# POWER8 in-core Cryptography

## The Unofficial Guide

**Jeffrey Walton**
**Steven Munroe**
**Dr. William Schmidt**

# POWER8 in-core Cryptography: The Unofficial Guide

by Jeffrey Walton, Steven Munroe, and Dr. William Schmidt

Publication date 1 June 2018

# Table of Contents

# Chapter 1. Introduction

This document is a guide to using IBM's POWER8 in-core cryptography [https://www.ibm.com/developerworks/learn/security/index.html]. The purpose of the book is to document in-core cryptography more completely for developers and quality assurance personnel who wish to take advantage of the features.

POWER8 in-core cryptography includes CPU instructions to accelerate AES, SHA-256, SHA-512 and polynomial multiplication. This document includes treatments of AES, SHA-256 and SHA-512. It does not include a discussion of polynomial multiplication at the moment, but the chapter is stubbed-out (and waiting for a contributor).

The POWER8 extensions for in-core cryptography find their ancestry in the Altivec SIMD coprocessor. The POWER8 vector unit includes Vector-Scalar Extensions (VSX) and the instruction set for in-core cryptography is a part of it. You can find additional information on VSX in Chapter 7 of the IBM Power ISA Version 3.0B [https://openpowerfoundation.org/?resource_lib=power-isa-version-3-0] at the OpenPOWER Foundation website.

## Architecture

There are two POWER architectures that you will encounter as you are working on your implementation. The first is POWER7, and it is governed by ISA 2.06B documents. The second is POWER8, and it is governed by ISA 2.07B documents.

In-core cryptography requires POWER8 and ISA 2.07B support. POWER8 is the ISA that has the instructions for AES, SHA and polynomial multiplication. POWER7 provides other useful instructions, like unaligned loads and stores. If you are working with POWER8, then you have everything in POWER7 and earlier.

Note that if you are developing for Little-Endian Linux on Power, POWER7 is not available. ISA 2.07B (POWER8) support is the minimum requirement for LE systems.

The OpenPOWER Foundation [https://openpowerfoundation.org] is an open technical community based on the POWER architecture. Its mission is to create an open ecosystem, using the POWER architecture to share expertise, investment, and server-class intellectual property to serve the evolving needs of customers and industry. The Foundation publishes many technical documents for the POWER architectures in its Resource Catalog [https://openpowerfoundation.org/technical/resource-catalog/], including ISA 2.07B. Change flags in this document show the added content relative to ISA 2.06B, which predates the OpenPOWER Foundation.

## Compilers

The book does not discriminate compilers. All the samples will compile with both GCC and IBM XL C/C++. XL C/C++ is IBM's flagship compiler, and it is referred to as XLC on occasion.

The samples may compile with LLVM's Clang but it was not tested. The compile farm does not have Clang installed so we could not test it. We would like to see how well Clang performs when compared to GCC and XLC. If you encounter a problem using Clang then please report it.

The compiler you use can make a measurable difference on you program. For example, you will probably obtain different benchmark results using GCC and XLC. You will even obtain different benchmark results among versions of the same compiler. For example, GCC 7.2 is generally faster than GCC 4.8.5, and both SHA-256 and SHA-512 builtin implementations will speed up by about 2 cycles per byte (cpb) using GCC 7.

Compilers are discussed in more detail at GCC Compiler, XLC Compiler and Clang Compiler.

# Source code

The source code in the book is a mix of C and C++. The SHA-256 and SHA-512 samples were written in C++ to avoid compile errors due to the SHA API requiring 4-bit literal constants. We could not pass parameters through functions and obtain the necessary `constexpr-ness` so template parameters were used instead.

There is no source code library to download *per se*. The code is taken from Botan, Crypto ++ and OpenSSL free software projects. Some code is taken from Andy Polyakov and Cryptogams. Some code is taken from GitHub projects. And some code was written and thrown away after testing.

The code for AES and SHA was collected and placed at AES Intrinsics [https://github.com/noloader/AES-Intrinsics] and SHA Intrinsics [https://github.com/noloader/SHA-Intrinsics] GitHubs but the code is not in library format. Rather they are stand alone proof of concepts.

# Compile Farm

The book makes frequent references to `gcc112` and `gcc119` from the GCC Compile Farm. The Compile Farm offers four 64-bit PowerPC machines, and `gcc112` and `gcc119` are the POWER8 iron (the other two are POWER7 hardware). `gcc112` is a Linux PowerPC, 64-bit, little-endian machine (ppc64-le), and `gcc119` is an AIX PowerPC, 64-bit, big-endian machine (ppc64-be).

Both POWER8 machines are IBM POWER System S822 with two CPU cards. `gcc112` has 160 logical CPUs and runs at 3.4 GHz. `gcc119` has 64 logical CPUs and runs at 4.1 GHz. At 4.1 GHz and 192 GB of RAM `gcc119` is probably a contender for one of the fastest machine you will work on.

If you are a free and open software developer then you are eligible for a free GCC Compile Farm [https://cfarm.tetaneutral.net/] account. The Cfarm provides machines for different architectures, including MIPS64, Aarch64 and 64-bit PowerPC. Access is provided through SSH.

If you work on the Compile Farm then be mindful of the default GCC compiler. It is probably GCC 4.8.5, and you usually get better code generation and performance with GCC 7.2 located at `/opt/cfarm/gcc-latest`.

# Contributing

This book is free software. If you see an opportunity for improvement, an error or an omission then please submit a pull request or open a bug report.

# Organization

The book proceeds in eight parts. First, administrivia is discussed, like how to determine machine endianness and how to load and store a vector from memory. A full treatment of vector programming is its own book, but the discussion should be adequate to move on to the more interesting tasks.

Second, runtime feature detections is discussed for AIX and Linux. Runtime detection allows you to switch to a faster implementation at runtime when the hardware provides the support.

Third, AES is discussed. AES is specified in FIPS 197, Advanced Encryption Standard (AES) [https://nvlpubs.nist.gov/nistpubs/fips/nist.fips.197.pdf]. You should read the standard if you are not familiar with the block cipher.

Fourth, SHA is discussed. SHA is specified in FIPS 180-4, Secure Hash Standard (SHS) [https://nvlpubs.nist.gov/nistpubs/fips/nist.fips.180-4.pdf]. You should read the standard if you are not familiar with the hash.

Fifth, polynomial multiplication is discussed. Polynomial multiplications is important for CRC-32, CRC-32C and GCM mode of operation for AES.

Sixth, performance is discussed. The implementations are compared against C and C++ routines and assembly language routines from OpenSSL. The OpenSSL routines are high quality and written by Andy Polyakov.

Seventh, assembly language integration is discussed. Andy Polyakov dual licenses his cryptographic implementations and you can use his routines once you know how to integrate them.

Finally, performance and benchmarking is discussed. C/C++, C++ using builtins and assembly language routines are benchmarked using GCC.

# Chapter 2. Vector programming

Several topics need to be discussed to minimize trouble when using the Altivec and POWER8 extensions. They include PowerPC compilers and options, Altivec headers, machine endianness, vector datatypes, memory and alignment, and loads and stores. It is enough information to get to the point you can use AES and SHA but not much more.

Memory alignment, loads, stores and shifts will probably cause the most trouble for someone new to PowerPC vector programming. If you are new to the platform you may want to read this chapter twice. If you are experienced with the platform then you probably want to skip this chapter.

Two compilers are used for testing. The first is GCC and the second is IBM XL C/C++ (XLC). Both produce high quality code. The GCC and XLC compilers are mostly the same but accept slightly different options. LLVM's Clang was not tested because the compile farm lacks a Clang installation.

## GCC compiler

Compiling a test program with GCC will generally look like below. The important part is `-mcpu=power8` which selects the POWER8 Instruction Set Architecture (ISA). The minimum architecture required for Altivec with GCC is `-march=power4`.

```
$ g++ -mcpu=power8 -maltivec test.cxx -o test.exe
```

GCC consumes `-g` and `-O3`. If you want Position Independent Code then use `-fPIC`.

GCC uses integer math to calculate the effective memory addresses from a memory address and offset. That is, if you have an array `uint32_t vals = {0,1,2,3,4,5,6,7}`, and you call `uint32x4_p v = vec_xl(vals, 4)`, then the vector will have the value `{1,2,3,4}`. That is, GCC uses the address of `vals` and simply adds `4` to it. Effective addresses are discussed in Effective Addresses, and loads and stores are discussed in Aligned Data References.

If you work on the Compile Farm then be mindful of the default GCC compiler. It may be GCC 4.8.5 which is a bit old and unsupported. You usually enjoy better code generation and performance with a modern GCC like 7.2. A newer compiler is usually located at `/opt/cfarm/gcc-latest`.

## XLC compiler

Compiling a test program with IBM XL C/C++ will generally look like below. The important parts are the C++ compiler name of `xlC`, and `-qarch=pwr8` which selects the POWER8 ISA. The minimum architecture required for Altivec with XLC is `-qarch=pwr6`.

```
$ xlC -qarch=pwr8 -qaltivec test.cxx -o test.exe
```

XLC consumes `-g` and `-O3`. If you want Position Independent Code then use `-qpic` for XLC.

XLC uses integer math to calculate the effective memory addresses from a memory address and offset. That is, if you have an array `uint32_t vals = {0,1,2,3,4,5,6,7}`, and you call `uint32x4_p v = vec_xl(vals, 4)`, then the vector will have the value `{1,2,3,4}`. That is, XLC uses the address of `vals` and simply adds `4` to it. Effective addresses are discussed in Effective Addresses, and loads and stores are discussed in Aligned Data References.

IBM XLC switched to a LLVM front-end for the Linux compiler at version 13.1 (AIX still uses the IBM front-end). When compiling with XLC version 13.1 or higher on Linux you may need the `-qcompatmacros` compiler option. Without `-qcompatmacros` the samples fail to compile because XLC 13.1 does not define `__xlc__` or `__xlC__`. However, LLVM defines `__GNUC__` but compilation fails because the compiler cannot consume the GCC POWER8 builtins.[1]

# Clang compiler

Compiling a test program with LLVM's Clang will generally look like below. The important part is `-mcpu=power8` which selects the POWER8 Instruction Set Architecture (ISA). The minimum architecture required for Altivec with Clang is unknown. The Clang compiler was not tested because the compile farm lacks a Clang installation.

```
$ g++ -mcpu=power8 -maltivec test.cxx -o test.exe
```

Clang consumes `-g` and `-O3`. If you want Position Independent Code then use `-fPIC`.

IBM switched to an LLVM front-end for the Linux compiler at version 13.1 as discussed in XLC Compiler. Be mindful of how you use preprocessor macros when LLVM is the front-end because the compiler claims to be three different compilers — GCC, Clang and XLC.

```
$ xlc -qxlcompatmacros -qshowmacros -E test.cxx | \
  egrep -i -E 'xlc|clang|llvm|gnuc'
#define __GNUC_GNU_INLINE__ 1
#define __GNUC_MINOR__ 2
#define __GNUC_PATCHLEVEL__ 1
#define __GNUC__ 4
#define __clang__ 1
#define __clang_major__ 4
#define __clang_minor__ 0
#define __clang_patchlevel__ 1
#define __llvm__ 1
#define __xlC__ 0x0d01
#define __xlC_ver__ 0x00000601
```

The front-end change triggered LLVM Issue 38378 [http://bugs.llvm.org/show_bug.cgi?id=38378] for one project because the compiler entered GCC code paths but it cannot consume GCC POWER8 builtins.

Clang uses pointer math to calculate the effective memory addresses from a memory address and offset. That is, if you have an array `uint32_t vals = {0,1,2,3,4,5,6,7}`, and you

[1] LLVM behavior is befuddling to some people. It is the XLC compiler but fails to define `__xlc__` or `__xlC__`. Then it defines `__GNUC__` but fails to consume the GCC POWER8 builtins.

call `uint32x4_p v = vec_xl(vals, 4)`, then the vector will have the value `{4,5,6,7}`. That is, Clang uses the address of `vals` and adds `4*sizeof(uint32_t)` to it. Effective addresses are discussed in Effective Addresses, and loads and stores are discussed in Aligned Data References.

If you use the Clang compiler then be sure it is version 7.1 or above due to Bug 39704 [https://bugs.llvm.org/show_bug.cgi?id=39704]. The 39704 bug is due to Clang discarding code it believed was illegal due to an unaligned Altivec load. The code was legal because POWER7 and VSX provides the load of data using natural alignment. The code should not have been removed.

# Altivec headers

The header file required for datatypes and functions is `<altivec.h>`. You must enable Altivec using compiler options `-maltivec` or `-qaltivec` as discussed in GCC Compiler and XLC Compiler.

To support C++ projects and compilers the `__vector` keyword is used rather than `vector`. A typical Altivec include looks as shown below.

```
#if defined(__ALTIVEC__)
# include <altivec.h>
# include <altivec.h>
# undef vector
# undef pixel
# undef bool
#endif
```

The `__ALTIVEC__` preprocessor macro will be defined when using `-qaltivec` or `-maltivec` compiler options as discussed in GCC Compiler and XLC Compiler. Macros and definitions are discussed more in Preprocessor Macros.

When you need to use the vector datatypes you use the `__vector` keyword as discussed in the section Vector Datatypes.

```
/* A vector with four elements initialized to 0 */
__vector unsigned int v = {0,0,0,0};
```

# Preprocessor macros

The following preprocessor macros and defines will be encountered depending on the platform and compiler:

- `__powerpc__` and `__powerpc` on AIX
- `__powerpc__` and `__powerpc64__` on Linux
- `_ARCH_PWR3` through `_ARCH_PWR9` on AIX and Linux
- `__linux__`, `__linux` and `linux` on Linux
- `_AIX`, and `_AIX32` through `_AIX72` on AIX

- `__GNUC__` when using GCC C/C++ compiler
- `__xlc__` and `__xlC__` when using IBM XL C/C++

# Machine endianness

You will experience both little-endian and big-endian machines in the field when working with a modern PowerPC architecture. Linux is generally little-endian, while AIX is big-endian.

When writing portable source code you should check the value of preprocessor macros `__LITTLE_ENDIAN__` or `__BIG_ENDIAN__` to determine the configuration. The value of the macros `__BIG_ENDIAN__` and `__LITTLE_ENDIAN__` are defined to non-0 when in effect. Source code checking endianness should look similar to the code shown below.

```
#if __LITTLE_ENDIAN__
# pragma message "Little-endian system"
#else
# pragma message "Big-endian system"
#endif
```

The compilers can show the endian-related preprocessor macros available on a platform. Below is from GCC on `gcc112` from the compile farm, which is ppc64-le.

```
$ g++ -dM -E test.cxx | grep -i endian
#define __ORDER_LITTLE_ENDIAN__ 1234
#define _LITTLE_ENDIAN 1
#define __FLOAT_WORD_ORDER__ __ORDER_LITTLE_ENDIAN__
#define __ORDER_PDP_ENDIAN__ 3412
#define __LITTLE_ENDIAN__ 1
#define __ORDER_BIG_ENDIAN__ 4321
#define __BYTE_ORDER__ __ORDER_LITTLE_ENDIAN__
```

And the complimentary view from IBM XL C/C++ on `gcc112` from the Compile Farm, which is ppc64-le.

```
$ xlC -qshowmacros -E test.cxx | grep -i endian
#define _LITTLE_ENDIAN 1
#define __BYTE_ORDER__ __ORDER_LITTLE_ENDIAN__
#define __FLOAT_WORD_ORDER__ __ORDER_LITTLE_ENDIAN__
#define __LITTLE_ENDIAN__ 1
#define __ORDER_BIG_ENDIAN__ 4321
#define __ORDER_LITTLE_ENDIAN__ 1234
#define __ORDER_PDP_ENDIAN__ 3412
#define __VEC_ELEMENT_REG_ORDER__ __ORDER_LITTLE_ENDIAN__
```

However, below is `gcc119` from the compile farm, which is ppc64-be. It runs AIX and notice `__BYTE_ORDER__`, `__ORDER_BIG_ENDIAN__` and `__ORDER_LITTLE_ENDIAN__` are not present.

```
$ xlC -qshowmacros -E test.cxx | grep -i endian
#define __BIG_ENDIAN__ 1
```

```
#define _BIG_ENDIAN 1
#define __THW_BIG_ENDIAN__ 1
#define __HHW_BIG_ENDIAN__ 1
```

PowerPC is not the only architecture to have the `__BIG_ENDIAN__` and `__LITTLE_ENDIAN__` trap. The trap is also laid on SPARCv8, and caused BIND users on NetBSD-8 to fail to validate DNSSEC keys. The problem was the endianess tests used `__ORDER_BIG_ENDIAN__` and `__ORDER_LITTLE_ENDIAN__`. Also see DNSSEC vs netbsd-8/sparc? [https://mail-index.netbsd.org/netbsd-users/2020/04/16/msg024529.html].

# Memory allocation

System calls like `malloc` and `calloc` (and friends) are used to acquire memory from the heap. The system calls *do not* guarantee alignment to any particular boundary on all platforms. Linux generally returns a pointer that is at least 16-byte aligned on all platforms, including ARM, PPC, MIPS and x86. AIX *does not* provide the same alignment behavior [http://stackoverflow.com/q/48373188/608639].

To avoid unexpected surprises when using heap allocations you should use `posix_memalign` [http://pubs.opengroup.org/onlinepubs/009695399/functions/posix_memalign.html] to acquire heap memory aligned to a particular boundary and `free` to return it to the system.

AIX provides routines for vector memory allocation and alignment. They are `vec_malloc` and `vec_free`, and you can use them like `_mm_malloc` on Intel machines with Streaming SIMD Extensions (SSE).

# Vector datatypes

Three vector datatypes are needed for vector programming. The three types used for crypto are listed below.

- `__vector unsigned char`
- `__vector unsigned int`
- `__vector unsigned long`

`__vector unsigned char` is arranged as 16 each 8-bit bytes, and it is a typedef to `uint8x16_p`. `__vector unsigned int` is arranged as 4 each 32-bit words, and it is a typedef to `uint32x4_p`.

POWER7 and VSX added `__vector unsigned long`, and it is arranged as 2 each 64-bit double words. The typedef is `uint64x2_p`. According to the Power Architecture ELF ABI Specification, you should always use `__vector unsigned long` for portability because it is available on 32-bit and 64-bit systems. You should not use `__vector unsigned long long`.

Though POWER7 and VSX added `__vector unsigned long`, the associated 64-bit vector operations did not arrive until POWER8. That means you can load a `__vector unsigned`

`long`, but you can't add, subtract, or, or xor them until POWER8. You will be OK with crypto operations since crypto is POWER8.

The typedef naming was selected to convey the arrangement, like `32x4` and `64x2`. The trailing `_p` was selected to convey the POWER architecture and avoid collisions with ARM NEON vector data types. The suffix `_p8` (for POWER8 architecture) or `_v` (for Vector) would work just as well. You should avoid `_t` because it is reserved for the language by the C standards.

# Vector shifts

Altivec shifts and rotates are performed using *Vector Shift Left Double by Octet Immediate*. The vector shift and rotate builtin is `vec_sld` and it compiles/assembles to `vsldoi`. Both shift and rotate operate on a concatenation of two vectors. Bytes are shifted out on the left and shifted in on the right. The instructions need an integral constant in the range 0 - 15, inclusive.

Vector shifts and rotates perform as expected on big-endian machines. Little-endian machines require special handling to produce correct results. The catch is, IBM manuals don't tell you about it [http://www.ibm.com/support/knowledgecenter/SSXVZZ_13.1.4/com.ibm.xlcpp1314.lelinux.doc/compiler_ref/vec_sld.html].

The issue is shifts and rotates are endian sensitive [http://stackoverflow.com/q/46341923/608639] and you have to use `16-n` integrals and swap vector arguments on little-endian systems. The C++ source code provides the following template function to compensate for the little-endian behavior.

```
template <unsigned int N, class T>
T VecShiftLeft(const T val1, const T val2)
{
#if __LITTLE_ENDIAN__
    enum {R = (16-N)&0xf};
    return vec_sld(val2, val1, R);
#else
    enum {R = N&0xf};
    return vec_sld(val1, val2, R);
#endif
}
```

A `VecRotateLeft` would be similar to the code below, if needed. Rotate is a special case of shift where both vector arguments are the same value.

```
template <unsigned int N, class T>
T VecRotateLeft(const T val)
{
#if __LITTLE_ENDIAN__
    enum {R = (16-N)&0xf};
    return vec_sld(val, val, R);
#else
    enum {R = N&0xf};
    return vec_sld(val, val, R);
#endif
```

```
}
```

# Vector permutes

Vector permutes allow you to rearrange elements in a vector. The values to be permuted can be in any arrangement like 64x2 or 32x4, but the mask is always an octet mask using an 8x16 arrangement.

The Altivec permute is very powerful and it stands out among architectures like ARM, Aarch64 and x86. The instruction allows you to select elements from two source vectors. When an index in the mask is in the range `[0,15]` then elements from the first vector are selected, and index values in the the range `[16,31]` select elements from the second vector.

As an example, suppose you have a big-endian byte array like a message to be hashed using SHA-256. SHA operates on 32-bit words so the message needs a shuffle on little-endian systems. The code to perform the permute on a little-endian machine would look like below.

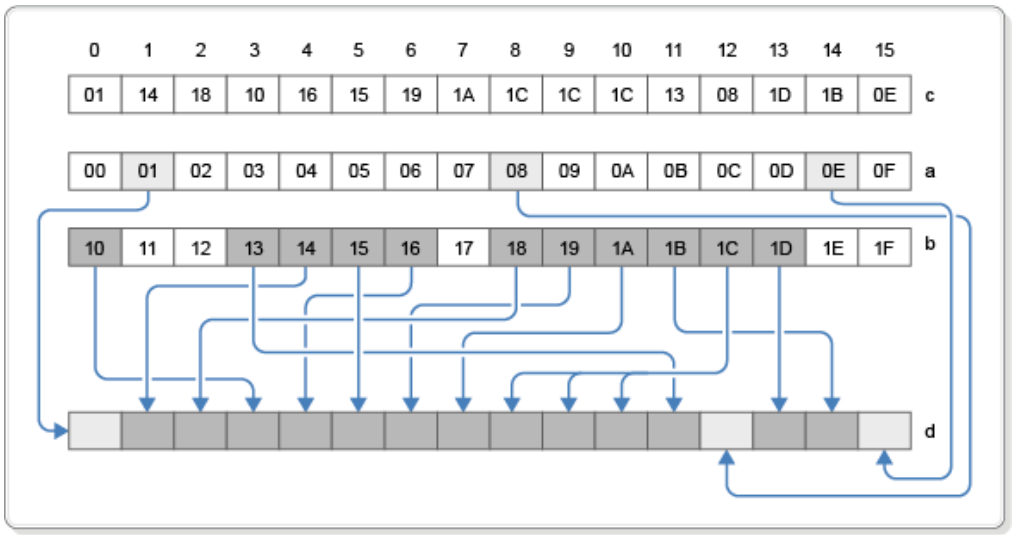
```
uint32x4_p msg = vec_ld(/*load from memory*/);
uint8x16_p mask = {3,2,1,0, 7,6,5,4, 11,10,9,8, 15,14,13,12};
msg = vec_perm(msg, msg, mask);
```

The previous example only needed one vector so it used `msg` twice in the call to `vec_perm`. The Altivec code is similar to `_mm_shuffle_epi8` on Intel machines. An example that interleaves two different vectors is shown below.

```
uint32x4_p a = { 0,  0,  0,  0};  // All 0 bits
uint32x4_p b = {-1, -1, -1, -1};  // All 1 bits
uint8x16_p m = {0,1,2,3, 16,17,18,19, 4,5,6,7, 20,21,22,23};
uint32x4_p c = vec_perm(a, b, m);
```

After the code above executes the vector c will have the value `{0, -1, 0, -1}`.

Below is the image IBM provides for the `vec_perm` documentation. The IBM example shows `d = vec_perm(a, b, c)`. The light gray blocks in vector d are from the first vector, and dark gray blocks in vector d are from the second vector.



# Effective addresses

Some vector operations, like loads and stores, can be sensitive to the alignment of a memory address. Operations like `vec_ld` and `vec_st` are sensitive, and the documentation clearly states it.

The effective address is a simple sum consisting of the memory address plus the offset into the address with the result rounded down to a multiple of 16. Effective addresses follow integer arithmetic and not pointer arithmetic.

You can calculate an effective address using the following code. Notice the bottom 4 bits are masked after calculating the sum to yield a multiple of 16.

```
uintptr_t maddr = (uintptr_t)mem_addr;
uintptr_t mask = ~(uintptr_t)0xf;
uintptr_t eaddr = (maddr+offset) & mask;
```

`vec_ld` takes a pointer and an offset to load a value into a VSX register. Each of the following yield the same VSX register value because the effective addresses are the same. (Old x86 programmers should reminisce on segmented memory).

```
uint8_t* ptr1 = 0x401000;
int off1 = 32;
uint8x16_p r1 = vec_ld(off1, ptr1);

uint8_t* ptr2 = 0x401010;
int off2 = 16;
uint8x16_p r2 = vec_ld(off2, ptr2);
```

The following also yields the same VSX register value because the effective address is the same. If you truly wanted to load 4 bytes beyond `ptr` then you loaded the wrong value because `(0x401010+4)&0xfffffff0 = 0x401010`.

```
uint8_t* ptr1 = 0x401000;
int off1 = 16;
uint8x16_p r1 = vec_ld(off1, ptr1);

uint8_t* ptr2 = 0x401010;
int off2 = 4;
uint8x16_p r2 = vec_ld(off2, ptr2);
```

The application of effective addresses are discussed more below in Aligned Data References and Unaligned Data References.

# Aligned data references

Altivec loads and stores have traditionally been performed using `vec_ld` and `vec_st` since at least the POWER4 days in the early 2000s. `vec_ld` and `vec_st` are sensitive to alignment of the effective address. Effective addresses were discussed in Effective Addresses.

Altivec *does not* raise a `SIGBUS` to indicate a misaligned load or store. Instead, the bottom 4 bits of the sum `address+offset` are masked-off and then the memory at the effective address is loaded.

You can use the Altivec loads and stores when you *control* buffers and ensure they are 16-byte aligned, like an AES key schedule table. Otherwise just use unaligned loads and stores to avoid trouble.

The C/C++ code to perform a load using `vec_ld` should look similar to below. Notice the `assert` to warn you of problems in debug builds.

```
template <class T>
uint32x4_p VecLoad(const T* mem_addr, int offset)
{
#ifndef NDEBUG
    uintptr_t maddr = (uintptr_t)mem_addr;
    uintptr_t mask = ~(uintptr_t)0xf;
    uintptr_t eaddr = (maddr+offset) & mask;
    assert(maddr == eaddr);
#endif

    return (uint32x4_p)vec_ld(offset, mem_addr);
}
```

The C/C++ code to perform a store using `vec_st` should look similar to below.

```
template <class T>
void VecStore(const uint32x4_p val, T* mem_addr, int offset)
{
#ifndef NDEBUG
    uintptr_t maddr = (uintptr_t)mem_addr;
    uintptr_t mask = ~(uintptr_t)0xf;
    uintptr_t eaddr = (maddr+offset) & mask;
    assert(maddr == eaddr);
#endif

    vec_st((uint8x16_p)val, offset, mem_addr);
}
```

# Unaligned data references

POWER7 and VSX introduced loads and stores for 32-bit and 64-bit datatypes that avoid the 16-byte aligned memory requirements. POWER9 introduced loads and stores for 8-bit and 16-bit datatypes. In this context "unaligned" means alignment less than 16-bytes required by Altivec, but alignment still must be natural. While POWER9 allows you to load a byte array, POWER7 and POWER8 limits you to 32-bit and 64-bit arrays. If you have a byte array on POWER7, then it must be at least 32-bit or 64-bit aligned or you have to use the Altivec load.

The preferred builtin functions for 32-bit and 64-bit datatype loads and stores are `vec_xl` and `vec_xst`. The builtins are available on all currently supported versions of GCC and XLC.

However, older versions of GCC such as those installed on many enterprise Linux distributions do not supply them. For compatibility with these older compilers you may use `vec_vsx_ld` and `vec_vsx_st` for GCC.

You should use the POWER7 and VSX loads and stores whenever you *do not control* buffers or their alignments, like a message in a buffer supplied by the user.

The C/C++ code to perform a load using `vec_xl` and `vec_vsx_ld` should look similar to below. The function name has a u added to indicate unaligned.

```
template <class T>
uint32x4_p VecLoadu(const T* mem_addr, int offset)
{
#if defined(__xlc__) || defined(__xlC__)
    return (uint32x4_p)vec_xl(offset, mem_addr);
#else
    return (uint32x4_p)vec_vsx_ld(offset, mem_addr);
#endif
}
```

The C/C++ code to perform a store using `vec_xst` and `vec_vsx_st` should look similar to below.

```
template <class T>
void VecStoreu(const uint32x4_p val, T* mem_addr, int offset)
{
#if defined(__xlc__) || defined(__xlC__)
    vec_xst((uint8x16_p)val, offset, mem_addr);
#else
    vec_vsx_st((uint8x16_p)val, offset, mem_addr);
#endif
}
```

If your code will only be compiled with supported compilers, you may simplify it to use the `vec_xl` and `vec_xst` variants for both XLC and GCC.

# Vector dereferences

The OpenPOWER ELF V2 ABI Specification [https://openpowerfoundation.org/?resource_lib=64-bit-elf-v2-abi-specification-power-architecture], version 1.4, incorrectly states that accessing vectors on Power should preferably be done with vector pointers and the dereference `operator*`. However, this is only permitted for aligned vector references. Examples in Chapter 6 of the ABI document show use of casting operations that represent undefined behavior according to the C standard. An errata document that corrects the ABI may be found at the OpenPOWER Foundation website [https://openpowerfoundation.org/?resource_lib=openpower-elfv2-errata-elfv2-abi-version-1-4]. Subsequent sections describe the proper way to use loads and stores of aligned and unaligned data.

# Chapter 3. Runtime features

Runtime feature detections allows code to switch to a faster implementation when the hardware permits. This chapter shows you how to determine POWER8 cryptography availability at runtime on AIX and Linux PowerPC platforms.

## Strategy

The strategy to detect availability of in-core cryptography on POWER processors is check for ISA 2.07 or above. Cryptography is an ISA 2.07 and POWER8 requirement, and the cryptography support cannot be disgorged.

AIX systems should check for POWER8. AIX does not provide separate bits for AES, SHA and polynomial multiplies. The ISA level signals the availability of the cryptography on AIX. Linux supplies separate bits for ISA 2.07 and vector cryptography, but you only need to check the ISA level.

There is no need to perform `SIGILL` probes on AIX or newer Linux systems. If you are using older versions of Glibc or Linux kernel then you may have to fallback to `SIGILL` probes. Older versions include Glibc 2.24 and Linux kernel 4.08 (and earlier).

## AIX features

The AIX system header `<systemcfg.h>` defines the `_system_configuration` structure that identifies system characteristics. The header also provides macros to access various fields of the structure. Runtime code to perform the POWER8 cryptography check should look similar to below.

```
#include <sys/systemcfg.h>

#ifndef __power_7_andup
# define __power_7_andup() 0
#endif

#ifndef __power_8_andup
# define __power_8_andup() 0
#endif

bool HasPower7()
{
    if (__power_7_andup() != 0)
        return true;
    return false;
}

bool HasPower8()
{
```

```
    if (__power_8_andup() != 0)
        return true;
    return false;
}

bool HasCrypto()
{
    if (__power_8_andup() != 0)
        return true;
    return false;
}
```

You *should not* use the `__power_vsx()` macro to detect in-core cryptography availability. Though cryptography is implemented in the VSX unit, the VSX unit is available in POWER7 and above.

OpenSSL uses the following on AIX to test for cryptography availability in `crypto/ppccap.c` [https://github.com/openssl/openssl/blob/master/crypto/ppccap.c]. The project effectively re-implements the `__power_8_andup()` macro.

```
/* POWER8 and later */
if (__power_set(0xffffffffU<<16))
    OPENSSL_ppccap_P |= PPC_CRYPTO207;
```

# Linux features

Some versions of Glibc and the kernel provide ELF auxiliary vectors with system information. `AT_HWCAP2` will show the `vcrypto` flag when in-core cryptography is available. This is guaranteed for the following little-endian Linux distributions:

- Ubuntu 14.04 and later
- SLES 12 and later
- RHEL 7 and later

Below is a screen capture using the loader's diagnostics to print the auxiliary vector for the `/bin/true` program on `gcc112`.

```
$ LD_SHOW_AUXV=1 /bin/true
AT_DCACHEBSIZE:  0x80
AT_ICACHEBSIZE:  0x80
AT_UCACHEBSIZE:  0x0
AT_SYSINFO_EHDR: 0x3fff877c0000
AT_HWCAP:        ppcle true_le  archpmu vsx arch_2_06 dfp ic_snoop
                 smt mmu fpu altivec ppc64 ppc32
AT_PAGESZ:       65536
AT_CLKTCK:       100
AT_PHDR:         0x10000040
AT_PHENT:        56
AT_PHNUM:        9
```

```
AT_BASE:         0x3fff877e0000
AT_FLAGS:        0x0
AT_ENTRY:        0x1000145c
AT_UID:          10455
AT_EUID:         10455
AT_GID:          10455
AT_EGID:         10455
AT_SECURE:       0
AT_RANDOM:       0x3fffeaeaa872
AT_HWCAP2:       vcrypto tar isel ebb dscr htm arch_2_07
AT_EXECFN:       /bin/true
AT_PLATFORM:     power8
AT_BASE_PLATFORM:power8
```

Linux systems with Glibc version 2.16 can use `getauxval` to determine CPU features. However, defines like `PPC_FEATURE2_ARCH_2_07` and `PPC_FEATURE2_VEC_CRYPTO` require Glibc 2.24. Runtime code to perform the check should look similar to below. The defines below were taken from the Linux kernel's cputable.h [https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/arch/powerpc/include/asm/cputable.h].

```
#ifndef AT_HWCAP
# define AT_HWCAP 16
#endif
#ifndef AT_HWCAP2
# define AT_HWCAP2 26
#endif
#ifndef PPC_FEATURE_ARCH_2_06
# define PPC_FEATURE_ARCH_2_06    0x00000100
#endif
#ifndef PPC_FEATURE2_ARCH_2_07
# define PPC_FEATURE2_ARCH_2_07   0x80000000
#endif
#ifndef PPC_FEATURE2_VEC_CRYPTO
# define PPC_FEATURE2_VEC_CRYPTO  0x02000000
#endif

bool HasPower7()
{
    if (getauxval(AT_HWCAP) & PPC_FEATURE_ARCH_2_06 != 0)
        return true;
    return false;
}

bool HasPower8()
{
    if (getauxval(AT_HWCAP2) & PPC_FEATURE2_ARCH_2_07 != 0)
        return true;
    return false;
}
```

```
bool HasCrypto()
{
    if (getauxval(AT_HWCAP2) & PPC_FEATURE2_VEC_CRYPTO != 0)
        return true;
    return false;
}
```

# L1 Data Cache

The L1 data cache line size is an important security parameter that can be used to avoid leaking information through timing attacks. IBM POWER System S822, like `gcc112` and `gcc119`, have a 128-byte L1 data cache line size.

`gcc119` runs AIX and L1 data cache line size can be queried as shown below.

```
#include <sys/systemcfg.h>

int cacheLineSize = getsystemcfg(SC_L1C_DLS);
if (cacheLineSize) <= 0)
    cacheLineSize = DEFAULT_L1_CACHE_LINE_SIZE;
```

`gcc112` runs Linux and L1 data cache line size can be queried as shown below. However, the call requires Glibc 2.26 and Linux kernel 4.10.

```
#include <unistd.h>

int cacheLineSize = sysconf(_SC_LEVEL1_DCACHE_LINESIZE);
if (cacheLineSize) <= 0)
    cacheLineSize = DEFAULT_L1_CACHE_LINE_SIZE;
```

It is important to check the return value from `sysconf` on Linux. CentOS 7.4 on `gcc112` returns 0 for the query because Glibc is version 2.17 and the Linux kernel is version 3.10. In addition to -1, you should consider a return value of 0 as failure.

You should have a fallback strategy that includes a sane default set for `DEFAULT_L1_CACHE_LINE_SIZE` because Glibc does not return a failure. On 32-bit systems you can usually use 32-bytes as a default, and on 64-bit systems you can usually use 64-bytes as a default.

Returning success with a value of 0 for an unimplemented `sysconf` parameter appears to be a Glibc bug. Also see sysconf and _SC_LEVEL1_DCACHE_LINESIZE returns 0? [https://lists.centos.org/pipermail/centos/2017-September/166236.html] on the CentOS mailing list and Issue 14599: sysconf(_SC_LEVEL1_DCACHE_LINESIZE) returns 0 instead of 128 [https://bugs.centos.org/view.php?id=14599] in the CentOS issue tracker.

# Chapter 4. Advanced Encryption Standard

AES is the Advanced Encryption Standard. AES is specified in FIPS 197, Advanced Encryption Standard (AES) [https://nvlpubs.nist.gov/nistpubs/fips/nist.fips.197.pdf]. You should read the standard if you are not familiar with the block cipher.

GCC and XL C/C++ use different data types and intrinsics to perform AES. GCC uses a 64x2 arrangement and IBM XL C/C++ uses a 8x16 arrangement. GCC uses `__builtin_crypto_vcipher`, `__builtin_crypto_vcipherlast`, `__builtin_crypto_vncipher` and `__builtin_crypto_vncipherlast` intrinsics. IBM XL C/C++ uses `__vcipher`, `__vcipherlast`, `__vncipher` and `__vncipherlast` intrinsics.

POWER8 offers instructions to perform encryption and decryption only. The ISA does not supply instructions that assist in key generation, like Intel's `AESKEYGENASSIST`.

Finally the code below is available online at AES Intrinsics [https://github.com/noloader/AES-Intrinsics]. The GitHub provides accelerated AES for Intel, ARMv8 and POWER8.

## Strategy

The strategy to perform AES encryption and decryption is straight forward. First the subkey or round key table is created based on the user key. The round keys are stored in big-endian format so a swap is avoided when loading a round key. Second, the message is loaded into the AES state array and an endian swap is performed as required. Third the the AES encryption or decryption round function is applied to the state array the required number of times. Each application of the round function is accompanied by a loading of a subkey. Finally the encrypted or decrypted message is stored after performing an endian swap as needed.

## Endianness

The AES hardware operates in big-endian mode. On little-endian systems like `gcc112` you have to convert from little-endian to big-endian during loads and stores. The code to perform the conversion is shown below. Also recall from Unaligned Data References POWER7 provides unaligned loads and stores so POWER8 has them available.

```
uint8x16_p VecReverse8x16(const uint8x16_p src)
{
    uint8x16_p mask = {15,14,13,12, 11,10,9,8, 7,6,5,4, 3,2,1,0};
    return vec_perm(src, src, mask);
}

uint64x2_p VecReverse64x2(const uint64x2_p src)
{
    uint8x16_p mask = {15,14,13,12, 11,10,9,8, 7,6,5,4, 3,2,1,0};
    uint8x16_p val = (uint8x16_p) src;
```

```
    return (uint64x2_p)vec_perm(val, val, mask);
}

uint8x16_p VecLoad8x16(const uint8_t src[16])
{
#if defined(__xlc__) || defined(__xlC__)
    return vec_xl_be(0, (uint8_t*)src);
#else
# if __LITTLE_ENDIAN__
    return VecReverse8x16(vec_vsx_ld(0, src));
# else
    return vec_vsx_ld(0, src);
# endif
#endif
}

void VecStore8x16(const uint8x16_p src, uint8_t dest[16])
{
#if defined(__xlc__) || defined(__xlC__)
    vec_xst_be(src, 0, (uint8_t*)dest);
#else
# if __LITTLE_ENDIAN__
    vec_vsx_st(VecReverse8x16(src), 0, dest);
# else
    vec_vsx_st(src, 0, dest);
# endif
#endif
}

uint64x2_p VecLoad64x2(const uint8_t src[16])
{
#if defined(__xlc__) || defined(__xlC__)
    return (uint64x2_p)vec_xl_be(0, (uint8_t*)src);
#else
# if __LITTLE_ENDIAN__
    return (uint64x2_p)VecReverse8x16(vec_vsx_ld(0, src));
# else
    return (uint64x2_p)vec_vsx_ld(0, src);
# endif
#endif
}

void VecStore64x2(const uint64x2_p src, uint8_t dest[16])
{
#if defined(__xlc__) || defined(__xlC__)
    vec_xst_be((uint8x16_p)src, 0, (uint8_t*)dest);
#else
# if __LITTLE_ENDIAN__
    vec_vsx_st(VecReverse8x16((uint8x16_p)src), 0, dest);
```

```
# else
    vec_vsx_st((uint8x16_p)src, 0, dest);
# endif
#endif
}
```

# Functions

GCC and IBM XL C/C++ uses different intrinsics and different datatypes for encryption and decryption. GCC uses a 64x2 vector arrangement while IBM XL CC++ uses a 8x16 arrangement. An intrinsics based implementation should wrap two functions for encryption and two functions for decryption.

For the encryption operation POWER8 provides a standard round function and a function to encrypt the last round. Source code should look similar to below.

```
template <class T>
T VecEncrypt(const T state, const T rkey)
{
#if defined(__xlc__) || defined(__xlC__)
    uint8x16_p s = (uint8x16_p)state;
    uint8x16_p k = (uint8x16_p)rkey;
    return (T)__vcipher(s, k);
#else
    uint64x2_p s = (uint64x2_p)state;
    uint64x2_p k = (uint64x2_p)rkey;
    return (T)__builtin_crypto_vcipher(s, k);
#endif
}

template <class T>
T VecEncryptLast(const T state, const T rkey)
{
#if defined(__xlc__) || defined(__xlC__)
    uint8x16_p s = (uint8x16_p)state;
    uint8x16_p k = (uint8x16_p)rkey;
    return (T)__vcipherlast(s, k);
#else
    uint64x2_p s = (uint64x2_p)state;
    uint64x2_p k = (uint64x2_p)rkey;
    return (T)__builtin_crypto_vcipherlast(s, k);
#endif
}
```

And the corresponding decryption functions are shown below.

```
template <class T>
T VecDecrypt(const T state, const T rkey)
{
```

```
#if defined(__xlc__) || defined(__xlC__)
    uint8x16_p s = (uint8x16_p)state;
    uint8x16_p k = (uint8x16_p)rkey;
    return (T)__vncipher(s, k);
#else
    uint64x2_p s = (uint64x2_p)state;
    uint64x2_p k = (uint64x2_p)rkey;
    return (T)__builtin_crypto_vncipher(s, k);
#endif
}

template <class T>
T VecDecryptLast(const T state, const T rkey)
{
#if defined(__xlc__) || defined(__xlC__)
    uint8x16_p s = (uint8x16_p)state;
    uint8x16_p k = (uint8x16_p)rkey;
    return (T)__vncipherlast(s, k);
#else
    uint64x2_p s = (uint64x2_p)state;
    uint64x2_p k = (uint64x2_p)rkey;
    return (T)__builtin_crypto_vncipherlast(s, k);
#endif
}
```

# Golden key

FIPS 197 Appendix B provides a user key expanded into round keys. We refer to it as the "golden key" and it allows us to independently test round key derivation, encryption and decryption. The sections AES Encryption and AES Decryption use the golden key to simplify the discussions.

Appendix B provides two key parameters. The first is the AES key supplied by the user. The second is the expanded subkey or round key table. Below is the user key supplied by Appendix B.

```
const uint8_t key[16] = {
     0x32, 0x43, 0xf6, 0xa8, 0x88, 0x5a, 0x30, 0x8d,
     0x31, 0x31, 0x98, 0xa2, 0xe0, 0x37, 0x07, 0x34
};
```

The round keys for AES-128 are as follows. Since we control the round key buffer we can make it aligned. The aligned loads will save a tiny amount of time during each load of a round key.

```
__attribute__((aligned(16)))
const uint8_t subkeys[10][16] = {
     {0xA0, 0xFA, 0xFE, 0x17, 0x88, 0x54, 0x2c, 0xb1,
        0x23, 0xa3, 0x39, 0x39, 0x2a, 0x6c, 0x76, 0x05},
     {0xF2, 0xC2, 0x95, 0xF2, 0x7a, 0x96, 0xb9, 0x43,
        0x59, 0x35, 0x80, 0x7a, 0x73, 0x59, 0xf6, 0x7f},
```

```
      {0x3D, 0x80, 0x47, 0x7D, 0x47, 0x16, 0xFE, 0x3E,
         0x1E, 0x23, 0x7E, 0x44, 0x6D, 0x7A, 0x88, 0x3B},
      {0xEF, 0x44, 0xA5, 0x41, 0xA8, 0x52, 0x5B, 0x7F,
         0xB6, 0x71, 0x25, 0x3B, 0xDB, 0x0B, 0xAD, 0x00},
      {0xD4, 0xD1, 0xC6, 0xF8, 0x7C, 0x83, 0x9D, 0x87,
         0xCA, 0xF2, 0xB8, 0xBC, 0x11, 0xF9, 0x15, 0xBC},
      {0x6D, 0x88, 0xA3, 0x7A, 0x11, 0x0B, 0x3E, 0xFD,
         0xDB, 0xF9, 0x86, 0x41, 0xCA, 0x00, 0x93, 0xFD},
      {0x4E, 0x54, 0xF7, 0x0E, 0x5F, 0x5F, 0xC9, 0xF3,
         0x84, 0xA6, 0x4F, 0xB2, 0x4E, 0xA6, 0xDC, 0x4F},
      {0xEA, 0xD2, 0x73, 0x21, 0xB5, 0x8D, 0xBA, 0xD2,
         0x31, 0x2B, 0xF5, 0x60, 0x7F, 0x8D, 0x29, 0x2F},
      {0xAC, 0x77, 0x66, 0xF3, 0x19, 0xFA, 0xDC, 0x21,
         0x28, 0xD1, 0x29, 0x41, 0x57, 0x5c, 0x00, 0x6E},
      {0xD0, 0x14, 0xF9, 0xA8, 0xC9, 0xEE, 0x25, 0x89,
         0xE1, 0x3F, 0x0c, 0xC8, 0xB6, 0x63, 0x0C, 0xA6}
};
```

# Key schedule

TODO. We don't have optimized code for key scheduling. Use Paulo Barreto's code to generate the key table in C/C++. It is available on the internet.

A brief discussion of an POWER8 optimized key schedule can be found at POWER8 in-core cryptography [https://www.ibm.com/developerworks/library/se-power8-in-core-cryptography/index.html].

# Encryption

AES encryption consists of three steps. First, the user's message is loaded into a state buffer. On little-endian machines the byte order will be reversed. Second, a round key is loaded and the AES round function is applied. The second part is repeated a required number of times. For example, AES with 128-bit key applies the round function 10 times. The third part stores the result of encrypting the state, which is the encrypted block. On little-endian machines the byte order will be reversed.

**Part 1.** Load the user message into the state vector. `VecLoad64x2` swaps endianness as required. The 64x2 arrangement tells this is a GCC code path.

```
uint64x2_p s = VecLoad64x2(input);
uint64x2_p k = VecLoad64x2(key);
s = VecXor(s, k);
```

**Part 2.** Load a subkey and encrypt the state buffer. The round key does not need an endian swap. Lather, rinse and repeat the required number of times.

In the code below remember that `subkeys` is `subkeys[10][16]`. The expression `subkeys[i]` is a byte pointer and indexes into the i-th 16-byte round key.

```
k = VecLoad64x2(subkeys[0]);
s = VecEncrypt(s, k);

k = VecLoad64x2(subkeys[1]);
s = VecEncrypt(s, k);

k = VecLoad64x2(subkeys[2]);
s = VecEncrypt(s, k);

...

k = VecLoad64x2(subkeys[7]);
s = VecEncrypt(s, k);

k = VecLoad64x2(subkeys[8]);
s = VecEncrypt(s, k);

k = VecLoad64x2(subkeys[9]);
s = VecEncryptLast(s, k);
```

**Part 3.** Store the new state which is the encrypted block. `VecStore64x2` swaps endianness as required.

```
VecStore64x2(s, output);
```

The AES-128 code shown above demonstrates a GCC code path using the 64x2 arrangement. Below is the IBM XL C/C++ code path using an 8x16 arrangement. In the code below remember that `subkeys` is `subkeys[10][16]`. The expression `subkeys[i]` is a byte pointer and indexes into the i-th 16-byte round key.

```
uint8x16_p s = VecLoad8x16(input);
uint8x16_p k = VecLoad8x16(key);
s = VecXor(s, k);

k = VecLoad8x16(subkeys[0]);
s = VecEncrypt(s, k);

k = VecLoad8x16(subkeys[1]);
s = VecEncrypt(s, k);

k = VecLoad8x16(subkeys[2]);
s = VecEncrypt(s, k);

...

k = VecLoad8x16(subkeys[7]);
s = VecEncrypt(s, k);

k = VecLoad8x16(subkeys[8]);
s = VecEncrypt(s, k);
```

```
k = VecLoad8x16(subkeys[9]);
s = VecEncryptLast(s, k);

VecStore8x16(s, output);
```

# Decryption

AES decryption is the reverse operation of AES encryption. There are three parts as with AES decryption. First, the encrypted message is loaded into a state buffer. The message is endian swapped as required. The second part loads a subkey and applies the AES inverse round function. The second part is repeated a required number of times. For example, AES with 128-bit key applies the inverse round function 10 times. The third part stores the result of decrypting the state, which is the decrypted block. The decrypted message is endian swapped as required.

AES decryption has two minor differences from the encryption algorithm. First the round keys are iterated in reverse order. Second, the user or master key is used last instead of first.

The code below demonstrates AES-128 using the GCC code path. GCC uses the 64x2 arrangement. In the code below remember that `subkeys` is `subkeys[10][16]`. The expression `subkeys[i]` is a byte pointer and indexes into the i-th 16-byte round key.

```
uint64x2_p s = VecLoad64x2(input);
uint64x2_p k = VecLoad64x2(subkeys[9]);
s = VecXor(s, k);

k = VecLoad64x2(subkeys[8]);
s = VecDecrypt(s, k);

k = VecLoad64x2(subkeys[7]);
s = VecDecrypt(s, k);

k = VecLoad64x2(subkeys[6]);
s = VecDecrypt(s, k);

...

k = VecLoad64x2(subkeys[1]);
s = VecDecrypt(s, k);

k = VecLoad64x2(subkeys[0]);
s = VecDecrypt(s, k);

k = VecLoad64x2(key);
s = VecDecryptLast(s, k);

VecStore8x16(s, output);
```

As with AES encryption there is a different code path for IBM XL C/C++ using the 8x16 datatypes. The code below shows XL C/C++ decryption using the 8x16 datatype. In the code below remember that `subkeys` is `subkeys[10][16]`. The expression `subkeys[i]` is a byte pointer and indexes into the i-th 16-byte round key.

```
uint8x16_p s = VecLoad8x16(input);
uint8x16_p k = VecLoad8x16(subkeys[9]);
s = VecXor(s, k);

k = VecLoad8x16(subkeys[8]);
s = VecDecrypt(s, k);

k = VecLoad8x16(subkeys[7]);
s = VecDecrypt(s, k);

k = VecLoad8x16(subkeys[6]);
s = VecDecrypt(s, k);

...

k = VecLoad8x16(subkeys[1]);
s = VecDecrypt(s, k);

k = VecLoad8x16(subkeys[0]);
s = VecDecrypt(s, k);

k = VecLoad8x16(key);
s = VecDecryptLast(s, k);

VecStore8x16(s, output);
```

# Performance

The code in AES Encryption and AES Decryption provides the basic AES algorithms. They will perform well when compared to C/C++ but there is room for improvement. You can improve the code to run closer to 1 to 2 cycle-per-byte (cpb) by processing multiple blocks at a time.

Experimentation shows 6 or 8 blocks at a time is a good place to be. Crypto++ processes 6 blocks at a time while Botan processes 8 blocks at a time. The Linux kernel processes 12 blocks at a time for some POWER8 algorithms.

The code below processes 16*8 or 128-bytes of data at a time using the GCC code path. The IBM code path would be similar.

In the code below remember that `subkeys` is `subkeys[10][16]`. The expression `subkeys[i]` is a byte pointer and indexes into the i-th 16-byte round key.

```
uint64x2_p k = VecLoad64x2(key);
```

```
uint64x2_p s0 = VecLoad64x2(input+0);
uint64x2_p s1 = VecLoad64x2(input+16);
uint64x2_p s2 = VecLoad64x2(input+32);
uint64x2_p s3 = VecLoad64x2(input+48);
uint64x2_p s4 = VecLoad64x2(input+64);
uint64x2_p s5 = VecLoad64x2(input+80);
uint64x2_p s6 = VecLoad64x2(input+96);
uint64x2_p s7 = VecLoad64x2(input+112);

s0 = VecXor(s0, k);
s1 = VecXor(s1, k);
s2 = VecXor(s2, k);
s3 = VecXor(s3, k);
s4 = VecXor(s4, k);
s5 = VecXor(s5, k);
s6 = VecXor(s6, k);
s7 = VecXor(s7, k);

for (size_t i=0; i<rounds-1; ++i)
{
    k = VecLoad64x2(subkeys[i]);
    s0 = VecEncrypt(s0, k);
    s1 = VecEncrypt(s1, k);
    s2 = VecEncrypt(s2, k);
    s3 = VecEncrypt(s3, k);
    s4 = VecEncrypt(s4, k);
    s5 = VecEncrypt(s5, k);
    s6 = VecEncrypt(s6, k);
    s7 = VecEncrypt(s7, k);
}

k = VecLoad64x2(subkeys[rounds-1]);
s0 = VecEncryptLast(s0, k);
s1 = VecEncryptLast(s1, k);
s2 = VecEncryptLast(s2, k);
s3 = VecEncryptLast(s3, k);
s4 = VecEncryptLast(s4, k);
s5 = VecEncryptLast(s5, k);
s6 = VecEncryptLast(s6, k);
s7 = VecEncryptLast(s7, k);

VecStore64x2(s0, output+0);
VecStore64x2(s1, output+16);
VecStore64x2(s2, output+32);
VecStore64x2(s3, output+48);
VecStore64x2(s4, output+64);
VecStore64x2(s5, output+80);
VecStore64x2(s6, output+96);
VecStore64x2(s7, output+112);
```

# Chapter 5. Secure Hash Standard

SHA is the Secure Hash Standard. SHA is specified in FIPS 180-4, Secure Hash Standard (SHS) [https://nvlpubs.nist.gov/nistpubs/fips/nist.fips.180-4.pdf]. You should read the standard if you are not familiar with the hash family.

The code below is available online at SHA Intrinsics [https://github.com/noloader/SHA-Intrinsics]. The GitHub provides accelerated SHA for Intel, ARMv8 and POWER8.

## Strategy

SHA provides a lot of freedom to an implementation. You can approach your SHA implementation in several ways, but most of them will result in an under-performing SHA. This section provides one of the strategies for a better performing implementation.

The first design element is to perform everything in vector registers. The only integer operations should be reading 2 longs or 4 integers from memory during a load, and writing 2 longs or 4 integers after the round during a store.

Second, when you need an integer for a calculation you will shift it out from a vector register to another vector register using `vec_sld`. Most of the time you only care about element 0 in a vector register, and the remainder of elements are "don't care" elements.

Third, don't maintain a full `W[64]` or `W[80]` table. Use `X[16]` instead, and transform each element in-place using a rolling strategy.

Fourth, the eight working variables `{A,B,C,D,E,F,G,H}` each get their own vector register. The one you care about is located at element 0, the remainder of the elements in the vector are "don't care" elements.

It does not matter if you rotate the working variables `{A,B,C,D,E,F,G,H}` in the caller or in the callee. Both designs have nearly the same performance characteristics.

Since you are operating on `X[16]` in a rolling fashion instead of `W[64]` or `W[80]` the main body of your compression function will look similar to below.

```
// SHA-256 partial compression function
uint32x4_p X[16];
...

for (i = 16; i < 64; i++)
{
   uint32x4_p s0, s1, T0, T1;

   s0 = sigma0(X[(i + 1) & 0x0f]);
   s1 = sigma1(X[(i + 14) & 0x0f]);

   T1 = (X[i & 0xf] += s0 + s1 + X[(i + 9) & 0xf]);
```

```
    T1 += h + Sigma1(e) + Ch(e, f, g) + KEY[i];
    T2 = Sigma0(a) + Maj(a, b, c);
    ...
}
```

# Ch function

The SHA `Ch` function is implemented in POWER systems using the `vsel` instruction or the `vec_sel` builtin. The implementation for the 32x4 arrangement is shown below. The code is the same for the 64x2 arrangement, but the function takes `uint64x2_p` arguments. The important piece of information is `x` used as the selector.

```
uint32x4_p
VecCh(uint32x4_p x, uint32x4_p y, uint32x4_p z)
{
    return vec_sel(z, y, x);
}
```

# Maj function

The SHA `Maj` function is implemented in POWER systems using the `vsel` instruction or the `vec_sel` builtin. The implementation for the 32x4 arrangement is shown below. The code is the same for the 64x2 arrangement, but the function takes `uint64x2_p` arguments. The important piece of information is `x^y` used as the selector.

```
uint32x4_p
VecCh(uint32x4_p x, uint32x4_p y, uint32x4_p z)
{
    return vec_sel(y, z, vec_xor(x, y));
}
```

# Sigma functions

POWER8 provides the `vshasigmaw` and `vshasigmad` instructions to accelerate SHA calculations for 32-bit and 64-bit words, respectively. The instructions take two integer arguments and the constants are used to select among `Sigma0`, `Sigma1`, `sigma0` and `sigma1`.

The builtin GCC functions for the instructions are `__builtin_crypto_vshasigmaw` and `__builtin_crypto_vshasigmad`. The XLC functions for the instructions are `__vshasigmaw` and `__vshasigmad`. The C/C++ wrapper for the SHA-256 functions should look similar to below.

```
uint32x4_p Vec_sigma0(const uint32x4_p val)
{
#if defined(__xlc__) || defined(__xlC__)
    return __vshasigmaw(val, 0, 0);
```

```
#else
    return __builtin_crypto_vshasigmaw(val, 0, 0);
#endif
}

uint32x4_p Vec_sigma1(const uint32x4_p val)
{
#if defined(__xlc__) || defined(__xlC__)
    return __vshasigmaw(val, 0, 0xf);
#else
    return __builtin_crypto_vshasigmaw(val, 0, 0xf);
#endif
}

uint32x4_p VecSigma0(const uint32x4_p val)
{
#if defined(__xlc__) || defined(__xlC__)
    return __vshasigmaw(val, 1, 0);
#else
    return __builtin_crypto_vshasigmaw(val, 1, 0);
#endif
}

uint32x4_p VecSigma1(const uint32x4_p val)
{
#if defined(__xlc__) || defined(__xlC__)
    return __vshasigmaw(val, 1, 0xf);
#else
    return __builtin_crypto_vshasigmaw(val, 1, 0xf);
#endif
}
```

# SHA-256

The SHA-256 implementation has four parts. The first part is loads the existing state and creates working variables {A,B,C,D,E,F,G,H}. The second part loads the message and performs the first 16 rounds. The third part performs the remaining rounds. The final part stores the new state.

**Part 1.** Load the existing state and create working variables {A,B,C,D,E,F,G,H}.

```
uint32x4_p abcd = VecLoad32x4u(state+0, 0);
uint32x4_p efgh = VecLoad32x4u(state+4, 0);

enum {A=0,B=1,C,D,E,F,G,H};
uint32x4_p X[16], S[8];

S[A] = abcd; S[E] = efgh;
S[B] = VecShiftLeft<4>(S[A]);
```

```
S[F] = VecShiftLeft<4>(S[E]);
S[C] = VecShiftLeft<4>(S[B]);
S[G] = VecShiftLeft<4>(S[F]);
S[D] = VecShiftLeft<4>(S[C]);
S[H] = VecShiftLeft<4>(S[G]);
```

**Part 2.** Load the message and perform the first 16 rounds.

```
const uint32_t* k = reinterpret_cast<const uint32_t*>(KEY256);
const uint32_t* m = reinterpret_cast<const uint32_t*>(data);

uint32x4_p vm, vk;
unsigned int i, offset=0;

vk = VecLoad32x4(k, offset);
vm = VecLoadMsg32x4(m, offset);
SHA256_ROUND1<0>(X,S, vk,vm);
SHA256_ROUND1<1>(X,S, VecShiftLeft<4>(vk), VecShiftLeft<4>(vm));
SHA256_ROUND1<2>(X,S, VecShiftLeft<8>(vk), VecShiftLeft<8>(vm));
SHA256_ROUND1<3>(X,S, VecShiftLeft<12>(vk), VecShiftLeft<12>(vm));
offset+=16;

vk = VecLoad32x4(k, offset);
vm = VecLoadMsg32x4(m, offset);
SHA256_ROUND1<4>(X,S, vk,vm);
SHA256_ROUND1<5>(X,S, VecShiftLeft<4>(vk), VecShiftLeft<4>(vm));
SHA256_ROUND1<6>(X,S, VecShiftLeft<8>(vk), VecShiftLeft<8>(vm));
SHA256_ROUND1<7>(X,S, VecShiftLeft<12>(vk), VecShiftLeft<12>(vm));
offset+=16;

vk = VecLoad32x4(k, offset);
vm = VecLoadMsg32x4(m, offset);
SHA256_ROUND1<8>(X,S, vk,vm);
SHA256_ROUND1<9>(X,S, VecShiftLeft<4>(vk), VecShiftLeft<4>(vm));
SHA256_ROUND1<10>(X,S, VecShiftLeft<8>(vk), VecShiftLeft<8>(vm));
SHA256_ROUND1<11>(X,S, VecShiftLeft<12>(vk), VecShiftLeft<12>(vm));
offset+=16;

vk = VecLoad32x4(k, offset);
vm = VecLoadMsg32x4(m, offset);
SHA256_ROUND1<12>(X,S, vk,vm);
SHA256_ROUND1<13>(X,S, VecShiftLeft<4>(vk), VecShiftLeft<4>(vm));
SHA256_ROUND1<14>(X,S, VecShiftLeft<8>(vk), VecShiftLeft<8>(vm));
SHA256_ROUND1<15>(X,S, VecShiftLeft<12>(vk), VecShiftLeft<12>(vm));
offset+=16;
```

**Part 3.** Perform the remaining rounds.

```
for (i=16; i<64; i+=16)
{
```

```
    vk = VecLoad32x4(k, offset);
    SHA256_ROUND2<0>(X,S, vk);
    SHA256_ROUND2<1>(X,S, VecShiftLeft<4>(vk));
    SHA256_ROUND2<2>(X,S, VecShiftLeft<8>(vk));
    SHA256_ROUND2<3>(X,S, VecShiftLeft<12>(vk));
    offset+=16;

    vk = VecLoad32x4(k, offset);
    SHA256_ROUND2<4>(X,S, vk);
    SHA256_ROUND2<5>(X,S, VecShiftLeft<4>(vk));
    SHA256_ROUND2<6>(X,S, VecShiftLeft<8>(vk));
    SHA256_ROUND2<7>(X,S, VecShiftLeft<12>(vk));
    offset+=16;

    vk = VecLoad32x4(k, offset);
    SHA256_ROUND2<8>(X,S, vk);
    SHA256_ROUND2<9>(X,S, VecShiftLeft<4>(vk));
    SHA256_ROUND2<10>(X,S, VecShiftLeft<8>(vk));
    SHA256_ROUND2<11>(X,S, VecShiftLeft<12>(vk));
    offset+=16;

    vk = VecLoad32x4(k, offset);
    SHA256_ROUND2<12>(X,S, vk);
    SHA256_ROUND2<13>(X,S, VecShiftLeft<4>(vk));
    SHA256_ROUND2<14>(X,S, VecShiftLeft<8>(vk));
    SHA256_ROUND2<15>(X,S, VecShiftLeft<12>(vk));
    offset+=16;
}
```

**Part 4.** Repack and store the new state.

```
abcd += VecPack(S[A],S[B],S[C],S[D]);
efgh += VecPack(S[E],S[F],S[G],S[H]);

VecStore32x4u(abcd, state+0, 0);
VecStore32x4u(efgh, state+4, 0);
```

**VecLoadMsg32x4.** Perform an endian-aware load of a user message into a word.

```
template <class T>
uint32x4_p VecLoadMsg32x4(const T* data, int offset)
{
#if __LITTLE_ENDIAN__
    uint8x16_p mask = {3,2,1,0, 7,6,5,4, 11,10,9,8, 15,14,13,12};
    uint32x4_p r = VecLoad32x4u(data, offset);
    return (uint32x4_p)vec_perm(r, r, mask);
#else
    return VecLoad32x4u(data, offset);
#endif
}
```

**SHA256_ROUND1.** Mix state with a round key and user message.

```
template <unsigned int R>
void SHA256_ROUND1(uint32x4_p X[16], uint32x4_p S[8],
                   const uint32x4_p K, const uint32x4_p M)
{
    uint32x4_p T1, T2;

    X[R] = M;
    T1 = S[H] + VecSigma1(S[E]);
    T1 += VecCh(S[E],S[F],S[G]) + K + M;
    T2 = VecSigma0(S[A]) + VecMaj(S[A],S[B],S[C]);

    S[H] = S[G]; S[G] = S[F]; S[F] = S[E];
    S[E] = S[D] + T1;
    S[D] = S[C]; S[C] = S[B]; S[B] = S[A];
    S[A] = T1 + T2;
}
```

**SHA256_ROUND2.** Mix state with a round key.

```
template <unsigned int R>
void SHA256_ROUND2(uint32x4_p X[16], uint32x4_p S[8],
                   const uint32x4_p K)
{
    // Indexes into the X[] array
    enum {IDX0=(R+0)&0xf, IDX1=(R+1)&0xf,
          IDX9=(R+9)&0xf, IDX14=(R+14)&0xf};

    const uint32x4_p s0 = Vec_sigma0(X[IDX1]);
    const uint32x4_p s1 = Vec_sigma1(X[IDX14]);

    uint32x4_p T1 = (X[IDX0] += s0 + s1 + X[IDX9]);
    T1 += S[H] + VecSigma1(S[E]) + VecCh(S[E],S[F],S[G]) + K;
    uint32x4_p T2 = VecSigma0(S[A]) + VecMaj(S[A],S[B],S[C]);

    S[H] = S[G]; S[G] = S[F]; S[F] = S[E];
    S[E] = S[D] + T1;
    S[D] = S[C]; S[C] = S[B]; S[B] = S[A];
    S[A] = T1 + T2;
}
```

**VecPack.** Repack working variables.

```
uint32x4_p VecPack(const uint32x4_p a, const uint32x4_p b,
                   const uint32x4_p c, const uint32x4_p d)
{
    uint8x16_p m1 = {0,1,2,3, 16,17,18,19, 0,0,0,0, 0,0,0,0};
    uint8x16_p m2 = {0,1,2,3, 4,5,6,7, 16,17,18,19, 20,21,22,23};
    return vec_perm(vec_perm(a,b,m1), vec_perm(c,d,m1), m2);
```

```
}
```

# SHA-512

The SHA-512 implementation is like SHA-256 and has four parts. The first part is loads the existing state and creates working variables {A,B,C,D,E,F,G,H}. The second part loads the message and performs the first 16 rounds. The third part performs the remaining rounds. The final part stores the new state.

**Part 1.** Load the existing state and create working variables {A,B,C,D,E,F,G,H}.

```
uint64x2_p ab = VecLoad64x2u(state+0, 0);
uint64x2_p cd = VecLoad64x2u(state+2, 0);
uint64x2_p ef = VecLoad64x2u(state+4, 0);
uint64x2_p gh = VecLoad64x2u(state+6, 0);

// Indexes into the S[] array
enum {A=0, B=1, C, D, E, F, G, H};
uint64x2_p X[16], S[8];

S[A] = ab; S[C] = cd;
S[E] = ef; S[G] = gh;
S[B] = VecShiftLeft<8>(S[A]);
S[D] = VecShiftLeft<8>(S[C]);
S[F] = VecShiftLeft<8>(S[E]);
S[H] = VecShiftLeft<8>(S[G]);
```

**Part 2.** Load the message and perform the first 16 rounds.

```
const uint64_t* k = reinterpret_cast<const uint64_t*>(KEY512);
const uint64_t* m = reinterpret_cast<const uint64_t*>(data);

vk = VecLoad64x2(k, offset);
vm = VecLoadMsg64x2(m, offset);
SHA512_ROUND1<0>(X,S, vk,vm);
SHA512_ROUND1<1>(X,S, VecShiftLeft<8>(vk),VecShiftLeft<8>(vm));
offset+=16;

vk = VecLoad64x2(k, offset);
vm = VecLoadMsg64x2(m, offset);
SHA512_ROUND1<2>(X,S, vk,vm);
SHA512_ROUND1<3>(X,S, VecShiftLeft<8>(vk),VecShiftLeft<8>(vm));
offset+=16;

vk = VecLoad64x2(k, offset);
vm = VecLoadMsg64x2(m, offset);
SHA512_ROUND1<4>(X,S, vk,vm);
SHA512_ROUND1<5>(X,S, VecShiftLeft<8>(vk),VecShiftLeft<8>(vm));
offset+=16;
```

```
vk = VecLoad64x2(k, offset);
vm = VecLoadMsg64x2(m, offset);
SHA512_ROUND1<6>(X,S, vk,vm);
SHA512_ROUND1<7>(X,S, VecShiftLeft<8>(vk),VecShiftLeft<8>(vm));
offset+=16;

vk = VecLoad64x2(k, offset);
vm = VecLoadMsg64x2(m, offset);
SHA512_ROUND1<8>(X,S, vk,vm);
SHA512_ROUND1<9>(X,S, VecShiftLeft<8>(vk),VecShiftLeft<8>(vm));
offset+=16;

vk = VecLoad64x2(k, offset);
vm = VecLoadMsg64x2(m, offset);
SHA512_ROUND1<10>(X,S, vk,vm);
SHA512_ROUND1<11>(X,S, VecShiftLeft<8>(vk),VecShiftLeft<8>(vm));
offset+=16;

vk = VecLoad64x2(k, offset);
vm = VecLoadMsg64x2(m, offset);
SHA512_ROUND1<12>(X,S, vk,vm);
SHA512_ROUND1<13>(X,S, VecShiftLeft<8>(vk),VecShiftLeft<8>(vm));
offset+=16;

vk = VecLoad64x2(k, offset);
vm = VecLoadMsg64x2(m, offset);
SHA512_ROUND1<14>(X,S, vk,vm);
SHA512_ROUND1<15>(X,S, VecShiftLeft<8>(vk),VecShiftLeft<8>(vm));
offset+=16;
```

**Part 3.** Perform the remaining rounds.

```
for (i=16; i<80; i+=16)
{
    vk = VecLoad64x2(k, offset);
    SHA512_ROUND2<0>(X,S, vk);
    SHA512_ROUND2<1>(X,S, VecShiftLeft<8>(vk));
    offset+=16;

    vk = VecLoad64x2(k, offset);
    SHA512_ROUND2<2>(X,S, vk);
    SHA512_ROUND2<3>(X,S, VecShiftLeft<8>(vk));
    offset+=16;

    vk = VecLoad64x2(k, offset);
    SHA512_ROUND2<4>(X,S, vk);
    SHA512_ROUND2<5>(X,S, VecShiftLeft<8>(vk));
    offset+=16;
```

```
    vk = VecLoad64x2(k, offset);
    SHA512_ROUND2<6>(X,S, vk);
    SHA512_ROUND2<7>(X,S, VecShiftLeft<8>(vk));
    offset+=16;

    vk = VecLoad64x2(k, offset);
    SHA512_ROUND2<8>(X,S, vk);
    SHA512_ROUND2<9>(X,S, VecShiftLeft<8>(vk));
    offset+=16;

    vk = VecLoad64x2(k, offset);
    SHA512_ROUND2<10>(X,S, vk);
    SHA512_ROUND2<11>(X,S, VecShiftLeft<8>(vk));
    offset+=16;

    vk = VecLoad64x2(k, offset);
    SHA512_ROUND2<12>(X,S, vk);
    SHA512_ROUND2<13>(X,S, VecShiftLeft<8>(vk));
    offset+=16;

    vk = VecLoad64x2(k, offset);
    SHA512_ROUND2<14>(X,S, vk);
    SHA512_ROUND2<15>(X,S, VecShiftLeft<8>(vk));
    offset+=16;
}
```

**Part 4.** Repack and store the new state.

```
ab += VecPack(S[A],S[B]);
cd += VecPack(S[C],S[D]);
ef += VecPack(S[E],S[F]);
gh += VecPack(S[G],S[H]);

VecStore64x2u(ab, state+0, 0);
VecStore64x2u(cd, state+2, 0);
VecStore64x2u(ef, state+4, 0);
VecStore64x2u(gh, state+6, 0);
```

**VecLoadMsg64x2.** Perform an endian-aware load of a user message into a word.

```
template <class T>
uint32x4_p VecLoadMsg64x2(const T* data, int offset)
{
#if __LITTLE_ENDIAN__
    uint8x16_p mask = {7,6,5,4, 3,2,1,0, 15,14,13,12, 11,10,9,8};
    uint64x2_p r = VecLoad64x2u(data, offset);
    return (uint64x2_p)vec_perm(r, r, mask);
#else
    return VecLoad64x2u(data, offset);
```

```
#endif
}
```

**SHA512_ROUND1.** Mix state with a round key and user message.

```
template <unsigned int R>
void SHA512_ROUND1(uint64x2_p X[16], uint64x2_p S[8],
                   const uint64x2_p K, const uint64x2_p M)
{
    uint64x2_p T1, T2;

    X[R] = M;
    T1 = S[H] + VecSigma1(S[E]);
    T1 += VecCh(S[E],S[F],S[G]) + K + M;
    T2 = VecSigma0(S[A]) + VecMaj(S[A],S[B],S[C]);

    S[H] = S[G]; S[G] = S[F]; S[F] = S[E];
    S[E] = S[D] + T1;
    S[D] = S[C]; S[C] = S[B]; S[B] = S[A];
    S[A] = T1 + T2;
}
```

**SHA512_ROUND2.** Mix state with a round key.

```
template <unsigned int R>
void SHA512_ROUND2(uint64x2_p X[16], uint64x2_p S[8],
                   const uint64x2_p K)
{
    // Indexes into the X[] array
    enum {IDX0=(R+0)&0xf, IDX1=(R+1)&0xf,
          IDX9=(R+9)&0xf, IDX14=(R+14)&0xf};

    const uint64x2_p s0 = Vec_sigma0(X[IDX1]);
    const uint64x2_p s1 = Vec_sigma1(X[IDX14]);

    uint64x2_p T1 = (X[IDX0] += s0 + s1 + X[IDX9]);
    T1 += S[H] + VecSigma1(S[E]) + VecCh(S[E],S[F],S[G]) + K;
    uint64x2_p T2 = VecSigma0(S[A]) + VecMaj(S[A],S[B],S[C]);

    S[H] = S[G]; S[G] = S[F]; S[F] = S[E];
    S[E] = S[D] + T1;
    S[D] = S[C]; S[C] = S[B]; S[B] = S[A];
    S[A] = T1 + T2;
}
```

**VecPack.** Repack working variables.

```
uint64x2_p VecPack(const uint64x2_p x, const uint64x2_p y)
{
    const uint8x16_p m = {0,1,2,3, 4,5,6,7, 16,17,18,19, 20,21,22,23};
```

```
    return vec_perm(x,y,m);
}
```

The SHA-512 implementation uses the same functions as SHA-256, but SHA-512 uses a 64x2 arrangement rather than the 32x4 arrangement. You should copy/paste/replace as required for SHA-512. For example, below is the SHA Ch for the 64x2 arrangement.

```
uint64x2_p
VecCh(uint64x2_p x, uint64x2_p y, uint64x2_p z)
{
    return vec_sel(z,y,x);
}
```

In fact, since this is C++ code, a template function works nicely. The language will use the template to instantiate VecCh using both uint32x4_p and uint64x2_p.

```
template <class T>
T VecCh(T x, T y, T z)
{
    return vec_sel(z,y,x);
}
```

Templates do not work the Sigma functions and you will have to supply C++ overloaded functions as shown below.

```
uint64x2_p Vec_sigma0(const uint64x2_p val)
{
#if defined(__xlc__) || defined(__xlC__)
    return __vshasigmad(val, 0, 0);
#else
    return __builtin_crypto_vshasigmad(val, 0, 0);
#endif
}

uint64x2_p Vec_sigma1(const uint64x2_p val)
{
#if defined(__xlc__) || defined(__xlC__)
    return __vshasigmad(val, 0, 0xf);
#else
    return __builtin_crypto_vshasigmad(val, 0, 0xf);
#endif
}

uint64x2_p VecSigma0(const uint64x2_p val)
{
#if defined(__xlc__) || defined(__xlC__)
    return __vshasigmad(val, 1, 0);
#else
    return __builtin_crypto_vshasigmad(val, 1, 0);
#endif
```

```
}

uint64x2_p VecSigma1(const uint64x2_p val)
{
#if defined(__xlc__) || defined(__xlC__)
    return __vshasigmad(val, 1, 0xf);
#else
    return __builtin_crypto_vshasigmad(val, 1, 0xf);
#endif
}
```

# Chapter 6. Polynomial multiplication

This chapter discusses polynomial multiplication used with CRC codes and the GCM mode of operation for AES.

## CRC-32 and CRC-32C

CRC checksums on POWER8 are nothing like the SSE4 or Aarch64 CRC instructions. Please refer to Anton Blanchard's GitHub at CRC32/vpmsum [https://github.com/antonblanchard/crc32-vpmsum] for the discussion and sample code.

## GCM mode

POWER8 GCM mode is implemented using the `vpmsumd` instruction. GCM uses the double-word variant and to perform 64x64 → 128-bit polynomial multiplies. The GCC builtin is `__builtin_crypto_vpmsumd`, and the XLC intrinsic is `__vpmsumd`.

`vpmsumd` creates two 128-bit products and xor's them together. One product is from the multiplication of the low dwords, and the second product is from the multiplication of the high dwords. The trick to using `vpmsumd` is to ensure one of the products is 0. You can ensure one of the products is 0 by setting one of the 64-bit dwords to 0.

The output of `vpmsumd` is endian sensitive like the `vec_sld` instruction. On big-endian systems the result of a multiply is `{a,b}`, where `a` and `b` are double words in a vector. The same multiplication on a little-endian system produces `{b,a}`. The swapping can be seen with the test program below.

```
$ cat test.cxx
#include <altivec.h>
#undef vector
typedef __vector unsigned long long uint64x2_p;

int main(int argc, char* argv[])
{
  uint64x2_p a = {0, 04};
  uint64x2_p b = {0, 64};
  uint64x2_p c = __builtin_crypto_vpmsumd(a, b);

  return 0;
}
```

Running the program on `gcc112`, which is `ppc64-le`, results in the following output. Notice the output is `{0x100, 0x0}`.

```
(gdb) r
Starting program: /home/test/test.exe
Breakpoint 1, main (argc=0x1, argv=0x3fffffff4b8) at test.cxx:11
11        return 0;
```

```
(gdb) p c
$1 = {0x100, 0x0}
```

And running the program on `gcc119`, which is `ppc64-be`, results in the following output. Notice the output is `{0x0, 0x100}`.

```
(gdb) r
Starting program: /home/test/test.exe
Breakpoint 1, main (argc=0x1, argv=0x2ff22bb8) at test.cxx:11
11        return 0;
(gdb) p c
$1 = {0x0, 0x100}
```

Three helper functions are needed for GCM mode. The first two are `VecGetHigh` and `VecGetLow`. The functions extract the high and low 64-bit double words and return them in a vector. The third is `SwapWords`. `SwapWords` exchanges 64-bit double words on little-endian systems after the multiplication.

`VecGetHigh` and `VecGetLow` return a vector padded on the left with 0's. That means the return vector already has one of the terms set to 0 so `vpmsumd` will behave like ARM's `pmull_p64` or Intel's `_mm_clmulepi64_si128`.

The source code for the three functions are shown below. Shifts and rotates are preferred over permutes or masks because shifts are generally faster and use fewer instructions.

```
uint64x2_p VecGetHigh(uint64x2_p val)
{
    return VecShiftRight<8>(val);
}

uint64x2_p VecGetLow(uint64x2_p val)
{
    return VecShiftRight<8>(VecShiftLeft<8>(val));
}

template <class T>
SwapWords(const T val)
{
    return (T)VecRotateLeft<8>(val);
}
```

Using `VecGetHigh` and `VecGetLow` the function `VecPolyMultiply` can be implemented as follows using the `vpmsumd` builtin.

```
uint64x2_p
VecPolyMultiply(uint64x2_p a, uint64x2_p b)
{
#if defined(__xlc__) || defined(__xlC__)
#  if __BIG_ENDIAN__
     return __vpmsumd(VecGetLow(a), VecGetLow(b));
```

```
#  else
     return SwapWords(__vpmsumd(VecGetLow(a), VecGetLow(b)));
#  endif
#else
#  if __BIG_ENDIAN__
     return __builtin_crypto_vpmsumd(
                VecGetLow(a), VecGetLow(b));
#  else
     return SwapWords(__builtin_crypto_vpmsumd(
                VecGetLow(a), VecGetLow(b)));
#  endif
#endif
}
```

The code listed above provides Intel's `_mm_clmulepi64_si128(a, b, 0x00)` or ARM's `pmull_p64`. The function may be better named `VecPolyMult00` because it multiplies the two low 64-bit double words. The table below shows how to create the four variations needed for GCM mode.

| Function | Parameter a | Parameter b |
|---|---|---|
| `VecPolyMult00` | `VecGetLow(a)` | `VecGetLow(b)` |
| `VecPolyMult01` | `VecGetLow(a)` | `VecGetHigh(b)` |
| `VecPolyMult10` | `VecGetHigh(a)` | `VecGetLow(b)` |
| `VecPolyMult11` | `VecGetHigh(a)` | `VecGetHigh(b)` |

The next steps are implement the GCM multiplication and reduction routines. The code for multiplication is easy and shown below. The code for reduction is the tricky one and left as an exercise to the reader.

```
uint64x2_p
GCM_Multiply(uint64x2_p x, uint64x2_p h)
{
    uint64x2_p c0 = VecPolyMult00(x, h);
    uint64x2_p c1 = VecXor(VecPolyMult01(x, h), VecPolyMult10(x, h));
    uint64x2_p c2 = VecPolyMult11(x, h);

    return GCM_Reduce(c0, c1, c2);
}
```

Implementing `GCM_Multiply` and `GCM_Reduce` will require some forethought. You will likely use a reflected algorithm, and nearly everything gets turned on its head. For example, GCM's vpmsum wrapper swaps words on big-endian systems and leaves words alone on little-endian systems. As another example GCM's vpmsum wrapper reads low words with `VecGetHigh` and high words with `VecGetLow` to ensure the correct words are presented to the function call.

Also see the head notes for POWER8 in the Crypto++ implementation at `gcm-simd.cpp` [https://github.com/weidai11/cryptopp/blob/master/gcm-simd.cpp]. If you are feeling adventurous then you can find the Cryptogams implementation for OpenSSL

at `ghashp8-ppc.pl` [https://github.com/openssl/openssl/blob/master/crypto/modes/asm/ghashp8-ppc.pl].

# Chapter 7. Assembly language

This chapter shows you how to build and link against a POWER8 SHA assembly language routine. The function is Cryptogams SHA-256 compression function.

Cryptogams [https://www.openssl.org/~appro/cryptogams/] is the incubator used by Andy Polyakov to develop assembly language routines for OpenSSL. Andy dual licenses his implementations and a more permissive license is available for his assembly language source code.

## Cryptogams

The steps that follow were carried out on `gcc112`, which is ppc64-le. Andy's GitHub is located at dot-asm [https://github.com/dot-asm], so clone the project and read the README.

```
$ git clone https://github.com/dot-asm/cryptogams
$ cd cryptogams
```

The README contains instructions for using the source files:

> "Flavor" refers to ABI family or specific OS. E.g. x86_64 scripts recognize "elf", "elf32", "macosx", "mingw64", "nasm". PPC scripts recognize "linux32", "linux64", "linux64le", "aix32", "aix64", "osx32", "osx64", and so on...

Unfortunately Andy has not uploaded the SHA gear to Cryptogams so you will have to switch to OpenSSL to get the Cryptogams sources. Make a `cryptogams` directory, and then copy `sha512p8-ppc.pl` and `ppc-xlate.pl` from the OpenSSL source directory:

```
$ mkdir cryptogams
$ cp openssl/crypto/sha/asm/sha512p8-ppc.pl cryptogams/
$ cp openssl/crypto/perlasm/ppc-xlate.pl cryptogams/
$ cd cryptogams/
```

Next examine the head notes in `sha512p8-ppc.pl`, which is used to create the source files for SHA-256 and SHA-512. The comments say the script takes two arguments. The first is a “flavor”, and the 32 or 64 is used to convey the platform architecture. Adding “le” to flavor will produce a source file for a little endian machine. The second argument is “output”, and 256 or 512 in the output filename selects either SHA-256 or SHA-512.

The commands to produce a SHA-256 assembly source file for `gcc112` and assemble it are shown below.

```
$ ./sha512p8-ppc.pl linux64le sha256le_compress.s
$ as -mpower8 sha256le_compress.s -o sha256le_compress.o
```

The head notes in `sha512p8-ppc.pl` do not state the public API. However the source file `crypto/ppccap.c` says:

```
$ grep -IR sha256_block_p8 *
```

```
crypto/ppccap.c:void sha256_block_p8(void *ctx, const void *inp,
size_t len);
...
```

In fact the signature for `sha256_block_p8` is better documented as shown below. There are no alignment requirements for `state` or `input`.

```
void sha256_block_p8(uint32_t *state,
            const uint8_t *input, size_t blocks);
```

Finally, a program that links to `sha256_block_p8` might look like the following.

```
$ cat test.cxx
#include <stdio.h>
#include <string.h>
#include <stdint.h>

extern "C" {
  void sha256_block_p8(uint32_t*, const uint8_t*, size_t);
}

int main(int argc, char* argv[])
{
  /* empty message with padding */
  uint8_t message[64];
  memset(message, 0x00, sizeof(message));
  message[0] = 0x80;

  /* initial state */
  uint32_t state[8] = {
    0x6a09e667, 0xbb67ae85, 0x3c6ef372, 0xa54ff53a,
    0x510e527f, 0x9b05688c, 0x1f83d9ab, 0x5be0cd19
  };

  size_t blocks = sizeof(message)/64;
  sha256_block_p8(state, message, blocks);

  const uint8_t b1 = (uint8_t)(state[0] >> 24);
  const uint8_t b2 = (uint8_t)(state[0] >> 16);
  const uint8_t b3 = (uint8_t)(state[0] >>  8);
  const uint8_t b4 = (uint8_t)(state[0] >>  0);
  const uint8_t b5 = (uint8_t)(state[1] >> 24);
  const uint8_t b6 = (uint8_t)(state[1] >> 16);
  const uint8_t b7 = (uint8_t)(state[1] >>  8);
  const uint8_t b8 = (uint8_t)(state[1] >>  0);

  /* e3b0c44298fc1c14... */
  printf("SHA256 hash of empty message: ");
  printf("%02X%02X%02X%02X%02X%02X%02X%02X...\n",
         b1, b2, b3, b4, b5, b6, b7, b8);
```

```
  int success = ((b1 == 0xE3) && (b2 == 0xB0) &&
                 (b3 == 0xC4) && (b4 == 0x42) &&
                 (b5 == 0x98) && (b6 == 0xFC) &&
                 (b7 == 0x1C) && (b8 == 0x14));

  if (success)
    printf("Success!\n");
  else
    printf("Failure!\n");

  return (success != 0 ? 0 : 1);
}
```

Compiling and linking to `sha256le_compress.o` would look similar to below.

```
$ g++ -mcpu=power8 test.cxx sha256le_compress.o -o test.exe
$ ./test.exe
SHA256 hash of empty message: E3B0C44298FC1C14...
Success!
```

# Chapter 8. Performance

This chapter presents benchmarking numbers and discusses some of the issues that affect performance. Benchmarking an application is an art and can be tricky to collect accurate results.

## Power states

Linux desktop systems are usually configured in either `on-demand` or `powersave` mode. The configuration is usually a kernel parameter, and the default energy states are usually efficient states that use less power. Before benchmarking you should leave `on-demand` or `powersave` mode, and enter a `performance` state.

Cryptogams uses a script to enter `performance` mode for benchmarking but it is not available online. A modified version of Andy's script is available at `governor.sh` [https://github.com/weidai11/cryptopp/blob/master/TestScripts/governor.sh]. The script changes the scaling frequency using the `/sys/devices/system/cpu/cpu*/cpufreq/scaling_governor` key (where `cpu*` is a logical cpu, like `cpu0`). Below is an example of running the script on a x86_64 Linux system.

```
$ sudo ./governor.sh perf
Current CPU governor scaling settings:
  CPU 0: powersave
  CPU 1: powersave
  CPU 2: powersave
  CPU 3: powersave
New CPU governor scaling settings:
  CPU 0: performance
  CPU 1: performance
  CPU 2: performance
  CPU 3: performance
```

TODO: We are not aware of a similar script for AIX. In fact we don't know how to check a similar setting to determine if a script is needed.

## Benchmarks

The table below presents benchmark statistics using standard C++, C++ with builtins, and assembly language routines. The "standard C++" and "C++ with builtins" columns were derived using Crypto++. The "assembly language" column was taken from Cryptogams and OpenSSL source file notes.

The measurements were taken on `gcc112`, which is a Linux PowerPC, 64-bit, little-endian machine. The hardware is IBM POWER System S822 with two CPU cards, 160 logical CPUs at 3.4 GHz. The kernel is CentOS 7.4 version `3.10.0-514` and the compiler is GCC 7.2.0.

| Algorithm | Standard C++ | | Built-ins | | Assembly | |
|---|---|---|---|---|---|---|
| | MiB/s | cpb | MiB/s | cpb | MiB/s | cpb |
| AES/ECB | 121 | 26.7 | 3151 | 1.03 | - | -† |
| AES/CTR | 120 | 27.1 | 2544 | 1.27 | - | 0.74‡ |
| AES/GCM | 93 | 34.7 | 2269 | 1.35 | - | -† |
| SHA-1 | 307 | 10.6 | N/A | N/A | - | -† |
| SHA-256 | 129 | 25.2 | 275 | 12.0 | - | 9.9‡ |
| SHA-512 | 281 | 11.5 | 368 | 8.8 | - | 6.3‡ |

† The Cryptogams and OpenSSL source files do not provide a meaningful metric for cycles per byte in this case. The metrics are reported as +150% or +90%, but we don't know the frame of reference.

‡ As reported in the head notes of the Cryptogams and OpenSSL source files. MiB/s is not provided, and the metric is only meaningful on the same hardware using the same test configuration.

POWER8 *does not* provide instructions for SHA-1, and based on the benchmarks it is easy to see why. A good compiler generates quality code. Other factors probably include SHA-1 has been superseded by SHA-256 and SHA-512.

# Chapter 9. References

## Cryptogams

- CRYPTOGAMS: low-level cryptographic primitives collection [https://www.openssl.org/~appro/cryptogams/]

## GitHub

- AES Intrinsics [https://github.com/noloader/AES-Intrinsics]
- SHA Intrinsics [https://github.com/noloader/SHA-Intrinsics]
- CRC32/vpmsum [https://github.com/antonblanchard/crc32-vpmsum]

## IBM and OpenPOWER websites

- Recommended debug, compiler, and linker settings for Power processor tuning [https://www.ibm.com/support/knowledgecenter/en/linuxonibm/liaal/iplsdkrecbldset.htm]
- AIX vector programming [https://www.ibm.com/support/knowledgecenter/en/ssw_aix_61/com.ibm.aix.genprogc/vector_prog.htm]
- POWER8 in-core cryptography [https://www.ibm.com/developerworks/library/se-power8-in-core-cryptography/index.html]
- IBM Advance Toolchain (for latest gcc and glibc) [https://developer.ibm.com/linuxonpower/advance-toolchain/]
- 64-Bit ELF V2 ABI Specification: Power Architecture [https://openpowerfoundation.org/?resource_lib=64-bit-elf-v2-abi-specification-power-architecture]
- IBM Power ISA Version 3.0B [https://openpowerfoundation.org/?resource_lib=power-isa-version-3-0]
- Function calls and the PowerPC 64-bit ABI [https://www.ibm.com/developerworks/library/l-powasm4/index.html]
- Performance Optimization and Tuning Techniques for IBM Power Systems Processors Including IBM POWER8 [https://www.redbooks.ibm.com/redbooks/pdfs/sg248171.pdf]

## NIST website

- FIPS 197, Advanced Encryption Standard (AES) [https://nvlpubs.nist.gov/nistpubs/fips/nist.fips.197.pdf]
- FIPS 180-4, Secure Hash Standard (SHS) [https://nvlpubs.nist.gov/nistpubs/fips/nist.fips.180-4.pdf]


# Stack Exchange


- Detect POWER8 in-core crypto through getauxval? [https://stackoverflow.com/q/46144668/608639]
- Is vec_sld endian sensitive? [https://stackoverflow.com/q/46341923/608639]

# Index

## Symbols



## A



## B



## C



## F



## G



## I



## L



## M



## N



## O