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 acclerate 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 its ancestry in Altivec SIMD coprocessor. The POWER8 vector unit includes Vector Multimedia Extension (VSX) and the instruction set for in-core cryptography are part it. You can find additional information on VSX at IBM's website at TODO [https://www.ibm.com/developerworks/].

Organization

The book proceeds in six parts. First, administriva 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 two POWER8 iron (the other two are POWER7 hardware). `gcc112` is a Linux ppc64-le machine (PowerPC, 64-bit, little-endian), and `gcc119` is an AIX ppc64-be machine (PowerPC, 64-bit, big-endian).

Both POWER8 machines are IBM POWER System S822 with two CPU cards. `gcc112` has 160 logical CPUs, and `gcc119` has 64 logical CPUs. 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, inlcuding MIPS64, Aarch64 and PowerPC64. Access is provided through SSH.

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

Chapter 2. Vector programming

Polynomial multiplies, AES, SHA-256 and SHA-512 acceleration occurs in the VSX unit. Data must be moved from main memory into a vector register, the data must be transformed, and then data must be written back to main memory. Data is moved to and from main memory using vector loads and stores.

Several topics need to be discussed to ensure trouble free loads and stores. They include PowerPC compilers and options, Altivec headers, machine endianness, vector datatypes and then finally the loads and stores.

PowerPC compilers

This documents uses two compilers for testing. The first is GCC and the second is IBM XL C/C++. Each compiler is slightly different with its 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

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
```

Malloc and new

The system calls `malloc` and `new` (and friends) are used to acquire memory from the heap. The system calls *do not* guarantee aligment 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.

To avoid unexpected surprises when using heap allocations you should use `posix_memalign` to acquire heap memory aligned to a particular boundary.

TODO: I believe AIX or XLC has another function to call for memory used with vector programming.

Vector datatypes

Three vector datatypes are used for in-core programming. The three types are listed below.

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

`__vector unsigned char` is 16 each 8-bit bytes, and it is typedef'd as `uint8x16_p8`. `__vector unsigned int` is 4 each 32-bit words, and it is typedef'd as `uint32x4_p8`.

`__vector unsigned long` is new to POWER8. `__vector unsigned long` is 2 each 64-bit double words, and it is typedef'd as `uint64x2_p8`.

Loads and stores

Altivec loads and stores have traditionally been performed using `vec_ld` and `vec_st` since at least the POWER4 days in the 1990s. `vec_ld` and `vec_st` are sensitive to alignment of the effective memory address. The effective address is the `address+offset` rounded down or masked to a multiple of 16.

The effective address used for `vec_ld` and `vec_st` must be aligned to a 16-byte boundary or incorrect results will arise. Altivec *does not* raise a `SIGBUS`. Instead, the bottom 3 bits of the address are masked-off and then the memory at the effective address is loaded.

POWER7 introduced unaligned loads and stores that avoid the aligned memory address requirement. The instructions to use for unaligned loads and stores are `vec_vsx_ld` and `vec_vsx_st` when using GCC; and `vec_xl` and `vec_xst` when using XLC.

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.

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. TODO: which versions?

```
$ 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 HasPower8Crypto()
{
    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

XXX

AES decryption

XXX

AES key scheduling

XXX

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 a `sigma` instruction to accelrate SHA calculations. The instruction takes 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

XXX

SHA-512

XXX

Chapter 6. Polynomial multiplication

The chapter of the document should discuss polynomial multiplcation 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

XXX.

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 chapter will show you how to build Andy's software.

Chapter 8. References

The following is a list of references we are aware of.

Cryptogams

The folowing is a list of Cryptogams references.

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

GitHub

The folowing is a list of GitHub references.

- 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

The folowing is a list of IBM references.

- Recommended debug, compiler, and linker settings for Power processor tuning [https://www.ibm.com/support/knowledgecenter/en/linuxonibm/liaal/iplsdkrecbldset.htm]
- POWER8 in-core cryptography [https://www.ibm.com/developerworks/library/se-power8-in-core-cryptography/index.html]

NIST

The folowing is a list of NIST references.

- 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

The folowing is a list of Stack Exchange references.

- Detect Power8 in-core crypto through getauxval? [https://stackoverflow.com/q/46144668/608639]

Chapter 9. Revision History

Revision History		
Revision 1	1 April 2018	jww, ws
Initial release		

Index

C

S