POWER8 in-core Cryptography

The Unofficial Guide

Jeffrey Walton
Dr. William Schmidt

POWER8 in-core Cryptography: The Unofficial Guide

by Jeffrey Walton and Dr. William Schmidt
Extensive review and rough drafts: Segher Boessenkool

Publication date 1 April 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.

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.

Organization

The book proceeds in six 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, 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.

Third, 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.

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

Fifth, 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.

Finally, 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.

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 PowerPC64. Access is provided through SSH.

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

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.

PowerPC compilers

Two compilers are used for testing. The first is GCC and the second is IBM XL C/C++. The compilers are mostly the same but accept slightly different options.

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).

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

Complimentary, 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.

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

When compiling source code to examine the quality of code generation the program should be compiled with `-O3`. Both compilers consume `-O3`.

Altivec headers

The header required for datatypes and functions is `<altivec.h>`. To support compiles with a C++ compiler `__vector` keyword is used rather than `vector`. A typical Altivec include looks as shown below.

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

In addition to `__ALTIVEC__` preprocessor macro you will see the following defines depending on the platform:

- `__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
- `__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 to activate the macro. Source code checking endianness should look similar to the code shown below.

```
#if __LITTLE_ENDIAN__
# error "Little-endian system"
#else
# error "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
```

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 in-core 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 typedef'd as `uint8x16_p8`. `__vector unsigned int` is arranged as 4 each 32-bit words, and it is typedef'd as `uint32x4_p8`.

POWER8 added `__vector unsigned long` and associated vector operations. `__vector unsigned long` is arranged as 2 each 64-bit double words, and it is typedef'd as `uint64x2_p8`.

The typedef naming was selected to convey the arrangement, like `32x4` and `64x2`. The trailing `_p8` was selected to avoid collisions with ARM NEON vector data types. The suffix _p (for POWER architecture) or _v (for Vector) would work just as well.

Vector shifts

Altivec shifts and rotates are performed using *Vector Shift Left Double by Octet Immediate*. The vector shift and rotate built-in 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 need a special handling to produce correct results and the 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]. If you are like many other developers then you will literally waste hours trying to figure it out what happened the first time you experience it.

The issue is shifts and rotates are endian sensitive [http://stackoverflow.com/q/46341923/608639], and you have to use `16-n` 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 VectorShiftLeft(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 `VectorRotateLeft` 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 VectorRotateLeft(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 POWER permute 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 permute 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 code only needed one vector so it used `msg` twice in the call to `vec_perm`. An example that interleaves two different vectors is shown below.

```
uint32x4_p a = { 0,  0,  0,  0};  // All 0 bits
uint32x4_p a = {-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};
uint8x16_p c = vec_perm(a, b, c);
```

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

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 here: [TODO: Provide the URL once the technical problems with posting the errata have been overcome.] Subsequent sections describe the proper way to use loads and stores of aligned and unaligned data.

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 memory address and the offset into the address. The effective address is the sum `address+offset` rounded down or masked to a multiple of 16.

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_p8 VectorLoad(const T* mem_addr, int offset)
{
#ifndef NDEBUG
    uintptr_t maddr = ((uintptr_t)mem_addr)+offset;
    uintptr_t mask = ~(uintptr_t)0xf;
    uintptr_t eaddr = maddr & mask;
    assert(maddr == eaddr);
#endif

    return (uint32x4_p8)vec_ld(offset, (uint8_t*)mem_addr);
}
```

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

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

    vec_st(offset, (uint8_t*)mem_addr);
}
```

Casting away `const`-ness on `mem_addr` is discussed in `const` pointers below.

Unaligned data references

POWER7 (PowerISA 2.07) introduced unaligned loads and stores that avoid the aligned memory requirements. The preferred intrinsic functions for unaligned loads and stores are `vec_xl` and `vec_xst`. Theses 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 loads and stores whenever you *do not control* buffers or their alignments, like messages supplied by user code.

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_p8 VectorLoadu(const T* mem_addr, int offset)
{
#if defined(__xlc__) || defined(__xlC__)
```

```
    return (uint32x4_p8)vec_xl(offset, (uint8_t*)mem_addr);
#else
    return (uint32x4_p8)vec_vsx_ld(offset, (uint8_t*)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 VectorStoreu(T* mem_addr, int offset)
{
#if defined(__xlc__) || defined(__xlC__)
    vec_xst((uint8x16_p8)val, offset, (uint8_t*)mem_addr);
#else
    vec_vsx_st((uint8x16_p8)val, offset, (uint8_t*)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.

Casting away `const`-ness on `mem_addr` is discussed in `const` pointers below.

Big-endian data references

POWER7 introduced `vec_xl_be` and `vec_st_be` which perform big-endian loads and stores. The big-endian load compiles/assembles to `lxvw4x/lxvd2x`, and the store compiles/assembles to `stxvw4x/stxvd2x`.

The big-endian variants can save two instructions on little-endian systems when the little-endian byte swap is not needed. This usually happens when you need to permute the data after a load or before a store.

The extraneous permutes can be seen in the disassembly below. The interleaved instructions were removed. The instructions which remain are (1) a load of the value, (2) a load of the mask, and (3) three permutations instead of one.

```
$ objdump --disassemble sha256-p8.exe

SHA256_SCHEDULE(unsigned int*, unsigned char const*):
...
100008a8:   99 4e 00 7c     lxvd2x  vs32,0,r9
100008bc:   99 26 20 7c     lxvd2x  vs33,0,r4
100008cc:   57 02 00 f0     xxswapd vs32,vs32
100008d0:   57 0a 21 f0     xxswapd vs33,vs33
100008d4:   97 05 00 f0     xxlnand vs32,vs32,vs32
100008d8:   2b 08 21 10     vperm   v1,v1,v1,v0
...
```

While not readily apparent, `v0` is another name for `vs32`, and `v1` is another name for `vs33`. So the permutation may be thought of as `vperm vs33,vs33,vs33,vs32` (although this will not assemble). Also see What does “vperm v0,v0,v0,v17” with unused v0 do? [https://stackoverflow.com/q/49132339/608639].

Access to `vec_xl_be` and `vec_st_be` was provided for IBM XL C/C++, but GCC only supports these intrinsic functions beginning with version 8 (Spring 2018 release). For earlier versions of GCC, you must use inline assembly to replace the missing built-ins with `VEC_XL_BE` and `VEC_ST_BE`. [*WJS: Is it intended to provide definitions of these macros here?*]

Const pointers

The Altivec built-ins have unusual behavior when using `const` pointers during a load operation. A program runs slower when the memory is marked as `const`. The behavior has been witnessed in two libraries on different machines and the decrease in performance is measurable. For example, AES runs 0.3 cycles per byte (cpb) faster [http://github.com/randombit/botan/pull/1459] when non-`const` pointers are used for loads. 0.3 cpb may not sound like much but it equates to 200 MiB/s for AES-128 on `gcc112`.

Source code using the `non-const` pointers should look similar to below:

```
template <class T>
uint32x4_p8 VectorLoad(const T* mem_addr, int offset)
{
    // mem_addr must be aligned to 16-byte boundary
    return (uint32x4_p8)vec_ld(offset, (uint8_t*)mem_addr);
}
```

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 in-core crypto availability at runtime on AIX and Linux PowerPC platforms.

AIX features

TODO: find out how to perform runtime feature detection on AIX. We checked `getsystemcfg` and `sysconf` for ISA 2.07, polynomial multiply, AES and SHA (and crypto) bits but they are missing.

The only thing we have found is `SIGILL` probes and signal handlers. It would be nice to avoid the nastiness.

[*WJS: I'm not an AIX expert either, but I've send a note to someone who should be able to help us.*]

Linux features

Some versions of Glibc and the kernel provide ELF auxiliary vectors with the information. `AT_HWCAP2` will show the `vcrypto` flag when in-core crypto 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

```
$ 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. Runtime code to perform the check should look similar to below. The defines 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_HWCAP2
# define AT_HWCAP2 26
#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 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;
}
```

SIGILL probes

TODO: show this nasty technique.

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 a program can query the L1 data cache line size 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 a program can query the L1 data cache line size as shown below.

```
#include <sys/sysconf.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. `gcc112` runs CentOS 7.4 and the machine returns 0 for the L1 cache line query. 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.

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.

Three topics are discussed for AES. The first is encryption, the second is decryption, and the third is keying. Keying is discussed last because encryption and decryption uses the golden key schedule from FIPS 197.

AES encryption

TODO

AES decryption

TODO

AES key schedule

TODO

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.

Sigma functions

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

Ch function

POWER8 provides the `vsel` instruction and it is SHA's `Ch` function. The implementation for the 32x4 arrangement is shown below. The code is the same for the 64x2 arrangement, but the function takes `uint64x2_p8` arguments. The important piece of information is `x` used as the selector.

```
uint32x4_p8
VectorCh(uint32x4_p8 x, uint32x4_p8 y, uint32x4_p8 z)
{
    return vec_sel(z, y, x);
}
```

Maj function

POWER8 provides the `vsel` instruction and it can be used for SHA's `Maj` function. The implementation for the 32x4 arrangement is shown below. The code is the same for the 64x2 arrangement, but the function takes `uint64x2_p8` arguments. The important piece of information is `x^y` used as the selector.

```
uint32x4_p8
VectorCh(uint32x4_p8 x, uint32x4_p8 y, uint32x4_p8 z)
{
    return vec_sel(y, z, vec_xor(x, y));
}
```

SHA-256

TODO

SHA-512

TODO

Chapter 6. Polynomial multiplication

The chapter of the document should discuss polynomial multiplication used with CRC codes and the GCM mode of operation for AES. However we have no experience with polynomial multiplication. Please refer to GitHub CRC32/vpmsum [https://github.com/antonblanchard/crc32-vpmsum].

CRC-32 and CRC-32C

No content.

GCM mode

No content.

Chapter 7. Assembly language

This chapter demonstrates building and linking against projects that provide SHA assembly language routines. The steps show you the integration of the routines without the baggage of a full blown library.

Cryptogams

Cryptogams [https://www.openssl.org/~appro/cryptogams/] is Andy Polyakov's incubator to develop assembly language routines for OpenSSL. Andy dual licenses his implementations, so a more permissive license is available for the assembly language source code. This section will show you how to build Andy's software.

sha2-le

The PPC64 team on GitHub [https://github.com/PPC64/sha2-le/] provide a SHA-256 assembly implementation built using `m4` macros. This section explains how to build the assembly source file, create a test program and link against the assembly-based object file.

The team only provides little-endian so you will need to modify the source files for big-endian. There is a pull request [https://github.com/PPC64/sha2-le/pull/6] that will be useful if you want both little-endian and big-endian support.

The steps that follow were carried out on `gcc112`, which is ppc64-le. To begin, clone the project, build the sources and test the program.

```
$ git clone https://github.com/PPC64/sha2-le.git
$ cd sha2-le
...

$ make COMPILERS=gcc
...

$ make test COMPILERS=gcc
...

========================================================
Testing gcc
========================================================
./bin/test256_gcc
./bin/test512_gcc
CC=gcc ./blackbox-test.sh
Running tests for SHA-256:
Test #1:        sha2-le is Ok   libcrypto is Ok c is Ok
Test #2:        sha2-le is Ok   libcrypto is Ok c is Ok
Test #3:        sha2-le is Ok   libcrypto is Ok c is Ok
```

```
Test #4:        sha2-le is Ok   libcrypto is Ok c is Ok
...
```

Next, create the assembly language source file from m4 sources, and then create the object file by assembling the source file.

```
$ make clean
...

$ m4 common.m4 sha256_compress_ppc.m4 > sha256_compress.s
$ as -mpower8 sha256_compress.s -o sha256_compress.o
```

You can examine the disassembly with the following command. The output below shows round calculations.

```
$ objdump --disassemble sha256_compress.o
sha256_compress.o:     file format elf64-powerpcle

Disassembly of section .text:

0000000000000000 <sha256_compress_ppc>:

 ...
 144:   6a 73 0f 10     vsel    v0,v15,v14,v13
 148:   c4 54 29 10     vxor    v1,v9,v10
 14c:   82 fe 6d 10     vshasigmaw v3,v13,1,15
 150:   80 d0 c0 10     vadduwm v6,v0,v26
 154:   80 18 b0 10     vadduwm v5,v16,v3
 158:   6a 58 2a 10     vsel    v1,v10,v11,v1
 15c:   80 30 e5 10     vadduwm v7,v5,v6
 160:   82 86 49 10     vshasigmaw v2,v9,1,0
 164:   80 08 02 11     vadduwm v8,v2,v1
 168:   80 38 8c 11     vadduwm v12,v12,v7
 16c:   80 40 07 12     vadduwm v16,v7,v8
 ...
```

The comments in `sha256_compress.s` state the public API for the function is as follows. The documentation does not state the alignment requirements of `state`, `input` or `keys`. When in doubt you should align the memory to a 16-byte boundary.

```
void sha256_compress_ppc(
    uint32_t *state,
    const uint8_t *input,
    const uint32_t *keys)
```

Finally, a program that links to sha2-le's `sha256_compress_ppc` might look like the following.

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

```
extern "C" {
  void sha256_compress_ppc(uint32_t*,
            const uint8_t*, const uint32_t*);
}

#define ALIGN16 __attribute__((aligned(16)))

const ALIGN16 uint32_t K256[] =
  {
    0x428A2F98, 0x71374491, 0xB5C0FBCF, 0xE9B5DBA5,
    0x3956C25B, 0x59F111F1, 0x923F82A4, 0xAB1C5ED5,
    0xD807AA98, 0x12835B01, 0x243185BE, 0x550C7DC3,
    0x72BE5D74, 0x80DEB1FE, 0x9BDC06A7, 0xC19BF174,
    0xE49B69C1, 0xEFBE4786, 0x0FC19DC6, 0x240CA1CC,
    0x2DE92C6F, 0x4A7484AA, 0x5CB0A9DC, 0x76F988DA,
    0x983E5152, 0xA831C66D, 0xB00327C8, 0xBF597FC7,
    0xC6E00BF3, 0xD5A79147, 0x06CA6351, 0x14292967,
    0x27B70A85, 0x2E1B2138, 0x4D2C6DFC, 0x53380D13,
    0x650A7354, 0x766A0ABB, 0x81C2C92E, 0x92722C85,
    0xA2BFE8A1, 0xA81A664B, 0xC24B8B70, 0xC76C51A3,
    0xD192E819, 0xD6990624, 0xF40E3585, 0x106AA070,
    0x19A4C116, 0x1E376C08, 0x2748774C, 0x34B0BCB5,
    0x391C0CB3, 0x4ED8AA4A, 0x5B9CCA4F, 0x682E6FF3,
    0x748F82EE, 0x78A5636F, 0x84C87814, 0x8CC70208,
    0x90BEFFFA, 0xA4506CEB, 0xBEF9A3F7, 0xC67178F2
  };

int main(int argc, char* argv[])
{
  /* empty message with padding */
  ALIGN16 uint8_t message[64];
  memset(message, 0x00, sizeof(message));
  message[0] = 0x80;

  /* initial state */
  ALIGN16 uint32_t state[8] = {
    0x6a09e667, 0xbb67ae85, 0x3c6ef372, 0xa54ff53a,
    0x510e527f, 0x9b05688c, 0x1f83d9ab, 0x5be0cd19
  };

  sha256_compress_ppc(state, message, K256);

  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 `sha256_compress_ppc.o` would look similar to below.

```
$ g++ test.cxx -o test.exe sha256_compress.o
$ ./test.exe
SHA256 hash of empty message: E3B0C44298FC1C14...
Success!
```

Chapter 8. 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]
- sha2-le [https://github.com/PPC64/sha2-le]

IBM website

- 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]

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