we're a proof of concept, we will assume
    // incount = $size;
    if (incount < $size)
    {
        printf("This function is currently hardcoded to handle: "
            "%i blocks at once, I can't handle %i blocks.\n",
            $size, $size - incount);
        exit(1);
    }
    if (incount > $size)
    {
        printf("This function is currently hardcoded to handle: "
            "%i blocks at once, ignoring extra %i blocks.\n",
            $size, incount - $size);
    }

    if ( ((unsigned long)in % 16) != 0) {
        printf("$action\_data$size\_vpu: in is NOT 16-byte aligned "
            "(%p %% 16 = %i) this would cause unexpected"

```

```

        "behaviour!\n",
        in, ((unsigned long)in % 16));
    exit(1);
}
if ( ((unsigned long)out % 16) != 0) {
    printf("$action\_data$size\_vpu: out is NOT 16-byte aligned "
        "(%p %% 16 = %i) this would cause unexpected "
        "behaviour!\n",
        out, ((unsigned long)out % 16));
    exit(1);
}

__END__
# not pretty, but it works.
foreach $mult (1..($half/8)) {
    @a = ();
    foreach (1..8) {
        push (@a, "v". ((8 * ($mult-1)) + $_));
    }
    print "\tvector unsigned char ", join (" ", @a), "\n";
}

print "\tvector unsigned char table1 = generateTable1();\n";
print "\tvector unsigned char table2 = generateTable2();\n\n\n";

print<<__END__;
#if USE_PREFETCH
    // mark the incoming data as about to be read from.
    // address, special value, identifier
    // 1 vector per unit 00001
    // 16 units total 00010000
    // spaced at 16 bytes each. 0000000000010000
    // with 3 leading bits and spacers:
    // 0000 0001 0001 0000 0000 0000 0001 0000
    // in hex: 0x01100010
    vec_dst((unsigned char *)in, 0x01100010, 0);
#endif

__END__
print "\t// load in all the data.\n";
foreach (1..$half) {
    printf "\tv$_\t= vec_ld( %2i * 16, "
        "(unsigned char *) in);\n",
        $_ - 1;
}

```

```

print "\n\n";

$diff = $blocks/2;
$round = 1;

@full = (1..$half);
while ($diff >= 1) {
    print "\t// Round $round\n";

    @nextfull = ();
    @high = @full;
    @low = splice(@high, ($#high + 1) / 2);

    while (@high) {
        $h = shift(@high);
        $l = shift(@low);

        printf "\tinterleave128c( %3s, %3s );\n",
            "v" . $h, "v" . $l;

        push (@nextfull, $h);
        push (@nextfull, $l);
    }
    $round++;
    print "\n";

    @full = @nextfull;
    $diff /= 2;
}

print<<__END__;

#if USE_PREFETCH
    // mark the outgoing data as about to be written to.
    // address, special value, identifier
    // 1 vector per unit 00001
    // 16 units total 00010000
    // spaced at 16 bytes each. 0000000000010000
    // with 3 leading bits and spacers:
    // 0000 0001 0001 0000 0000 0000 0001 0000
    // in hex: 0x01100010
    vec_dst((unsigned char *)out, 0x01100010, 0);
#endif

__END__

```

```
# we're done, write them out.
print "\t// Final output\n";
$mem = 0;
foreach (@full) {
    printf "\tvec_st( %3s, %2i * 16, "
        "(unsigned char *)out );\n",
        "v" . $_, $mem++;
}

print "}\n\n";
}
}
```

C.4 swizzle_iu.c

A C language source file containing my hand-written integer-unit swizzle functions for both 32-bit and 128-bit swizzling.

```
/*
 * swizzle_iu.c
 * Contains hand-written swizzle functions for the integer unit
 * Eric Seidel, 2003.
 */

#include "bitslice.h"

/*
 * Fills vector with data from array of blocks.
 * Written by Eric Seidel, 2003.
 */

void fill_data32_iu ( const unsigned char in[][8],
                    short incount, unsigned long *out)
{
    // This function makes left to right order assumptions.
    // I OR all the data together before storing to achieve
    // greater efficiency as memory accesses are SLOW.

    short i, whichblock;

    if (incount > bitlength)
    {
        printf("Can only handle: %i blocks at once, "
              "ignoring extra %i blocks.\n",
              bitlength, incount - bitlength);
        incount = bitlength;
    }

    for (i=0; i < 64; i++)
    {
        const short  whichbyte = i / 8;
        // which of 8 bytes
        const short  whichbit  = i % 8;
        // which of 8 bits
        const unsigned char mask = (128 >> whichbit);
        // mask to isolate bit
        const short  whichbit_inv = 7 - whichbit;
    }
}
```

```

        // inverse of whichbit, used for shift.
        unsigned long temp = 0;
        // to the block, as we build it.

        for (whichblock = 0; whichblock < incount; whichblock++)
        {
// Shift it to the right (to the 0 bit)
// and then shift it back left (to the correct block bit)
// combined these both into one operation...
// This if statement tries to optimize out the two shifts, since
// the PowerPC does not accept negative values for shr.
// Unknown if the if block is faster, however the shift is known
// to require two cycles, plus a 2 cycle stall on the data.
            if ((31 - whichblock) > whichbit_inv)
                temp |= (in[whichblock][whichbyte] & mask)
                    << ((31 - whichblock) - whichbit_inv);
            else
                temp |= (in[whichblock][whichbyte] & mask)
                    >> (whichbit_inv - (31 - whichblock));
        }
#ifdef BIG_ENDIAN_ORDER
        out[i] = temp; // big endian.
#else
        out[63-i] = temp; // little endian
#endif
    }
}

/*
 * Extracts block data from vector into an array of blocks.
 * Written by Eric Seidel, 2003.
 */

void extract_data32_iu (
    const unsigned long *in, short incount,
    unsigned char out[][8])
{
    short i, whichblock;

    if (incount > bitlength)
    {
        printf("Can only handle: %i blocks at once, ignoring "
            "extra %i blocks.\n",
            bitlength, incount - bitlength);
    }
}

```

```

    incount = bitlength;
}

// Make sure the memory is cleared.
memset(out,0, incount * sizeof(unsigned char) * 8);

for (i=0; i < 64; i++)
{
    // Written out in long form to be easier to understand.
    // The compiler should optimize away all these constants.
    const short whichbyte = i / 8;
    // which of 8 bytes
    const short whichbit = i % 8;
    // which of (first 7 of the) 8 bits

    // iterate over the blocks.
    for (whichblock = 0; whichblock < incount; whichblock++)
    {
        // First we generate a mask with which to isolate the bit
        // we store that in mask. Then we apply that mask to the
        // data and store the isolated bit in isolated.
#ifdef BIG_ENDIAN_ORDER
        const unsigned long mask
            = (1 << (31 - whichblock)); // mask - big
        const unsigned char isolated
            = (in[i] & mask) >> (31 - whichblock); // big
#else
        const unsigned long mask
            = (1 << whichblock); // mask - little
        const unsigned char isolated
            = (in[63-i] & mask) >> whichblock; // little
#endif
        out[whichblock][whichbyte] |=
            isolated << (7 - whichbit);
        // shift it to the correct place, and save.
    }
}

}

/*
 * Fills vector with data from array of blocks.
 * Written by Eric Seidel, 2003.
 */

```

```

void fill_data128_iu ( const unsigned char in[] [8],
                      short incount, VEC *out)
{
    int i, whichblock;

    if (incount > 128)
    {
        printf("Can only handle: %i blocks at once, ignoring "
              "extra %i blocks.\n",
              128, incount - 128);
        incount = 128;
    }

    // Make sure the memory is cleared.
    memset(out,0, 64 * sizeof(VEC));

    for (i=0; i < 64; i++)
    {
        const short  whichbyte = i / 8;
        // which of 8 bytes
        const short whichbit = i % 8;
        // which of 8 bits
        const unsigned char mask = (128 >> whichbit);
        // mask to isolate bit
        const short whichbit_inv = 7 - whichbit;
        // inverse of whichbit, used for shift.
        VEC temp;
        // to the block, as we build it.
        memset(&temp, 0, sizeof(VEC));

        for (whichblock = 0; whichblock < incount; whichblock++)
        {
            if ((31 - (whichblock % 32)) > whichbit_inv)
                temp.i[whichblock/32]
                    |= (in[whichblock][whichbyte] & mask)
                    << ((31 - (whichblock % 32)) - whichbit_inv);
            else
                temp.i[whichblock/32]
                    |= (in[whichblock][whichbyte] & mask)
                    >> (whichbit_inv - (31 - (whichblock % 32)));
        }
    }
    #if BIG_ENDIAN_ORDER
        out[i] = temp; // big endian.
    #else
        out[63-i] = temp; // little endian
    #endif
}

```

```

#endif
    }
}

/*
 * Extracts block data from vector into an array of blocks.
 * Written by Eric Seidel, 2003.
 */

void extract_data128_iu ( const VEC *in, short incount,
                        unsigned char out[][8] )
{
    int i, whichblock;

    if (incount > 128)
    {
        printf("Can only handle: %i blocks at once, ignoring "
              "extra %i blocks.\n",
              128, incount - 128);
        incount = 128;
    }

    // Make sure the memory is cleared.
    // necessary?
    memset(out,0, 128 * sizeof(unsigned char) * 8);

    for (i=0; i < 64; i++)
    {
        // Written out in long form to be easier to understand.
        // The compiler should optimize away all these constants.
        const short  whichbyte = i / 8;
        // which of 8 bytes
        const short whichbit = i % 8;
        // which of (first 7 of the) 8 bits

        // iterate over the blocks.
        for (whichblock = 0; whichblock < incount; whichblock++)
        {
#if BIG_ENDIAN_ORDER
            const unsigned long mask
                = (1 << (31 - (whichblock % 32))); // mask - big
            const unsigned char isolated
                = (in[i].i[whichblock/32] & mask)
                  >> (31 - (whichblock % 32)); // big
#endif
        }
    }
}

```

```

#else
    const unsigned long mask
        = (1 << (whichblock % 32)); // mask - little
    const unsigned char isolated
        = (in[63-i].i[whichblock/32] & mask)
            >> (whichblock % 32); // little
#endif
    out[whichblock][whichbyte]
        |= isolated << (7 - whichbit);
    // shift it to the correct place, and save.
}
}
}

```

C.5 swizzle_vpu.h

A C language header file containing necessary support functions for the auto-generated Altivec swizzling code found in swizzle_vpu.c (see Appendix C.6).

```
// $id$

/*
 * swizzle_vpu.h
 * This file contains all the support-code necessary
 * for proper compilation (and execution) of the
 * autogenerated code in swizzle_vpu.c
 */

#ifndef _swizzle_vpu_h_
#define _swizzle_vpu_h_ 1

#define GENERATED_CONSTANTS 1
#define USE_PREFETCH 1

/*
 * This function generates the "table1" constant used
 * by the interleave functions.
 *
 * The vector constant could be generated instead of loaded
 * from memory in order to increase computation speed
 * (It takes as many as 40 cycles to load the constant from memory.)
 *
 * Table1 is simply an array of all 4 bit numbers with each
 * bit padded by a 0 to the right.
 *
 * i.e.
 * 00: 0000 0000
 * 01: 0000 0010
 * ...
 * 07: 0010 1010
 * ...
 * 12: 1010 0000
 * ...
 * 14: 1010 1000
 * 15: 1010 1010
 */

inline vector unsigned char
```

```

generateTable1()
{
    // this could be replaced by generation code for further speed.
    // right now we pay a 40 cycle penalty to load this data.
    return ( vector unsigned char )
        ( 0x00, 0x02, 0x08, 0x0A,
          0x20, 0x22, 0x28, 0x2A,
          0x80, 0x82, 0x88, 0x8A,
          0xA0, 0xA2, 0xA8, 0xAA );
}

```

```

/*
 * This function generates the "table2" constant used
 * by the interleave functions.
 *
 * The vector constant could be generated instead of loaded
 * from memory in order to increase computation speed
 * (It takes as many as 40 cycles to load the constant from memory.)
 *
 * Table1 is simply an array of all 4 bit numbers with each
 * bit padded by a 0 to the left.
 *
 * i.e.
 * 00: 0000 0000
 * 01: 0000 0001
 * ...
 * 07: 0001 0101
 * ...
 * 12: 0101 0000
 * ...
 * 14: 0101 0100
 * 15: 0101 0101
 */

```

```

inline vector unsigned char
generateTable2()
{
    // this could be replaced by generation code eventually.
    // right now we pay a 40 cycle penalty to load this data.
    return ( vector unsigned char )
        ( 0x00, 0x01, 0x04, 0x05,
          0x10, 0x11, 0x14, 0x15,
          0x40, 0x41, 0x44, 0x45,
          0x50, 0x51, 0x54, 0x55 );
}

```

```

}

/*
 * This function bit-interleaves two 128 bit vectors and is
 * for the "swizzling" necessary to convert from block streams
 * to vectors of blocks for bitslice.
 *
 * This code is derived from code found in bitInterleave.c of the
 * bitInterleave project of the AltiVec Examples distributed
 * by Apple Computer.
 */

inline void
interleave128 (
    vector unsigned char stream1, vector unsigned char stream2,
    vector unsigned char table1, vector unsigned char table2,
    vector unsigned char *high, vector unsigned char *low)
{
    vector unsigned char v1, v2, v3, v4, v10, v11, v30, v31;

#ifdef GENERATED_CONSTANTS
    v30 = vec_splat_u8(0x04);
    // more efficient. 1 cycle instead of possibly 40.
    // more efficient. 6 cycles instead of 40
    v31 = ( vector unsigned char ) vec_splat_u16(0x0f);
    // ( 0x00, 0x0f, 0x00, 0x0f, ... ,0x00, 0x0f)
    v31 = vec_or(v31, vec_sll(v31, v30));
    // ( 0x00, 0xff, 0x00, 0xff, ... ,0x00, 0xff)
    v31 = vec_slo(v31, vec_splat_u8(0x08));
    // ( 0xff, 0x00, 0xff, 0x00, ... ,0xff, 0x00)
#else
    v30 = ( vector unsigned char ) ( 0x04 );
    v31 = ( vector unsigned char )
        ( 0xff, 0x00, 0xff, 0x00,
          0xff, 0x00, 0xff, 0x00,
          0xff, 0x00, 0xff, 0x00,
          0xff, 0x00, 0xff, 0x00 );
#endif

    v1 = vec_mergel ( stream1, stream1 );
    v2 = vec_mergel ( stream2, stream2 );
    v3 = vec_sll ( v1, v30 );
    v4 = vec_sll ( v2, v30 );
    v1 = vec_sel ( v1, v3, v31 );

```

```

v2 = vec_sel    ( v2, v4, v31 );
v10 = vec_perm ( table1, table1, v1 );
v11 = vec_perm ( table2, table2, v2 );
*low = vec_vor ( v10, v11 );

v1 = vec_sld    ( stream1, stream1, 8 );
v2 = vec_sld    ( stream2, stream2, 8 );
v1 = vec_mergel ( v1, v1 );
v2 = vec_mergel ( v2, v2 );
v3 = vec_sll    ( v1, v30 );
v4 = vec_sll    ( v2, v30 );
v1 = vec_sel    ( v1, v3, v31 );
v2 = vec_sel    ( v2, v4, v31 );
v10 = vec_perm ( table1, table1, v1 );
v11 = vec_perm ( table2, table2, v2 );
*high = vec_vor ( v10, v11 );
}

/*
 * convenience function; pre-processed out.
 */
#define interleave128c(a,b)    interleave128(a,b,table1,table2,&a,&b);

#endif

```

C.6 Excerpts from swizzle_vpu.c

A C language source file containing auto-generated vector-unit swizzling functions for both 32-bit and 128-bit swizzling. I have listed here the auto-generated code for swizzling of 32 64-bit blocks into 64 32-bit blocks. Other sources can be generated using `generate_swizzle_vpu.c.pl`.

```
// $id$

/*
 * swizzle_vpu.c
 * This file was generated by generate_swizzle_vpu.c.pl
 * Contains functions used for bit swizzling on the
 * AltiVec Vector Processing Unit
 * Eric Seidel, 2003.
 */

#include "bitslice.h"
#include <stdio.h>
#include <stdlib.h>
#include "swizzle_vpu.h"

/*
 * Swizzles data in or out of bitslice arrays.
 * Uses the altivec instructions for speed.
 * Autogenerated from perl code
 * written by Eric Seidel, 2003.
 */

void fill_data32_vpu (
const unsigned char *in,
unsigned incount,
unsigned long *out
) {
    if ( ((unsigned long)in % 16) != 0) {
        printf("fill_data32_vpu: in is NOT 16-byte aligned "
              "(%p %% 16 = %i) this would cause unexpected behaviour!\n",
              in, ((unsigned long)in % 16));
        exit(1);
    }
    if ( ((unsigned long)out % 16) != 0) {
        printf("fill_data32_vpu: out is NOT 16-byte aligned "
              "(%p %% 16 = %i) this would cause unexpected behaviour!\n",
```

```

        out, ((unsigned long)out % 16));
    exit(1);
}

vector unsigned char v1, v2, v3, v4, v5, v6, v7, v8;
vector unsigned char v9, v10, v11, v12, v13, v14, v15, v16;
vector unsigned char table1 = generateTable1();
vector unsigned char table2 = generateTable2();

#if USE_PREFETCH
    // mark the incoming data as about to be read from.
    // address, special value, identifier
    // 1 vector per unit 00001
    // 16 units total 00010000
    // spaced at 16 bytes each. 00000000000010000
    // with 3 leading bits and spacers:
    // 0000 0001 0001 0000 0000 0000 0001 0000
    // in hex: 0x01100010
    vec_dst((unsigned char *)in, 0x01100010, 0);
#endif

// load in all the data.
v1 = vec_ld( 0 * 16, (unsigned char *) in);
v2 = vec_ld( 1 * 16, (unsigned char *) in);
v3 = vec_ld( 2 * 16, (unsigned char *) in);
v4 = vec_ld( 3 * 16, (unsigned char *) in);
v5 = vec_ld( 4 * 16, (unsigned char *) in);
v6 = vec_ld( 5 * 16, (unsigned char *) in);
v7 = vec_ld( 6 * 16, (unsigned char *) in);
v8 = vec_ld( 7 * 16, (unsigned char *) in);
v9 = vec_ld( 8 * 16, (unsigned char *) in);
v10 = vec_ld( 9 * 16, (unsigned char *) in);
v11 = vec_ld( 10 * 16, (unsigned char *) in);
v12 = vec_ld( 11 * 16, (unsigned char *) in);
v13 = vec_ld( 12 * 16, (unsigned char *) in);
v14 = vec_ld( 13 * 16, (unsigned char *) in);
v15 = vec_ld( 14 * 16, (unsigned char *) in);
v16 = vec_ld( 15 * 16, (unsigned char *) in);

// Round 1
interleave128c( v1, v9 );
interleave128c( v2, v10 );
interleave128c( v3, v11 );

```

```
interleave128c( v4, v12 );
interleave128c( v5, v13 );
interleave128c( v6, v14 );
interleave128c( v7, v15 );
interleave128c( v8, v16 );
```

```
// Round 2
```

```
interleave128c( v1, v5 );
interleave128c( v9, v13 );
interleave128c( v2, v6 );
interleave128c( v10, v14 );
interleave128c( v3, v7 );
interleave128c( v11, v15 );
interleave128c( v4, v8 );
interleave128c( v12, v16 );
```

```
// Round 3
```

```
interleave128c( v1, v3 );
interleave128c( v5, v7 );
interleave128c( v9, v11 );
interleave128c( v13, v15 );
interleave128c( v2, v4 );
interleave128c( v6, v8 );
interleave128c( v10, v12 );
interleave128c( v14, v16 );
```

```
// Round 4
```

```
interleave128c( v1, v2 );
interleave128c( v3, v4 );
interleave128c( v5, v6 );
interleave128c( v7, v8 );
interleave128c( v9, v10 );
interleave128c( v11, v12 );
interleave128c( v13, v14 );
interleave128c( v15, v16 );
```

```
// Round 5
```

```
interleave128c( v1, v9 );
interleave128c( v2, v10 );
interleave128c( v3, v11 );
interleave128c( v4, v12 );
interleave128c( v5, v13 );
interleave128c( v6, v14 );
interleave128c( v7, v15 );
interleave128c( v8, v16 );
```

```

#if USE_PREFETCH
    // mark the outgoing data as about to be written to.
    // address, special value, identifier
    // 1 vector per unit 00001
    // 16 units total 00010000
    // spaced at 16 bytes each. 0000000000010000
    // with 3 leading bits and spacers:
    // 0000 0001 0001 0000 0000 0000 0001 0000
    // in hex: 0x01100010
    vec_dst((unsigned char *)out, 0x01100010, 0);
#endif

// Final output
vec_st( v1, 0 * 16, (unsigned char *)out );
vec_st( v9, 1 * 16, (unsigned char *)out );
vec_st( v2, 2 * 16, (unsigned char *)out );
vec_st( v10, 3 * 16, (unsigned char *)out );
vec_st( v3, 4 * 16, (unsigned char *)out );
vec_st( v11, 5 * 16, (unsigned char *)out );
vec_st( v4, 6 * 16, (unsigned char *)out );
vec_st( v12, 7 * 16, (unsigned char *)out );
vec_st( v5, 8 * 16, (unsigned char *)out );
vec_st( v13, 9 * 16, (unsigned char *)out );
vec_st( v6, 10 * 16, (unsigned char *)out );
vec_st( v14, 11 * 16, (unsigned char *)out );
vec_st( v7, 12 * 16, (unsigned char *)out );
vec_st( v15, 13 * 16, (unsigned char *)out );
vec_st( v8, 14 * 16, (unsigned char *)out );
vec_st( v16, 15 * 16, (unsigned char *)out );
}

```

C.7 main.c

This is the main file of the Bitslice demo program. This contains the main method, a usage method – for printing program usage information – POSIX signal handling code, and a few other utility methods.

```
/*
 * main.c
 * part of bitslice
 * All the standard program logic and maintenance functions.
 * Eric Seidel, 2003.
 *
 * Parts derived from work by Matthew Kwan and Eric Young.
 */

#include "bitslice.h"
#include "des.h"

#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <string.h>
#include <unistd.h>
#include <errno.h>
#include <sys/time.h>

/*
 * Definitions
 */

/*
 * This controls how many rounds of testing are done by
 * the speed test. It is SUGGESTED that this number be
 * both even and ideally share factors with all numbers 1 - 128
 * although both of these conditions are not mandatory.
 */
#define TEST_SIZE 6553600

/*
 * This is here to allow compatibility with previous Bitslice
 * implementations such as that by Matthew Kwan, 1997.
 */
```

```

    * Set to 0 to compile code which organizes bits in a
    * right-to-left fashion similar to Kwan's original code.
    */
#define BIG_ENDIAN_ORDER 1

/*
 * Local variables.
 */
static int bitlength = 32;
static int bitlength_log2 = 5;

static long total_processed = 0;
static long total_verified = 0;

/*
 * Other c-file includes
 */

#include "swizzle_iu.c"
#ifdef __VEC__
    #include "swizzle_vpu.c"
#endif
#include "test_bs_speed.c"
#include "test_swizzle_speed.c"
#include "swipe_tests.c"

/*
 * Executes an (illegal) privileged instruction
 * to signal the trace program "amber" to
 * start/stop tracing execution.
 */

void
Start_StopAmber (void)
{
    __asm__ volatile ( "mfspr r3, 953" );
}

/*
 * Set the bit length variables.
 * Written by Matthew Kwan - April 1997.
 */

static void

```

```

set_bitlength (void)
{
unsigned long x = ~0UL;

bitlength = 0;
for (x = ~0UL; x != 0; x >>= 1)
    bitlength++;

printf ("%d-bit machine\n", bitlength);

if (bitlength == 64)
    bitlength_log2 = 6;
else if (bitlength == 32)
    bitlength_log2 = 5;
else {
    fprintf (stderr,
            "Cannot handle %d-bit machines\n", bitlength);
    exit (1);
}
}

/*
 * Prints out a 64-bit data block as hex.
 * Taken from Eric Young's libdes
 * Modified for formating.
 */

char *pt(unsigned char *p)
{
    char *ret;
    int i;
    static char *f="0123456789ABCDEF";

    ret=(char *)malloc(25);
    for (i=0; i<8; i++)
    {
        ret[i*3]=f[(p[i]>>4)&0xf];
        ret[i*3+1]=f[p[i]&0xf];
        ret[i*3+2]=' ';
    }
    ret[24]='\0';
    return(ret);
}

/*

```

```

* Loads random data from /dev/random into memory
* and returns a pointer to that data.
*/

unsigned char *
load_random_data()
{
    char *name = "/dev/random";
    int random = -1;
    int bytesread = 0;
    unsigned char *data = 0;

    // open /dev/random
    random = open(name, O_RDONLY);
    if (random <= 0)
    {
        printf("Error (%i) encountered while opening %s.  "
               "I must exit now.\nInfo: %s\n",
               errno, name, strerror(errno));
        exit(1);
    }

    // We test TEST_SIZE blocks, each of which are 64 bits,
    // thus 8 bytes per test block.
    // We need 2 times that number because we do keys too.
    // We add 2 to prevent from running off the edge of memory.
    data = malloc(2 * 8 * (TEST_SIZE + 2));

    if (data <= 0)
    {
        printf("Error (%i) trying to malloc data.  I must exit "
               "now.\nInfo: %s\n",
               errno, strerror(errno));
        exit(1);
    }

    // now read in the data.
    bytesread = read(random, data, sizeof(data));
    if (bytesread != sizeof(data) || errno)
    {
        printf("Error (%i) encountered while reading data from "
               "%s.  %i bytes were read."
               "I must exit now.\nInfo: %s\n", errno,
               name, bytesread, strerror(errno));
        exit(1);
    }
}

```

```

    }
    return data;
}

static void
test_swizzle_speed(unsigned char* data, int rounds)
{
    printf("Performing swizzle speed test with %i round%s, %li "
           "blocks per round.\n",
           rounds, (rounds > 1)?"s":"", TEST_SIZE);

    int i = 0, j = 0;
    for (i = 0; i < (8 * rounds); i++) {
        // do each one rounds times before goign on to the next.
        j = i/rounds;

    if (j == 0) {
        test_speed_fill32_iu(data);
    } else if (j == 1) {
        test_speed_extract32_iu(data);
    } else if (j == 2) {
        test_speed_fill32_vpu(data);
    } else if (j == 3) {
        test_speed_extract32_vpu(data);
    } else if (j == 4) {
        test_speed_fill128_iu(data);
    } else if (j == 5) {
        test_speed_extract128_iu(data);
    } else if (j == 6) {
        test_speed_fill128_vpu(data);
    } else if (j == 7) {
        test_speed_extract128_vpu(data);
    }
    }
}

/*
 * Test the Bitslice encryption routine for speed with random data.
 * this calls in turn a host of autogenerated functions
 * found in test_bs_speed.c
 */

static void

```

```

test_speed_bs_all(unsigned char *data, int rounds)
{
    int i = 0, j = 0;
    for (i = 0; i < (6 * rounds); i++) {
        // do each one rounds times before goign on to the next.
        j = i/rounds;

if (j == 0) {
        test_speed_bs_iu_iu(data);
    } else if (j == 1) {
        test_speed_bs_vpu_iu(data);
    } else if (j == 2) {
        test_speed_bs_none_iu(data);
    } else if (j == 3) {
        test_speed_bs_iu_vpu(data);
    } else if (j == 4) {
        test_speed_bs_vpu_vpu(data);
    } else if (j == 5) {
        test_speed_bs_none_vpu(data);
    }
}
}

/*
 * Test the libdes encryption routine for speed with random data.
 * Modeled loosely after original written by Matthew Kwan - April 1997.
 */

static void
test_speed_ld_unique(unsigned char *data)
{
    unsigned long i;
    int j, t, q;
    double td, ts, Mbps;
    struct timeval start_tv, end_tv;
    des_cblock out;
    des_key_schedule ks;
    unsigned char *pin, *kin;
    unsigned long processed = 0;

    // setup pointers
    pin = data;
    kin = pin + 32*8;
    // we skip 32 (8 byte) blocks to make sure that we are reading
    // the same keys as bitslice.

```

```

// Do a dummy run to get the function loaded into memory
for (i = 0; i < 32; i ++)
{
    if ((j=key_sched((C_Block *) (kin),ks)) != 0)
        printf("Key error %2lu:%d data: %s\n", i+1, j,
            pt(kin));
    bzero(out,8);
    des_ecb_encrypt((C_Block *) (pin),
        (C_Block *)out,ks,DES_ENCRYPT);
    kin += 8; // skip to the next key
    pin += 8; // skip to the next plain-text
}

// reset pointers
pin = data;
kin = pin + 32*8;

// Begin the actual run
gettimeofday (&start_tv, NULL);

for (i=0; i<TEST_SIZE/32; i++)
{
    // encrypt the 32 blocks.
    for (q = 0; q < 32; q ++)
    {
        if ((j=key_sched((C_Block *) (kin),ks)) != 0)
            printf("Key error %2d:%d data: %s\n",q+1,j,
                pt(kin));
        //bzero(out,8);
        des_ecb_encrypt((C_Block *) (pin),
            (C_Block *)out,ks,DES_ENCRYPT);

        kin += 8;
        pin += 8;
    }
    processed += 32;
}

gettimeofday (&end_tv, NULL);

t = (end_tv.tv_sec - start_tv.tv_sec) * 1000000
    + (end_tv.tv_usec - start_tv.tv_usec);
td = (double) t / 1000000.0;
ts = (double) TEST_SIZE/td;
Mbps = (ts * 64.0)/(1024.0 * 1024.0);

```

```

// sanity check:
if (processed != TEST_SIZE)
    printf("Error: only processed: %lu blocks, we think we "
           "did: %i.\n", processed, TEST_SIZE);

printf ("libdes\t\t\tswipe: %3i, %5.2f sec @ %11.2f "
        "blocks/sec %7.2f Mbps %7.2f MBps\n",
        1, td, ts, Mbps, Mbps/8.0);
}

/*
 * Wrapper for running libdes speed tests
 */

static void
test_speed_ld_all(unsigned char *data, int rounds)
{
    int i = 0;
    for (i = 0; i < rounds; i++)
        test_speed_ld_unique(data);
}

/*
 * Wrapper for running swipe speed tests
 * This calls the modified swipe speed function
 * found in swipe_tests.c
 */

static int swipe_count = 23;
static int swipe_order[] =
    { 4,  8, 12, 16, 20, 24,
      26, 28, 29, 30, 31, 32,
      34, 36, 38, 40, 50, 60,
      70, 80, 100, 120, 128};

static void
test_variable_swipe(unsigned char* data, int rounds)
{
    printf("Performing swipe size speed test with %i round%s, "
           "%li blocks per round.\n",
           rounds, (rounds > 1)?"s":"", TEST_SIZE);

    int i = 0, j = 0, k = 0;

```

```

    for (i = 0; i < (2 * rounds); i++) {
        // do each one rounds times before goign on to the next.
        j = i/rounds;

        if (j == 0) {
            test_speed_ld_unique(data);
        } else if (j == 1) {
            for (k = 0; k < swipe_count; k++)
                test_swipe_bs_vpu_vpu(data, swipe_order[k]);
        }
    }
}

/*
 * Called when we recieve a SIG_TERM or SIG_INT
 * Allows us to print out some statistics.
 * Used mostly for the -T test.
 * Written by Eric Seidel, 2003.
 */
void handleEXIT SIGNAL(int signo)
{
    printf("Totals:\n");
    printf("Total Blocks Processed: %lu\n", total_processed);
    printf("Total Blocks Verified: %lu\n", total_verified);
    exit(0);
}

/*
 * Sets up the necessary signal handlers to allow us to do
 * things when the program exits.
 * Taken from Eric Seidel's OpenAG, 2002.
 */
void set_signalhandlers()
{
    struct sigaction act, oact;
    act.sa_handler = &handleEXIT SIGNAL;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;
    if (sigaction(SIGTERM, &act, &oact) < 0)
    {
        printf("Sorry, an error occored while try to set up the "
            "signal handlers, I can't continue.\n");
    }
}

```

```

        exit(1);
    }
    if (sigaction(SIGINT, &act, &oact) < 0)
    {
        printf("Sorry, an error occurred while try to set up the "
              "signal handlers, I can't continue.\n");
        exit(1);
    }
}

void run_practice_data()
{
    unsigned long p[64], c[64], k[64];
    unsigned long cout[64], pout[64];
    unsigned char temp[32][8];
    int num_tests = 64;
    int whichblock = 0;
    short testsOK = 0;
    short testsOKInARow = 0;

    // load up our practice data.
    fill_data32_iu((const unsigned char *)key_data, NUM_TESTS, k);
    fill_data32_iu((const unsigned char *)plain_data,
                  NUM_TESTS, p);
    fill_data32_iu((const unsigned char *)cipher_data,
                  NUM_TESTS, c);

    // run a test with kwan's eval code.
    unsigned long res = bs_des_eval_iu (p, c, k);

    // this should always find keys, should always be 0xFFFFFFFF
    if (res != 0)
        printf("Found keys: %08lX\n", res);

    // Now we test encryption.
    printf("Testing BITSlice Encryption:\n");

    // encrypt and extract the data.
    bs_des_encrypt_iu(p,cout,k);
    extract_data32_iu(cout, NUM_TESTS, (unsigned char *)temp);

    // check each extracted block against the
    // known cipher-text
    for (whichblock=0; whichblock < 32; whichblock++)

```

```

{
    printf("Text #02i:\t", whichblock);
    if (bcmp(cipher_data[whichblock],temp[whichblock],8) != 0)
    {
        printf("FAILED: Original: %s != Final: %s\n",
            pt(plain_data[whichblock]),
            pt(temp[whichblock]));
        testsOKInARow = 0;
    }
    else
    {
        printf("OK\n");
        testsOK++; testsOKInARow++;
    }
}

// Now we test decryption.
printf("Testing BITSlice Decryption:\n");

// decrypt and extract the decrypted blocks.
bs_des_decrypt_iu(c,pout,k);
extract_data32_iu(pout, NUM_TESTS, (unsigned char *)temp);

// test the decrypted blocks against the original plaintext.
for (whichblock=0; whichblock < 32; whichblock++)
{
    printf("Text #02i:\t", whichblock);
    if (bcmp(plain_data[whichblock],temp[whichblock],8) != 0)
    {
        printf("FAILED:\tOriginal: %s != Final: %s\n",
            pt(plain_data[whichblock]),
            pt(temp[whichblock]));
        testsOKInARow = 0;
    }
    else
    {
        printf("OK\n");
        testsOK++; testsOKInARow++;
    }
}

if (testsOK != num_tests)
    printf("Tests completed, %i TESTS FAILED.\n",
        num_tests - testsOK);
else

```

```

        printf("Tests completed, all Tests OK.\n");
    }

/*
 * Test's bitslice's swizzling routines.
 * Both for iu and vpu data.
 * both using the vpu and iu swizzling algorithms.
 */
void run_swizzle_test(int altivec_swizzle_flag)
{
    int testsOK = 0, testsOKInARow = 0;
    unsigned long p[64];
    VEC pv[64];
    unsigned char or[128][8], tv[128][8], temp[32][8];
    int whichblock = 0;
    int num_tests = 32 + 128;

    // Perform swizzling test for IU sized data.
    printf("Testing %s-unit swizzling with 32 plain-data blocks: \n",
           altivec_swizzle_flag?"vector":"integer");

    // fill the bitslice array with data.
    if (altivec_swizzle_flag)
        fill_data32_vpu((const unsigned char *)plain_data,
                       NUM_TESTS, p);
    else
        fill_data32_iu((const unsigned char *)plain_data,
                      NUM_TESTS, p);

    // pull out the data again.
    if (altivec_swizzle_flag)
        extract_data32_vpu(p, NUM_TESTS, (unsigned char *)temp);
    else
        extract_data32_iu(p, NUM_TESTS, (unsigned char *)temp);

    // Now check to make sure the data agrees with before.
    for (whichblock=0; whichblock < 32; whichblock++)
    {
        printf("Text #%02i:\t", whichblock);
        if (bcmp(plain_data[whichblock],temp[whichblock],8) != 0)
        {
            printf("FAILED:\tOriginal: %s != Final: %s\n",
                   pt(plain_data[whichblock]),

```

```

        pt(temp[whichblock]));
    testsOKInARow = 0;
}
else
{
    printf("OK\n");
    testsOK++; testsOKInARow++;
}
}

// Perform swizzling test on VPU sized blocks.
printf("Testing %s-unit swizzling with 128 plain-data blocks: \n",
        altivec_swizzle_flag?"vector":"integer");

// copy in some practice data.
memcpy(or, plain_data, 8*NUM_TESTS);

// fill up the bitslice array
if (altivec_swizzle_flag)
    fill_data128_vpu((unsigned char *)or, 128, pv);
else
    fill_data128_iu((unsigned char *)or, 128, pv);

// extract the data from the bitslice array
if (altivec_swizzle_flag)
    extract_data128_vpu(pv, 128, (unsigned char *)tv);
else
    extract_data128_iu(pv, 128, (unsigned char *)tv);

// check to make sure the data survived the swizzle.
for (whichblock=0; whichblock < 128; whichblock++)
{
    printf("Text #%02i:\t", whichblock);
    if (bcmp(or[whichblock],tv[whichblock],8) != 0)
    {
        printf("FAILED:\tOriginal: %s != Final: %s\n",
                pt(or[whichblock]), pt(tv[whichblock]));
        testsOKInARow = 0;
    }
    else
    {
        printf("OK\n");
        testsOK++; testsOKInARow++;
    }
}
}

```

```

// print test summary information.
if (testsOK != num_tests)
    printf("Tests completed, %i TESTS FAILED.\n", num_tests - testsOK);
else
    printf("Tests completed, all Tests OK.\n");
}

/*
 * Runs first the bitslice version of DES against random data,
 * then runs libdes against the same random data, and checks the two.
 * if they differ, it prints an error.
 *
 * libdes code taken modeled after code from
 * destest.c of Eric Young's libdes.
 */
void run_libdes_test()
{
    int i,j;
    unsigned long p[64], c[64], d[64], k[64];
    unsigned char pin[32][8], kin[32][8];
    unsigned char cry[32][8], dcp[32][8];

    // set up to read in random data.
    char *name = "/dev/random";
    int random;
    random = open(name, O_RDONLY);
    int bytesread = 0;

    des_cblock out, outin;
    des_key_schedule ks;

    if (random <= 0)
    {
        printf("Error (%i) encountered while opening %s.  "
            "I must exit now.\nInfo: %s\n",
            errno, name, strerror(errno));
        exit(1);
    }

    // run until we get ^C
    while (1)
    {
        // read in the data.

```

```

bytesread = read(random, pin,
                 sizeof(unsigned char) * 32 * 8);
if (bytesread != (sizeof(unsigned char) * 32 * 8))
{
    if (errno)
    {
        printf("Error (%i) encountered while reading "
              "plaintext from %s. "
              "I must exit now.\nInfo: %s\n",
              errno, name, strerror(errno));
        exit(1);
    }
    else
    {
        printf("Plaintext only read: %i", bytesread);
        exit(1);
    }
}
bytesread = read(random, kin,
                 sizeof(unsigned char) * 32 * 8);
if (bytesread != (sizeof(unsigned char) * 32 * 8))
{
    if (errno)
    {
        printf("Error (%i) encountered while reading "
              "keytext from %s. "
              "I must exit now.\nInfo: %s\n",
              errno, name, strerror(errno));
        exit(1);
    }
    else
    {
        printf("Keytext only read: %i", bytesread);
        exit(1);
    }
}
// done reading random data.

// populate our vectors.
fill_data32_iu(kin, 32, k);
fill_data32_iu(pin, 32, p);

//encrypt & decrypt
bs_des_encrypt_iu(p,c,k);
bs_des_decrypt_iu(c,d,k);

```

```

total_processed += 32;

// extract data from the bitslice arrays.
extract_data32_iu(c, 32, cry);
extract_data32_iu(d, 32, dcp);

// now run libdes over the same data to
// test it against libdes implmentation.
for (i=0; i<32; i++)
{
    // schedule keys.
    if ((j=key_sched((C_Block *) (kin[i]), ks)) != 0)
        printf("Key error %2d:%d data: %s\n", i+1, j,
            pt(kin[i]));
    bzero(out, 8);
    bzero(outin, 8);

    // encrypt & decrypt
    des_ecb_encrypt((C_Block *) pin[i],
        (C_Block *) out, ks, DES_ENCRYPT);
    des_ecb_encrypt((C_Block *) out,
        (C_Block *) outin, ks, DES_DECRYPT);

    // test the encrypt data against what bitslice produced
    if (bcmp(out, cry[i], 8) != 0)
    {
        printf("BITSLICE Encryption error\n"
            "k=%s p=%s o=%s act=%s\n",
            pt(kin[i]), pt(pin[i]), pt(cry[i]),
            pt(out));
    }
    else
    {
        total_verified++;
    }

    // test the decrypt data.
    if (bcmp(pin[i], outin, 8) != 0)
    {
        printf("LIBDES Decryption error\n"
            "k=%s p=%s o=%s act=%s\n",
            pt(kin[i]), pt(out), pt(pin[i]), pt(outin));
    }
    if (bcmp(dcp[i], outin, 8) != 0)
    {

```

```

                printf("BITSLICE Decryption error\n"
                       "k=%s c=%s p=%s act=%s\n",
                       pt(kin[i]),pt(out),pt(outin),pt(dcp[i]));
            }
        }
    }

/*
 * Displays program usage information.
 * Written by Eric Seidel, 2003.
 */
void usage(char* name)
{
    printf("Usage: %s [options] [file]\n", name);
    printf("Options:\n");
    printf("  -S[n]\tPerforms an n-round speed test of "
           "%i 64-bit blocks. "
           "0 < n < 10, default n=1\n", TEST_SIZE);
    printf("  -T\tTests encryption against libdes.  Runs "
           "continuously until ^C.\n");
    printf("  -L[a]\tPerforms swizzling tests (loading/unloading from "
           "vector). ('a' enables altivec code)\n");
    printf("  -P\tPerforms practice run with libdes practice data.\n");
    printf("  -W[n]\tPerforms an n-round swizzling speed test."
           "0 < n < 10, default n=1\n");
    printf("  -E\tPerforms swipe-size test against all modes "
           "of Bislice\n");
    printf("  -a\tRuns Bitslice with amber trace start/stop "
           "instructions.\n");
    exit(1);
}

/*
 * Main entry point.
 * Modeled after Matthew Kwan's original main function
 * (and just about every other unix main function)
 */
int
main (
int argc,
char *argv[]

```

```

) {
// variables and flags.
    short i = 0;
    int bs_speed_flag = 0;
    short bs_speed_rounds = 1;
    int swizzle_speed_flag = 0;
    short swizzle_speed_rounds = 1;
    int practice_flag = 0;
int correctness_test_flag = 0;
    int swipe_size_test_flag = 0;
    short swipe_test_rounds = 1;
    int swizzle_test_flag = 0;
    int altivec_swizzle_flag = 0;
    int trace_flag = 0;

// parse the arguments.
for (i=1; i<argc; i++) {
    if (argv[i][0] != '-')
        continue;

    if (argv[i][1] == 'S')
    {
        bs_speed_flag = 1;
        if (('1' <= argv[i][2]) && (argv[i][2] <= '9'))
            bs_speed_rounds = argv[i][2] - '0';
    }
    else if (argv[i][1] == 'W')
    {
        swizzle_speed_flag = 1;
        if (('1' <= argv[i][2]) && (argv[i][2] <= '9'))
            swizzle_speed_rounds = argv[i][2] - '0';
    }
    else if (argv[i][1] == 'T')
        correctness_test_flag = 1;

    else if (argv[i][1] == 'P')
        practice_flag = 1;

    else if (argv[i][1] == 'L')
    {
        swizzle_test_flag = 1;
        if (argv[i][2] == 'a')
            altivec_swizzle_flag = 1;
    }
    else if (argv[i][1] == 'a')

```

```

        trace_flag = 1;

    else if (argv[i][1] == 'E')
    {
        swipe_size_test_flag = 1;
        if (('1' <= argv[i][2]) && (argv[i][2] <= '9'))
            swipe_test_rounds = argv[i][2] - '0';
    }
    else
    {
        usage(argv[0]);
    }
}

// check what kind of processor we have.
set_bitlength ();

// break off into various tests.
if (correctness_test_flag)
{
    set_signalhandlers();
    run_libdes_test();
}
else if (swizzle_test_flag)
    run_swizzle_test(altivec_swizzle_flag);

else if (bs_speed_flag)
{
    printf("Performing encryption speed test with "
           "%i round%s, %i blocks per round.\n",
           bs_speed_rounds, (bs_speed_rounds > 1)?"s":"",
           TEST_SIZE);

    unsigned char *data = load_random_data();
    test_speed_ld_all(data, bs_speed_rounds);
    printf("\n");
    test_speed_bs_all(data, bs_speed_rounds);
    free(data);
}
else if (swizzle_speed_flag)
{
    unsigned char *data = load_random_data();
    test_swizzle_speed(data, swizzle_speed_rounds);
    free(data);
}

```

```

else if (practice_flag)
    run_practice_data();

else if (trace_flag)
{
    vector unsigned int p[64],c[64],k[64];
    // once to load it into memory.
    bs_des_encrypt_vpu(p,c,k);

    // run
    Start_StopAmber();
    bs_des_encrypt_vpu(p,c,k);
    Start_StopAmber();
}
else if (swipe_size_test_flag)
{
    unsigned char *data = load_random_data();
    test_variable_swipe(data, swipe_test_rounds);
    free(data);
}

else
    usage(argv[0]);
exit (0);
}

```

C.8 generate_bs_speed_tests.pl

A perl script which generates a file containing all the speed tests used on Bitslice when executed with the “-S” flag.

```
#!/usr/bin/perl -w

foreach $random (1) {
    foreach $encryptsize (32, 128) {
        if ($encryptsize == 32) {
            $encrypt = "iu";
            $vector = "unsigned long";
        } else {
            $encrypt = "vpu";
            $vector = "vector unsigned int";
        }

        foreach $swizzle ("iu", "vpu", "none") {

            print "/*\n";
            print " * This runs a bitslice speed test using the $swizzle ",
                "for swizzling\n";
            print " * and the $encrypt for encrypting.\n";
            print " * This simulates encrypting a large file in ECB mode ",
                "with a single key\n";
            print " */\n\n";

            print "static void\n";
            print "test_speed_bs_", $swizzle, "_$encrypt(unsigned char ",
                "*data)\n";
            print <<__END__>>

{
    unsigned long i = 0, processed = 0;
    int t = 0;
    double td, ts, Mbps;
    struct timeval start_tv, end_tv;
    unsigned char *pin, *kin, cout[$encryptsize];
    $vector p[64], c[64], k[64];

    // setup pointers
    pin = data;
    kin = pin + $encryptsize*8; // we skip $encryptsize 8-byte blocks

__END__

```

```

if ($swizzle ne "none") {
    print "    // Do a dummy run to get the function loaded into memory\n";
    print "    // populate our vectors.\n";
    print "    fill_data", $encryptsize,
        "_$swizzle(kin, $encryptsize, k);\n";
    print "    fill_data", $encryptsize,
        "_$swizzle(pin, $encryptsize, p);\n";
} else {
    print "    // populate the key vector.\n";
    print "    fill_data", $encryptsize, "_vpu(kin, $encryptsize, k);\n";
}
print <<__END__;
    bs_des_encrypt_$encrypt(p,c,k);

    // Begin the actual run
    gettimeofday (&start_tv, NULL);

    for (i=0; i< TEST_SIZE/$encryptsize; i++) {
__END__
if ($swizzle ne "none") {
    print "\t// populate\n";
    print "\tfill_data", $encryptsize,
        "_$swizzle(pin, $encryptsize, p);\n\n";

    print "\t//encrypt\n";
    print "\tbs_des_encrypt_$encrypt(p,c,k);\n\n";

    print "\t// unswizzle\n";
    print "\textract_data", $encryptsize,
        "_$swizzle(c, $encryptsize, cout);\n\n";
} else {
    print "\t//encrypt\n";
    print "\tbs_des_encrypt_$encrypt(pin,c,k);\n\n";
}
if ($random ){
    print "\t// advance pointers\n";
    print "\tpin += $encryptsize*8*2; // skip to the next ",
        "batch of $encryptsize 8-byte data blocks\n";
    if ($swizzle ne "none") {
        print "\tkin += $encryptsize*8*2; // skip to the next ",
            "batch of $encryptsize 8-byte key blocks\n";
    }
}

    print "\tprocessed += $encryptsize;\n";
print <<__END__;

```

```

}

gettimeofday (&end_tv, NULL);

t = (end_tv.tv_sec - start_tv.tv_sec) * 1000000
    + (end_tv.tv_usec - start_tv.tv_usec);
td = (double) t / 1000000.0;
ts = (double) TEST_SIZE/td;
Mbps = (ts * 64.0)/(1024.0 * 1024.0);

// sanity check:
if (processed != TEST_SIZE)
    printf("Error: only processed: %li blocks, we think we did: %i.\n",
          processed, TEST_SIZE);

printf ("Bitslice-DES ($swizzle/$encrypt)\tswipe: %3i, %5.2f sec @ "
        "%11.2f blocks/sec %7.2f Mbps %7.2f MBps\n",
        $encryptsize, td, ts, Mbps, Mbps/8.0);
}

__END__
}
}
}

```

C.9 generate_swizzle_speed_tests.pl

A perl script which generates a file containing all the speed tests used to test AltiVec swizzling. These tests are run when Bitslice is executed with the “-W” flag.

```
#!/usr/bin/perl -w

$random = 1;
foreach $action ("fill", "extract") {
    foreach $encryptsize (32, 128) {
        if ($encryptsize == 32) {
            $encrypt = "iu";
            $vector = "unsigned long";
        } else {
            $encrypt = "vpu";
            $vector = "vector unsigned int";
        }

        foreach $swizzle ("iu", "vpu") {

            print "/*\n";
            print " * This runs a $encryptsize-bit swizzle "
                "speed test using the $swizzle for "
                "swizzling\n";
            print " */\n\n";

            print "static void\n";
            print "test_speed_$action$encryptsize", "_",
                "$swizzle, "(unsigned char *data)\n";
            print <<__END__;
{
    unsigned long i = 0, processed = 0;
    int t = 0;
    double td, ts, Mbps;
    struct timeval start_tv, end_tv;
    $vector d[64];

__END__
if ($swizzle ne "none") {
    print " // Do a dummy run to get the function loaded "
        "into memory\n";
    print " $action\_data", $encryptsize,
        "\_$swizzle(data, $encryptsize, d);\n";
}
}
```

```

print <<__END__;

    // Begin the actual run
    gettimeofday (&start_tv, NULL);

    for (i=0; i< TEST_SIZE/$encryptsize; i++) {
__END__
if ($swizzle ne "none") {
    print "\t// this is probably very inefficient...\n";
    print "\t// populate\n";
    print "\t$action\_data", $encryptsize,
        "\t_$swizzle(data, $encryptsize, d);\n";
}
if ($random) {
    print "\t// advance pointers\n";
    print "\tdata += $encryptsize*8; "
        "\t// skip to the next batch of $encryptsize "
        "\t8-byte data blocks\n";
}

    print "\tprocessed += $encryptsize;\n\n";
print <<__END__;
}
gettimeofday (&end_tv, NULL);

t = (end_tv.tv_sec - start_tv.tv_sec) * 1000000
    + (end_tv.tv_usec - start_tv.tv_usec);
td = (double) t / 1000000.0;
ts = (double) TEST_SIZE/td;
Mbps = (ts * 64.0)/(1024.0 * 1024.0);

// sanity check:
if (processed != TEST_SIZE)
    printf("Error: only processed: %li blocks, we think "
        "we did: %i.\n", processed, TEST_SIZE);

printf ("%s\t%5.2f sec @ %11.2f blocks/sec %7.2f Mbps "
        "%7.2f MBps\n",
        "$action\_data$encryptsize\_swizzle()",
        td, ts, Mbps, Mbps/8.0);
}
__END__
    print "\n\n\n";
}
}
}

```

C.10 swipe_tests.c

A hand-written C source file which contains a version `test_swipe_bs_vpu_vpu` used for the swipe size tests.

```
#include <unistd.h>
#include <sys/time.h>
#include "bitslice.h"

/*
 * This runs a bitslice speed test using the vpu for swizzling
 * and the vpu for encrypting.
 * This simulates encrypting a large file in ECB mode
 * with a single key
 */

void
test_swipe_bs_vpu_vpu(unsigned char *data, unsigned int PERSWIPE)
{
    unsigned long  i = 0, processed = 0;
    int t = 0;
    double td, ts, Mbps;
    struct timeval start_tv, end_tv;
    unsigned char *pin, *kin, cout[128][8];
    vector unsigned int p[64], c[64], k[64];

    // setup pointers
    pin = data;
    kin = pin + 128*8; // we skip 128 8-byte blocks

    fill_data128_vpu(kin, 128, k);
    fill_data128_vpu(pin, 128, p);
    bs_des_encrypt_vpu(pin,c,k);

    // Begin the actual run
    gettimeofday (&start_tv, NULL);

    for (i=0; i< TEST_SIZE/PERSWIPE; i++) {
// populate
fill_data128_vpu(pin, 128, p);

//encrypt
bs_des_encrypt_vpu(p,c,k);
```

```

// unswizzle
extract_data128_vpu(c, 128, cout);

processed += PERSWIPE;
}

gettimeofday (&end_tv, NULL);

t = (end_tv.tv_sec - start_tv.tv_sec) * 1000000
    + (end_tv.tv_usec - start_tv.tv_usec);
td = (double) t / 1000000.0;
ts = (double) TEST_SIZE/td;
Mbps = (ts * 64.0)/(1024.0 * 1024.0);

// no sanity check, the swipe size not be a multiple
// of the test size.

printf ("Bitslice-DES (vpu/vpu)\tswipe: %3i, %5.2f sec"
        "@ %11.2f blocks/sec %7.2f Mbps %7.2f MBps\n",
        PERSWIPE, td, ts, Mbps, Mbps/8.0);
}

```

C.11 altivec_sboxes_c.pl

This is a perl script which I wrote to convert Kwan's S-Boxes (found in Appendix C.13) to an altivec compatible form.

```
#!/usr/bin/perl

print "/*\n";
print " * Altivec-enabled Standard S-Boxes\n";
print " * Derived directly from Matthew Kwan's S-Boxes.\n";
print " * Modified using altivec_sboxes_c.pl\n";
print " * Eric Seidel - March 2002\n";
print " */\n\n";
print "#include \"sboxes.h\"\n\n";
while(<>)
{
s/unsigned long/vector unsigned int/;
    s/([^\s(),;]+)\s&\s~\s([^\s(),;]+)/vec_andc($1, $2)/g;
    s/([^\s(),;]+)\s\^~\s([^\s(),;]+)/vec_nor($1, $2)/g;
    s/\s~\s([^\s(),;]+)/ vec_nor($1, $1)/g;
s/([^\s(),;]+)\s\^~\s([^\s(),;]+)/vec_xor($1, $2)/g;
s/([^\s(),;]+)\s\|\s([^\s(),;]+)/vec_or($1, $2)/g;
s/([^\s(),;]+)\s&\s([^\s(),;]+)/vec_and($1, $2)/g;
    s/([^\s(),;]+)\s\^=\s([^\s(),;]+)/$1 = vec_xor($1, $2)/g;
s/([^\s(),;]+)\s&=\s([^\s(),;]+)/$1 = vec_and($1, $2)/g;
s/([^\s(),;]+)\s\|=\s([^\s(),;]+)/$1 = vec_or($1, $2)/g;
print;
}

# Altivec Logical Instructions:
# vec_and
# vac_andc
# vec_or
# vec_nor
# vec_xor
```

C.12 swap_endian_bitslice_c.pl

This is a perl script which I wrote to swap the bit ordering used throughout Matthew Kwan's Bitslice code. This was useful during testing to allow me to test my code against his implementation.

```
#!/usr/bin/perl

# swap_endian_bitslice.pl
# Converts Kwan's bitslice code from right-to-left bit
# ordering to left-to-right bit ordering for comparision
# with my own bitslice code.
# Eric Seidel, 2003.

# exp is used to expand a references to a
# 56-bit array, into a reference to a 64-bit
# array. This is used for re-figuring Kwan's
# key lookups
@exp=( 0, 1, 2, 3, 4, 5, 6,
      8, 9, 10, 11, 12, 13, 14,
      16, 17, 18, 19, 20, 21, 22,
      24, 25, 26, 27, 28, 29, 30,
      32, 33, 34, 35, 36, 37, 38,
      40, 41, 42, 43, 44, 45, 46,
      48, 49, 50, 51, 52, 53, 54,
      56, 57, 58, 59, 60, 61, 62);

# process all the input from files
while(<>)
{
    # replace corresponding p,c,k array lookups
    # with lookups for the reverse ordering.
    s/p\[(\d{1,2})\]/"p[".(63-$1)."]"/ge;
    s/c\[(\d{1,2})\]/"c[".(63-$1)."]"/ge;
    s/k\[(\d{1,2})\]/"k[".$exp[55-$1]."]"/ge;
    print $_;
}
```

C.13 Excerpt from kwan.c

I have included only one of Matthew Kwan’s original “standard” integer unit S-Boxes⁹⁷. To include all eight integer unit S-Boxes here is unnecessary as they can be retrieved from Kwan’s site[29], and with only minor modifications⁹⁸ made to work with my source. I have also chosen not to include Kwan’s `bs_des_eval` source as that to can be retrieved and made to work with my source with only minor modifications.

```
/*
 * Left-to-right IU des_eval function and S-boxes
 * Modified from Matthew Kwan’s original right-to-left
 * des_eval function and S-boxes.
 * Eric Seidel, 2003
 * Matthew Kwan, 1998.
 */

#include "bitslice.h"

static inline void
s1_iu (
unsigned long a1,
unsigned long a2,
unsigned long a3,
unsigned long a4,
unsigned long a5,
unsigned long a6,
unsigned long *out1,
unsigned long *out2,
unsigned long *out3,
unsigned long *out4
) {
unsigned long x1, x2, x3, x4, x5, x6, x7, x8;
unsigned long x9, x10, x11, x12, x13, x14, x15, x16;
unsigned long x17, x18, x19, x20, x21, x22, x23, x24;
unsigned long x25, x26, x27, x28, x29, x30, x31, x32;
unsigned long x33, x34, x35, x36, x37, x38, x39, x40;
unsigned long x41, x42, x43, x44, x45, x46, x47, x48;
```

⁹⁷Kwan’s “standard” S-Boxes are designed for platforms not supporting the “non-standard” ANDC, NOR and NAND boolean operations. For a discussion of Boolean operations and their meaning, see Appendix A.3.

⁹⁸S-Box functions must be renamed from `s1` to `s1_iu` and the `altivec.sboxes.pl` script must be applied to generate vpu enabled S-Boxes.

```
unsigned long x49, x50, x51, x52, x53, x54, x55, x56;  
unsigned long x57, x58, x59, x60, x61, x62, x63;
```

```
x1 = ~a4;  
x2 = ~a1;  
x3 = a4 ^ a3;  
x4 = x3 ^ x2;  
x5 = a3 | x2;  
x6 = x5 & x1;  
x7 = a6 | x6;  
x8 = x4 ^ x7;  
x9 = x1 | x2;  
x10 = a6 & x9;  
x11 = x7 ^ x10;  
x12 = a2 | x11;  
x13 = x8 ^ x12;  
x14 = x9 ^ x13;  
x15 = a6 | x14;  
x16 = x1 ^ x15;  
x17 = ~x14;  
x18 = x17 & x3;  
x19 = a2 | x18;  
x20 = x16 ^ x19;  
x21 = a5 | x20;  
x22 = x13 ^ x21;  
*out4 ^= x22;  
x23 = a3 | x4;  
x24 = ~x23;  
x25 = a6 | x24;  
x26 = x6 ^ x25;  
x27 = x1 & x8;  
x28 = a2 | x27;  
x29 = x26 ^ x28;  
x30 = x1 | x8;  
x31 = x30 ^ x6;  
x32 = x5 & x14;  
x33 = x32 ^ x8;  
x34 = a2 & x33;  
x35 = x31 ^ x34;  
x36 = a5 | x35;  
x37 = x29 ^ x36;  
*out1 ^= x37;  
x38 = a3 & x10;  
x39 = x38 | x4;  
x40 = a3 & x33;
```

```
x41 = x40 ^ x25;
x42 = a2 | x41;
x43 = x39 ^ x42;
x44 = a3 | x26;
x45 = x44 ^ x14;
x46 = a1 | x8;
x47 = x46 ^ x20;
x48 = a2 | x47;
x49 = x45 ^ x48;
x50 = a5 & x49;
x51 = x43 ^ x50;
*out2 ^= x51;
x52 = x8 ^ x40;
x53 = a3 ^ x11;
x54 = x53 & x5;
x55 = a2 | x54;
x56 = x52 ^ x55;
x57 = a6 | x4;
x58 = x57 ^ x38;
x59 = x13 & x56;
x60 = a2 & x59;
x61 = x58 ^ x60;
x62 = a5 & x61;
x63 = x56 ^ x62;
*out3 ^= x63;
}
```

D Program Output

D.1 Usage Statement

```
[localhost:~/] % ./bitslice
```

```
Usage: ./bitslice [options] [file]
```

```
Options:
```

- S[n] Performs an n-round speed test of 6553600 64-bit blocks.
0 < n < 10, default n=1
- T Tests encryption against libdes. Runs continuously until ^C.
- L[a] Performs swizzling tests (loading/unloading from vector).
(‘a’ enables altivec code)
- P Performs practice run with libdes practice data.
- W[n] Performs an n-round swizzling speed test. 0 < n < 10, default n=1
- E Performs swipe-size test against all modes of bislice
- a Runs bitslice with amber trace start/stop instructions.

D.2 Sample Output “-S”

```
[localhost:~/] % ./bitslice -S
```

```
32-bit machine
```

```
Performing encryption speed test with 1 round,
```

```
6553600 blocks per round.
```

```
libdes swipe: 1, 15.36 sec @ 426609.98 blocks/sec  
26.04 Mbps 3.25 MBps
```

```
Bitslice-DES (iu/iu) swipe: 32, 16.98 sec @ 385960.32 blocks/sec  
23.56 Mbps 2.94 MBps
```

```
Bitslice-DES (vpu/iu) swipe: 32, 5.41 sec @ 1212215.83 blocks/sec  
73.99 Mbps 9.25 MBps
```

```
Bitslice-DES (none/iu) swipe: 32, 4.34 sec @ 1511052.99 blocks/sec  
92.23 Mbps 11.53 MBps
```

```
Bitslice-DES (iu/vpu) swipe: 128, 24.30 sec @ 269652.27 blocks/sec  
16.46 Mbps 2.06 MBps
```

```
Bitslice-DES (vpu/vpu) swipe: 128, 3.22 sec @ 2036890.67 blocks/sec  
124.32 Mbps 15.54 MBps
```

```
Bitslice-DES (none/vpu) swipe: 128, 1.38 sec @ 4761254.36 blocks/sec  
290.60 Mbps 36.33 MBps
```

D.3 Sample Output “-W”

```
[localhost:~/] % ./bitslice -W
```

32-bit machine

Performing swizzle speed test with 1 round, 6553600 blocks per round.

```
fill_data32_iu() 7.18 sec @ 913145.94 blocks/sec
    55.73 Mbps    6.97 MBps
extract_data32_iu() 6.28 sec @ 1043971.50 blocks/sec
    63.72 Mbps    7.96 MBps
fill_data32_vpu() 0.81 sec @ 8057034.60 blocks/sec
    491.76 Mbps   61.47 MBps
extract_data32_vpu() 0.89 sec @ 7385286.67 blocks/sec
    450.76 Mbps   56.35 MBps
fill_data128_iu() 11.73 sec @ 558937.00 blocks/sec
    34.11 Mbps    4.26 MBps
extract_data128_iu() 12.27 sec @ 534054.18 blocks/sec
    32.60 Mbps    4.07 MBps
fill_data128_vpu() 1.03 sec @ 6343454.05 blocks/sec
    387.17 Mbps   48.40 MBps
extract_data128_vpu() 0.86 sec @ 7653508.70 blocks/sec
    467.13 Mbps   58.39 MBps
```

D.4 Sample Output “-E”

```
[localhost:~/] % ./bitslice -E
```

32-bit machine

Performing swipe size speed test with 1 round, 6553600 blocks per round.

```
libdes          swipe: 1, 13.16 sec @ 497965.88 blocks/sec
    30.39 Mbps    3.80 MBps
Bitslice-DES (vpu/vpu) swipe: 4, 93.46 sec @ 70124.95 blocks/sec
    4.28 Mbps     0.54 MBps
Bitslice-DES (vpu/vpu) swipe: 8, 47.89 sec @ 136837.04 blocks/sec
    8.35 Mbps     1.04 MBps
Bitslice-DES (vpu/vpu) swipe: 12, 31.38 sec @ 208858.02 blocks/sec
    12.75 Mbps    1.59 MBps
Bitslice-DES (vpu/vpu) swipe: 16, 23.66 sec @ 277018.27 blocks/sec
    16.91 Mbps    2.11 MBps
Bitslice-DES (vpu/vpu) swipe: 20, 18.97 sec @ 345506.26 blocks/sec
    21.09 Mbps    2.64 MBps
Bitslice-DES (vpu/vpu) swipe: 24, 15.84 sec @ 413650.52 blocks/sec
```

25.25 Mbps	3.16 MBps		
Bitslice-DES (vpu/vpu)	swipe: 26,	14.67 sec @	446870.17 blocks/sec
27.27 Mbps	3.41 MBps		
Bitslice-DES (vpu/vpu)	swipe: 28,	13.67 sec @	479274.89 blocks/sec
29.25 Mbps	3.66 MBps		
Bitslice-DES (vpu/vpu)	swipe: 29,	13.18 sec @	497245.75 blocks/sec
30.35 Mbps	3.79 MBps		
Bitslice-DES (vpu/vpu)	swipe: 30,	12.69 sec @	516319.13 blocks/sec
31.51 Mbps	3.94 MBps		
Bitslice-DES (vpu/vpu)	swipe: 31,	12.26 sec @	534451.86 blocks/sec
32.62 Mbps	4.08 MBps		
Bitslice-DES (vpu/vpu)	swipe: 32,	11.90 sec @	550600.31 blocks/sec
33.61 Mbps	4.20 MBps		
Bitslice-DES (vpu/vpu)	swipe: 34,	11.19 sec @	585746.96 blocks/sec
35.75 Mbps	4.47 MBps		
Bitslice-DES (vpu/vpu)	swipe: 36,	10.57 sec @	619795.75 blocks/sec
37.83 Mbps	4.73 MBps		
Bitslice-DES (vpu/vpu)	swipe: 38,	10.02 sec @	653898.54 blocks/sec
39.91 Mbps	4.99 MBps		
Bitslice-DES (vpu/vpu)	swipe: 40,	9.54 sec @	686603.34 blocks/sec
41.91 Mbps	5.24 MBps		
Bitslice-DES (vpu/vpu)	swipe: 50,	7.71 sec @	849673.21 blocks/sec
51.86 Mbps	6.48 MBps		
Bitslice-DES (vpu/vpu)	swipe: 60,	6.41 sec @	1022401.54 blocks/sec
62.40 Mbps	7.80 MBps		
Bitslice-DES (vpu/vpu)	swipe: 70,	5.50 sec @	1191284.23 blocks/sec
72.71 Mbps	9.09 MBps		
Bitslice-DES (vpu/vpu)	swipe: 80,	4.87 sec @	1346524.08 blocks/sec
82.19 Mbps	10.27 MBps		
Bitslice-DES (vpu/vpu)	swipe: 100,	3.90 sec @	1678936.67 blocks/sec
102.47 Mbps	12.81 MBps		
Bitslice-DES (vpu/vpu)	swipe: 120,	3.24 sec @	2023183.75 blocks/sec
123.49 Mbps	15.44 MBps		
Bitslice-DES (vpu/vpu)	swipe: 128,	3.09 sec @	2122331.34 blocks/sec
129.54 Mbps	16.19 MBps		

D.5 Sample Output “-L”

```
[localhost:~/] % ./bitslice -L
```

```
32-bit machine
```

```
Testing integer-unit swizzling with 32 plain-data blocks:
```

```
Text #00: OK
```

```
.  
. .  
Text #31: OK  
  
Testing integer-unit swizzling with 128 plain-data blocks:  
Text #00: OK  
. .  
Text #127: OK  
Tests completed, all Tests OK.
```

D.6 Sample Output “-P”

```
[localhost:~/] % ./bitslice -P  
  
32-bit machine  
Found keys: FFFFFFFF  
Testing BITSlice Encryption:  
Text #00: OK  
. .  
Text #31: OK  
Testing BITSlice Decryption:  
Text #00: OK  
. .  
Text #31: OK  
Tests completed, all Tests OK.
```

References

- [1] Selim G. Akl. Parallel real-time computation: Sometimes quantity means quality. In *ISPAN: Proceedings of the International Symposium on Parallel Architectures, Algorithms, and Networks (I-SPAN)*. IEEE Computer Society Press, 2000.
- [2] AltivecTMtechnology programming interface manual. Technical report, Motorola, 1999.
- [3] AltivecTMtechnology programming environments manual. Technical report, Motorola, 2001.
- [4] Eli Biham. A fast new DES implementation in software. *Lecture Notes in Computer Science*, 1267, 1997.
- [5] C. Burwick, D. Coppersmith, E. D’Avignon, R. Gennaro, S. Halevi, C. Jutla, S. Matyas, L. O’Connor, M. Peyravian, D. Safford, and N. Zunic. MARS — a candidate cipher for AES. First AES conference, 1998.
- [6] R. Safavi-Naini C. Charnes, L. O’Connor, J. Pieprzyk and Y. Zheng. Further comments on the soviet encryption algorithm. Technical report, University of Wollongong Department of Computer Science, 1994.
- [7] J. Daemen and V. Rijmen. AES proposal: Rijndael. First AES conference, 1998.
- [8] Joan Daemen, Lars Knudsen, and Vincent Rijmen. The block cipher SQUARE. *Lecture Notes in Computer Science*, 1267, 1997.
- [9] Joan Daemen and Vincent Rijmen. *The Design of Rijndael*. Springer, 2002.
- [10] Hans Eberle. A high-speed DES implementation for network applications. *Lecture Notes in Computer Science*, 740:521–539, 1993.
- [11] Federal information processing standards publication. Technical Report 46-3, National Institute of Standards and Technology, 1999.

- [12] J. Orlin Grabbe. The DES algorithm illustrated. *Laissez Faire City Times*, 2(28), 1998.
- [13] M. Kwan. Reducing the gate count of bitslice DES. 2000, unpublished.
- [14] F. Thomson Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays Trees Hypercubes*. Morgan Kaufmann Publishers, Inc., 1992.
- [15] Lauren May, Lyta Penna, and Andrew Clark. An implementation of bitsliced DES on the Pentium MMXTM processor, 2000.
- [16] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1997.
- [17] E. Nahum, S. O'Malley, H. Orman, and R. Schroepel. Towards high-performance cryptographic software. Technical report, Department of Computer Science, University of Arizona, 1995.
- [18] E. Nahum, D. Yates, O. Orman, and H. Schroepel. Parallelized network security protocols. Internet Society Symposium on Network and Distributed System Security (SNDSS), San Diego, CA, February 1996.
- [19] Ian Ph.D. Ollmann. Altivec (a.k.a velocity engine). Technical report, California Institute of Technology, 2001.
- [20] Josef Pieprzyk and Leonid Tombak. Soviet encryption algorithm. Technical report, University of Wollongong Department of Computer Science, 1994.
- [21] Anderson R., E. Biham, and L. Knudsen. Serpent: a flexible block cipher with maximum assurance. First AES conference, 1998.
- [22] Ronald L. Rivest, M. J. B. Robshaw, R. Sidney, and Y. L. Yin. The RC6 block cipher. First AES conference, 1998.

- [23] B. Schneier and J. Kelsey. Unbalanced Feistel networks and block cipher design. *Lecture Notes in Computer Science*, 1039:121–144, 1996.
- [24] Bruce Schneier. *Applied Cryptography*. John Wiley & Sons, Inc., 2nd edition, 1996.
- [25] Bruce Schneier, John Kelsey, Doug Whiting, David Wagner, and Chris Hall. On the twofish key schedule. In *Selected Areas in Cryptography*, pages 27–42, 1998.
- [26] Eric C. Seidel. Tomorrow’s cryptography: Parallel computation via multiple processors, vector processing, and multi-cored chips. 2002, unpublished.
- [27] vDSP library. Technical report, Apple Computer, 2001.
- [28] Michael Welschenbach. *Cryptography in C and C++*. Springer-Verlag New York, 2001.
- [29] <http://www.darkside.com.au/bitslice/>.
- [30] <http://www.ssh.com/support/cryptography/algorithms/>.
- [31] <http://www.tropsoft.com/strongenc/des.htm>.