Linux on Power Porting Guide

Vector Intrinsics

Workgroup Notes

Revision 1.0 (April 11, 2018)

OpenPOWER™

www.openpowerfoundation.org

Linux on Power Porting Guide: Vector Intrinsics

System Software Work Group<`syssw-chair@openpowerfoundation.org`>
OpenPOWER Foundation

Revision 1.0 (April 11, 2018)

Abstract

The goal of this project is to provide functional equivalents of the Intel MMX, SSE, and AVX intrinsic functions, that are commonly used in Linux applications, and make them (or equivalents) available for the PowerPC64LE platform.

This document is a Non-standard Track, Work Group Note work product owned by the System Software Workgroup and handled in compliance with the requirements outlined in the *OpenPOWER Foundation Work Group (WG) Process* document. It was created using the *Master Template Guide* version 0.9.5. Comments, questions, etc. can be submitted to the public mailing list for this document at <`syssw-programming-guides@mailinglist.openpowerfoundation.org`>.

Table of Contents

Preface iv
1. Conventions iv
2. Document change history v
1. Intel Intrinsic porting guide for Power64LE 1
1.1. Look at the source, Luke 1
1.1.1. The structure of the intrinsic includes 2
1.1.2. The types used for intrinsics 3
1.1.2.1. Dealing with MMX 5
1.1.2.2. Dealing with AVX and AVX512 5
1.1.3. How the API is implemented 7
1.1.3.1. Some simple examples 7
1.1.3.2. Those extra attributes 8
1.1.3.3. How did I find this out? 10
1.1.3.4. Examples implemented using other intrinsics 10
2. How do we work this? 12
2.1. Preferred methods 13
2.2. Prepare yourself 13
2.2.1. GCC Vector Extensions 14
2.2.2. Intel Intrinsic functions 15
2.2.2.1. Packed vs scalar intrinsics 17
2.2.2.2. To vec_not or not 20
2.2.2.3. Crossing lanes 22
2.2.3. PowerISA Vector facilities 24
2.2.3.1. The PowerISA 24
2.2.3.2. PowerISA Vector Intrinsics 29
2.2.3.3. How vector elements change size and type 30
2.2.4. Some more intrinsic examples 31
2.3. Profound differences 32
2.3.1. Floating Point Exceptions 32
2.3.2. Floating-point rounding modes 33
2.3.3. Performance 33
2.3.3.1. Using SSE float and double scalars 34
2.3.3.2. Using MMX intrinsics 34
A. Document references 35
A.1. OpenPOWER and Power documents 35
A.2. A.2 Intel documents 35
A.3. A.3 GNU Compiler Collection (GCC) documents 35
B. Intel Intrinsic suffixes 36
B.1. MMX 36
B.2. SSE 36
B.3. SSE2 36
B.4. AVX/AVX2 __m256_* 37
B.5. AVX512 __m512_* 37
C. OpenPOWER Foundation overview 39
C.1. Foundation documentation 39
C.2. Technical resources 39
C.3. Contact the foundation 40

Preface

1. Conventions

The OpenPOWER Foundation documentation uses several typesetting conventions.

Notices

Notices take these forms:

Note

A handy tip or reminder.

Important

Something you must be aware of before proceeding.

Warning

Critical information about the risk of data loss or security issues.

Changes

At certain points in the document lifecycle, knowing what changed in a document is important. In these situations, the following conventions will used.

- *New text will appear like this.* Text marked in this way is completely new.
- ~~Deleted text will appear like this.~~ Text marked in this way was removed from the previous version and will not appear in the final, published document.
- Changed text will appear like this. Text marked in this way appeared in previous versions but has been modified.

Command prompts

In general, examples use commands from the Linux operating system. Many of these are also common with Mac OS, but may differ greatly from the Windows operating system equivalents.

For the Linux-based commands referenced, the following conventions will be followed:

$ prompt	Any user, including the `root` user, can run commands that are prefixed with the $ prompt.
# prompt	The `root` user must run commands that are prefixed with the # prompt. You can also prefix these commands with the **sudo** command, if available, to run them.

Document links

Document links frequently appear throughout the documents. Generally, these links include a text for the link, followed by a page number in parenthesis. For example, this link, Preface [iv], references the Preface chapter on page iv.

2. Document change history

This version of the guide replaces and obsoletes all earlier versions.

The following table describes the most recent changes:

Revision Date	Summary of Changes
March 14, 2018	• Revision 1.0 - Minor updates. Published version.
October 30, 2017	• Revision 0.3 - Updates to describe issues associated with larger vector sizes and proposed solutions.
September 14, 2017	• Revision 0.2 - Miscellaneous correction for spelling, grammar and punctuation.
July 26, 2017	• Revision 0.1 - initial draft from Steve Munroe

1. Intel Intrinsic porting guide for Power64LE

The goal of this project is to provide functional equivalents of the Intel MMX, SSE, and AVX intrinsic functions, that are commonly used in Linux applications, and make them (or equivalents) available for the PowerPC64LE platform. These X86 intrinsics started with the Intel and Microsoft compilers but were then ported to the GCC compiler. The GCC implementation is a set of headers with inline functions. These inline functions provide an implementation mapping from the Intel/Microsoft dialect intrinsic names to the corresponding GCC Intel built-ins or directly via C language vector extension syntax.

The current proposal is to start with the existing X86 GCC intrinsic headers and port them (copy and change the source) to POWER using C language vector extensions, VMX and VSX built-ins. Another key assumption is that we will be able to use many of the existing Intel DejaGNU test cases in ./gcc/testsuite/gcc.target/i386. This document is intended as a guide to developers participating in this effort. However this document provides guidance and examples that should be useful to developers who may encounter X86 intrinsics in code that they are porting to another platform.

Note

(*We have started contributions of X86 intrinsic headers to the GCC project.*) The current status of the project is the BMI (bmiintrin.h), BMI2 (bmi2intrin.h), MMX (mmintrin.h), and SSE (xmmintrin.h) intrinsic headers are committed to GCC development trunk for GCC 8. Work on SSE2 (emmintrin.h) is in progress.

1.1. Look at the source, Luke

So if this is a code porting activity, where is the source? All the source code we need to look at is in the GCC source trees. You can either git (https://gcc.gnu.org/wiki/GitMirro) the gcc source or download one of the recent AT source tars (for example: ftp://ftp.unicamp.br/pub/linuxpatch/toolchain/at/ubuntu/dists/xenial/at10.0/). You will find the intrinsic headers in the ./gcc/config/i386/ sub-directory.

If you have an Intel Linux workstation or laptop with GCC installed, you already have these headers, if you want to take a look:

```
$ find /usr/lib -name '*mmintrin.h'
/usr/lib/gcc/x86_64-redhat-linux/4.4.4/include/wmmintrin.h
/usr/lib/gcc/x86_64-redhat-linux/4.4.4/include/mmintrin.h
/usr/lib/gcc/x86_64-redhat-linux/4.4.4/include/xmmintrin.h
/usr/lib/gcc/x86_64-redhat-linux/4.4.4/include/emmintrin.h
/usr/lib/gcc/x86_64-redhat-linux/4.4.4/include/tmmintrin.h
...
$
```

But depending on the vintage of the distro, these may not be the latest versions of the headers. Looking at the header source will tell you a few things: the include structure (what other headers are implicitly included), the types that are used at the API, and finally, how the API is implemented.

```
smmintrin.h (SSE4.1) includes tmmintrin,h
tmmintrin.h (SSSE3) includes pmmintrin.h
```

```
pmmintrin.h (SSE3)  includes emmintrin,h
emmintrin.h (SSE2)  includes xmmintrin.h
xmmintrin.h (SSE)   includes mmintrin.h and mm_malloc.h
mmintrin.h  (MMX)
```

1.1.1. The structure of the intrinsic includes

The GCC x86 intrinsic functions for vector were initially grouped by technology (MMX and SSE), which starts with MMX and continues with SSE through SSE4.1 stacked like a set of Russian dolls.

Basically each higher layer include needs typedefs and helper macros defined by the lower level intrinsic includes. mm_malloc.h simply provides wrappers for posix_memalign and free. Then it gets a little weird, starting with the crypto extensions:

```
wmmintrin.h  (AES) includes emmintrin.h
```

For AVX, AVX2, and AVX512 they must have decided that the Russian Dolls thing was getting out of hand. AVX et al. is split across 14 files:

```
#include <avxintrin.h>
#include <avx2intrin.h>
#include <avx512fintrin.h>
#include <avx512erintrin.h>
#include <avx512pfintrin.h>
#include <avx512cdintrin.h>
#include <avx512vlintrin.h>
#include <avx512bwintrin.h>
#include <avx512dqintrin.h>
#include <avx512vlbwintrin.h>
#include <avx512vldqintrin.h>
#include <avx512ifmaintrin.h>
#include <avx512ifmavlintrin.h>
#include <avx512vbmiintrin.h>
#include <avx512vbmivlintrin.h>
```

but they do not want the applications to include these individually.

So **immintrin.h** includes everything Intel vector, including all the AVX, AES, SSE, and MMX flavors.

```
#ifndef _IMMINTRIN_H_INCLUDED
# error "Never use <avxintrin.h> directly; include <immintrin.h> instead."
#endif
```

So why is this interesting? The include structure provides some strong clues about the order that we should approach this effort. For example if you need to use intrinsics from SSE4 (smmintrin.h) you are likely to need to type definitions from SSE (emmintrin.h). So a bottoms up (MMX, SSE, SSE2, …) approach seems like the best plan of attack. Also saving the AVX parts for later make sense, as most are just wider forms of operations that already exist in SSE.

We should use the same include structure to implement our PowerISA equivalent API headers. This will make porting easier (drop-in replacement) and should get the application running quickly on POWER. Then we will be in a position to profile and analyze the resulting application. This will show any hot spots where the simple one-to-one transformation results in bottlenecks and additional tuning is needed. For these cases we should improve our tools (SDK MA/SCA) to identify opportunities for, and perhaps propose, alternative sequences that are better tuned to PowerISA and our micro-architecture.

1.1.2. The types used for intrinsics

The type system for Intel intrinsics is a little strange. For example from xmmintrin.h:

```
/* The Intel API is flexible enough that we must allow aliasing with other
   vector types, and their scalar components.  */
typedef float __m128 __attribute__ ((__vector_size__ (16), __may_alias__));

/* Internal data types for implementing the intrinsics.  */
typedef float __v4sf __attribute__ ((__vector_size__ (16)));
```

So there is one set of types that are used in the function prototypes of the API, and the internal types that are used in the implementation. Notice the special attribute `__may_alias__`. From the GCC documentation:

> Accesses through pointers to types with this attribute are not subject to type-based alias analysis, but are instead assumed to be able to alias any other type of objects. ... This extension exists to support some vector APIs, in which pointers to one vector type are permitted to alias pointers to a different vector type.

There are a couple of issues here:

- The use of __may_alias__ in the API seems to force the compiler to assume aliasing of any parameter passed by reference.
- The GCC vector builtin type system (example above) is slightly different syntax from the original Altivec __vector types. Internally the two typedef forms may represent the same 128-bit vector type, but for early source parsing and overloaded vector builtins they are handled differently.
- The data type used at the interface may not be the correct type for the implied operation.

Normally the compiler assumes that parameters of different size do not overlap in storage, which allows more optimization. However parameters for different vector element sizes [char | short | int | long] are all passed and returned as type `__m128i` (defined as vector long long).

This may not matter when using x86 built-ins but does matter when the implementation uses C vector extensions or in our case using PowerPC overloaded vector built-ins (Section 2.2.3.2, “PowerISA Vector Intrinsics” [29]). For the latter cases the type must be correct for the compiler to generate the correct code for the type (char, short, int, long) (Section 1.1.3, “How the API is implemented” [7]) for overloaded builtin operations. There is also concern that excessive use of `__may_alias__` will limit compiler optimization. We are not sure how important this attribute is to the correct operation of the API. So at a later stage we should experiment with removing it from our implementation for PowerPC.

The good news is that PowerISA has good support for 128-bit vectors and (with the addition of VSX) all the required vector data (char, short, int, long, float, double) types. However Intel supports a wider variety of the vector sizes than PowerISA does. This started with the 64-bit MMX vector support that preceded SSE and extends to 256-bit and 512-bit vectors of AVX, AVX2, and AVX512 that followed SSE.

Within the GCC Intel intrinsic implementation these are all implemented as vector attribute extensions of the appropriate size (`__vector_size__` ({8 | 16 | 32, and 64}). For the PowerPC target GCC currently only supports the native `__vector_size__` (16). These we can support directly in VMX/VSX registers and associated instructions.

GCC will compile code with other `__vector_size__` values, but the resulting types are treated as simple arrays of the element type. This does not allow the compiler to use the vector registers for parameter passing and return values. For example this intrinsic from immintrin.h:

```
typedef double __m256d __attribute__ ((__vector_size__ (32), __may_alias__));

extern __inline __m256d __attribute__((__gnu_inline__, __always_inline__,
 __artificial__))
_mm256_add_pd (__m256d __A, __m256d __B)
{
  return (__m256d) ((__v4df)__A + (__v4df)__B);
}
```

And test case:

```
__m256d
test_mm256_add_pd (__m256d __A, __m256d __B)
{
  return (_mm256_add_pd (__A, __B));
}
```

Current GCC generates:

```
0000000000000970 <test_mm256_add_pd>:
 970: 10 00 20 39  li      r9,16
 974: 98 26 80 7d  lxvd2x  vs12,0,r4
 978: 98 2e 40 7d  lxvd2x  vs10,0,r5
 97c: 20 00 e0 38  li      r7,32
 980: f8 ff e1 fb  std     r31,-8(r1)
 984: b1 ff 21 f8  stdu    r1,-80(r1)
 988: 30 00 00 39  li      r8,48
 98c: 98 4e 04 7c  lxvd2x  vs0,r4,r9
 990: 98 4e 65 7d  lxvd2x  vs11,r5,r9
 994: 00 53 8c f1  xvadddp vs12,vs12,vs10
 998: 00 00 c1 e8  ld      r6,0(r1)
 99c: 78 0b 3f 7c  mr      r31,r1
 9a0: 00 5b 00 f0  xvadddp vs0,vs0,vs11
 9a4: c1 ff c1 f8  stdu    r6,-64(r1)
 9a8: 98 3f 9f 7d  stxvd2x vs12,r31,r7
 9ac: 98 47 1f 7c  stxvd2x vs0,r31,r8
 9b0: 98 3e 9f 7d  lxvd2x  vs12,r31,r7
 9b4: 98 46 1f 7c  lxvd2x  vs0,r31,r8
 9b8: 50 00 3f 38  addi    r1,r31,80
 9bc: f8 ff e1 eb  ld      r31,-8(r1)
 9c0: 98 1f 80 7d  stxvd2x vs12,0,r3
 9c4: 98 4f 03 7c  stxvd2x vs0,r3,r9
 9c8: 20 00 80 4e  blr
```

The compiler treats the parameters and return value as scalar arrays, which are passed by reference. The operation is vectorized in this case, but the 256-bit result is returned through storage.

This is not what we want to see for a simple 4 by double add. It would be better if we can pass and return MMX (Section 1.1.2.1, “Dealing with MMX” [5]) and AVX (Section 1.1.2.2, “Dealing with AVX and AVX512” [5]) values as PowerPC registers and avoid the storage references. If we can get the parameter and return values as registers, this example will reduce to:

```
0000000000000970 <test_mx256_add_pd>:
 970: xvadddp vs34,vs34,vs36
 974: xvadddp vs35,vs35,vs37
 978: blr
```

So the PowerISA VMX/VSX facilities and GCC compiler support for 128-bit/16-byte vectors and associated vector built-ins are well matched to implementing equivalent X86 SSE intrinsic functions. However implementing the older MMX (64-bit) and the latest AVX (256 / 512-bit) extensions requires more thought and some ingenuity.

1.1.2.1. Dealing with MMX

MMX is actually the harder case. The `__m64` type supports SIMD vector int types (char, short, int, long). The Intel API defines `__m64` as:

```
typedef int __m64 __attribute__ ((__vector_size__ (8), __may_alias__));
```

Which is problematic for the PowerPC target (not really supported in GCC) and we would prefer to use a native PowerISA type that can be passed in a single register. The PowerISA Rotate Under Mask instructions can easily extract and insert integer fields of a General Purpose Register (GPR). This implies that MMX integer types can be handled as an internal union of arrays for the supported element types. So a 64-bit unsigned long long is the best type for parameter passing and return values, especially for the 64-bit (_si64) operations as these normally generate a single PowerISA instruction. So for the PowerPC implementation we will define `__m64` as:

```
typedef __attribute__ ((__aligned__ (8))) unsigned long long __m64;
```

The SSE extensions include some copy / convert operations for `_m128` to / from `_m64` and this includes some int to / from float conversions. However in these cases the float operands always reside in SSE (XMM) registers (which match the PowerISA vector registers) and the MMX registers only contain integer values. POWER8 (PowerISA-2.07) has direct move instructions between GPRs and VSRs. So these transfers are normally a single instruction and any conversions can be handled in the vector unit.

When transferring a `__m64` value to a vector register we should also execute a xxsplatd instruction to insure there is valid data in all four float element lanes before doing floating point operations. This avoids causing extraneous floating point exceptions that might be generated by uninitialized parts of the vector. The top two lanes will have the floating point results that are in position for direct transfer to a GPR or stored via Store Float Double (stfd). These operation are internal to the intrinsic implementation and there is no requirement to keep temporary vectors in correct Little Endian form.

Also for the smaller element sizes and higher element counts (MMX `_pi8` and `_p16` types) the number of Rotate Under Mask instructions required to disassemble the 64-bit `__m64` into elements, perform the element calculations, and reassemble the elements in a single `__m64` value can get larger. In this case we can generate shorter instruction sequences by transfering (via direct move instruction) the GPR `__m64` value to the a vector register, performance the SIMD operation there, then transfer the `__m64` result back to a GPR.

1.1.2.2. Dealing with AVX and AVX512

AVX is a bit easier for PowerISA and the ELF V2 ABI. First we have lots (64) of vector registers and a superscalar vector pipeline (can execute two or more independent 128-bit vector operations concurrently). Second the ELF V2 ABI was designed to pass and return larger aggregates in vector registers:

- Up to 12 qualified vector arguments can be passed in v2–v13.
- A qualified vector argument corresponds to:
  - A vector data type

- A member of a homogeneous aggregate of multiple like data types passed in up to eight vector registers.
- Homogeneous floating-point or vector aggregate return values that consist of up to eight registers with up to eight elements will be returned in floating-point or vector registers that correspond to the parameter registers that would be used if the return value type were the first input parameter to a function.

So the ABI allows for passing up to three structures each representing 512-bit vectors and returning such (512-bit) structures all in VMX registers. This can be extended further by spilling parameters (beyond 12 X 128-bit vectors) to the parameter save area, but we should not need that, as most intrinsics only use 2 or 3 operands.. Vector registers not needed for parameter passing, along with an additional 8 volatile vector registers, are available for scratch and local variables. All can be used by the application without requiring register spill to the save area. So most intrinsic operations on 256- or 512-bit vectors can be held within existing PowerISA vector registers.

For larger functions that might use multiple AVX 256 or 512-bit intrinsics and, as a result, push beyond the 20 volatile vector registers, the compiler will just allocate non-volatile vector registers by allocating a stack frame and spilling non-volatile vector registers to the save area (as needed in the function prologue). This frees up to 64 vectors (32 x 256-bit or 16 x 512-bit structs) for code optimization.

Based on the specifics of our ISA and ABI we will not use `__vector_size__` (32) or (64) in the PowerPC implementation of `__m256` and `__m512` types. Instead we will typedef structs of 2 or 4 vector (`__vector`) fields. This allows efficient handling of these larger data types without requiring new GCC language extensions or vector builtins. For example:

```
/* Internal data types for implementing the AVX in PowerISA intrinsics.  */
typedef struct __v4df
{
  __vector double vd0;
  __vector double vd1;
} __vx4df;

/* The Intel API is flexible enough that we must allow aliasing with other
   vector types, and their scalar components.  */
typedef struct __m256d
{
  __vector double vd0;
  __vector double vd1;
}__attribute__ ((__may_alias__)) __m256d;
```

This requires a different syntax for operations where the 128-bit vector chunks are explicitly referenced. For example:

```
extern __inline __mx256d __attribute__((__gnu_inline__, __always_inline__,
 __artificial__))
_mm256_add_pd (__m256d __A, __m256d __B)
{
  __m256d temp;
  temp.vd0 = __A.vd0 + __B.vd0;
  temp.vd1 = __A.vd1 + __B.vd1;
  return (temp);
}
```

But this creates a new issue because the C language does not allow direct casts between structs. This can be an issue where the intrinsic interface type is not the correct type for the operation. For example AVX2 integer operations:

```
/* The Intel API is flexible enough that we must allow aliasing with other
   vector types, and their scalar components.  */
typedef struct __m256i
{
  __vector long long vdi0;
  __vector long long vdi1;
} __m256i;

/* Internal data types for implementing the AVX in PowerISA intrinsics.  */
typedef struct __v16hi
{
  __vector short vhi0;
  __vector short vhi1;
} __v16hi;
```

For the AVX2 intrinsic `_mm256_add_epi16` we need to cast the input vectors of 64-bit long long (`__m256i`) into vectors of 16-bit short (`__v16hi`) before the overloaded add operations. Here we need to use a pointer reference cast. For example:

```
extern __inline __m256i __attribute__((__gnu_inline__, __always_inline__,
 __artificial__))
_mx256_add_epi16 (__m256i __A, __m256i __B)
{
  __m256i result;
  __v16hi a = *((__v16hi *)&__A);
  __v16hi b = *((__v16hi *)&__B);
  __v16hi c;

  c.vhi0 = a.vhi0 + b.vhi0;
  c.vhi1 = a.vhi1 + b.vhi1;

  result = *((__m256i *)&c);
  return (result);
}
```

As this and related examples are inlined, we expect the compiler to recognize this is a "nop cast" and avoid generating any additional instructions.

In the end we should try to use the same type names and definitions as the GCC X86 intrinsic headers where possible. Where that is not possible we can define new typedefs that provide the best mapping to the underlying PowerISA hardware.

1.1.3. How the API is implemented

One pleasant surprise is that many (at least for the older Intel) Intrinsics are implemented directly in C vector extension code and/or a simple mapping to GCC target specific builtins.

1.1.3.1. Some simple examples

For example; a vector double splat looks like this:

```
/* Create a vector with both elements equal to F.  */
extern __inline __m128d __attribute__((__gnu_inline__, __always_inline__,
 __artificial__))
_mm_set1_pd (double __F)
{
  return __extension__ (__m128d){ __F, __F };
}
```

Another example:

```
extern __inline __m128d __attribute__((__gnu_inline__, __always_inline__,
 __artificial__))
_mm_add_pd (__m128d __A, __m128d __B)
{
  return (__m128d) ((__v2df)__A + (__v2df)__B);
}
```

Note in the example above the cast to __v2df for the operation. Both __m128d and __v2df are vector double, but __v2df does no have the `__may_alias__` attribute. And one more example:

```
extern __inline __m128i __attribute__((__gnu_inline__, __always_inline__,
 __artificial__))
_mm_mullo_epi16 (__m128i __A, __m128i __B)
{
  return (__m128i) ((__v8hu)__A * (__v8hu)__B);
}
```

Note this requires a cast for the compiler to generate the correct code for the intended operation. The parameters and result are the generic interface type `__m128i`, which is a vector long long with the `__may_alias__` attribute. But operation is a vector multiply low on unsigned short elements. So not only do we use the cast to drop the `__may_alias__` attribute but we also need to cast to the correct type (`__v8hu` or vector unsigned short) for the specified operation.

I have successfully copied these (and similar) source snippets over to the PPC64LE implementation unchanged. This of course assumes the associated types are defined and with compatible attributes.

1.1.3.2. Those extra attributes

You may have noticed there are some special attributes:

```
__gnu_inline__

This attribute should be used with a function that is also declared with the
inline keyword. It directs GCC to treat the function as if it were defined in
gnu90 mode even when compiling in C99 or gnu99 mode.

If the function is declared extern, then this definition of the function is used
only for inlining. In no case is the function compiled as a standalone function,
not even if you take its address explicitly. Such an address becomes an external
reference, as if you had only declared the function, and had not defined it. This
has almost the effect of a macro. The way to use this is to put a function
definition in a header file with this attribute, and put another copy of the
function, without extern, in a library file. The definition in the header file
causes most calls to the function to be inlined.
```

```
__always_inline__

Generally, functions are not inlined unless optimization is specified. For func-
tions declared inline, this attribute inlines the function independent of any
restrictions that otherwise apply to inlining. Failure to inline such a function
is diagnosed as an error.

__artificial__

This attribute is useful for small inline wrappers that if possible should appear
during debugging as a unit. Depending on the debug info format it either means
marking the function as artificial or using the caller location for all instructions
within the inlined body.

__extension__

... -pedantic' and other options cause warnings for many GNU C extensions.
You can prevent such warnings within one expression by writing __extension__
```

So far I have been using these attributes unchanged.

But most intrinsics map the Intel intrinsic to one or more target specific GCC builtins. For example:

```
/* Load two DPFP values from P.  The address must be 16-byte aligned.  */
extern __inline __m128d __attribute__((__gnu_inline__, __always_inline__,
 __artificial__))
_mm_load_pd (double const *__P)
{
  return *(__m128d *)__P;
}

/* Load two DPFP values from P.  The address need not be 16-byte aligned.  */
extern __inline __m128d __attribute__((__gnu_inline__, __always_inline__,
 __artificial__))
_mm_loadu_pd (double const *__P)
{
  return __builtin_ia32_loadupd (__P);
}
```

The first intrinsic (_mm_load_pd) is implement as a C vector pointer reference, but from the comment assumes the compiler will use a **movapd** instruction that requires 16-byte alignment (will raise a general-protection exception if not aligned). This implies that there is a performance advantage for at least some Intel processors to keep the vector aligned. The second intrinsic uses the explicit GCC builtin **`__builtin_ia32_loadupd`** to generate the **`movupd`** instruction which handles unaligned references.

The opposite assumption applies to POWER and PPC64LE, where GCC generates the VSX **`lxvd2x`** / **`xxswapd`** instruction sequence by default, which allows unaligned references. The PowerISA equivalent for aligned vector access is the VMX **`lvx`** instruction and the **`vec_ld`** builtin, which forces quadword aligned access (by ignoring the low order 4 bits of the effective address). The **`lvx`** instruction does not raise alignment exceptions, but perhaps should as part of our implementation of the Intel intrinsic. This requires that we use PowerISA VMX/VSX built-ins to insure we get the expected results.

The current prototype defines the following:

```
/* Load two DPFP values from P.  The address must be 16-byte aligned.  */
extern __inline __m128d __attribute__((__gnu_inline__, __always_inline__,
 __artificial__))
_mm_load_pd (double const *__P)
{
  assert(((unsigned long)__P & 0xfUL) == 0UL);
  return ((__m128d)vec_ld(0, (__v16qu*)__P));
}

/* Load two DPFP values from P.  The address need not be 16-byte aligned.  */
extern __inline __m128d __attribute__((__gnu_inline__, __always_inline__,
 __artificial__))
_mm_loadu_pd (double const *__P)
{
  return (vec_vsx_ld(0, __P));
}
```

The aligned load intrinsic adds an assert which checks alignment (to match the Intel semantic) and uses the GCC builtin **`vec_ld`** (generates an **`lvx`**). The assert generates extra code but this can be eliminated by defining **`NDEBUG`** at compile time. The unaligned load intrinsic uses the GCC builtin `vec_vsx_ld` (for PPC64LE generates **`lxvd2x`** / **`xxswapd`** for POWER8 and will simplify to **`lxv`** or **`lxvx`** for POWER9). And similarly for **`__mm_store_pd`** / **`__mm_storeu_pd`**, using **`vec_st`** and **`vec_vsx_st`**. These concepts extent to the load/store intrinsics for vector float and vector int.

1.1.3.3. How did I find this out?

The next question is where did I get the details above. The GCC documentation for **`__builtin_ia32_loadupd`** provides minimal information (the builtin name, parameters and return types). Not very informative.

Looking up the Intel intrinsic description is more informative. You can Google the intrinsic name or use the Intel Intrinsic guide for this. The Intrinsic Guide is interactive and includes Intel (Chip) technology and text based search capabilities. Clicking on the intrinsic name opens to a synopsis including; the underlying instruction name, text description, operation pseudo code, and in some cases performance information (latency and throughput).

The key is to get a description of the intrinsic (operand fields and types, and which fields are updated for the result) and the underlying Intel instruction. If the Intrinsic guide is not clear you can look up the instruction details in the "Intel® 64 and IA-32 Architectures Software Developer's Manual".

Information about the PowerISA vector facilities is found in the PowerISA Version 2.07B (for POWER8 and 3.0 for POWER9) manual, Book I, Chapter 6. Vector Facility and Chapter 7. Vector-Scalar Floating-Point Operations. Another good reference is the OpenPOWER ELF V2 application binary interface (ABI) document, Chapter 6. Vector Programming Interfaces and Appendix A. Predefined Functions for Vector Programming.

Another useful document is the original Altivec Technology Programmers Interface Manual with a user friendly structure and many helpful diagrams. But alas the PIM does does not cover the recent PowerISA (power7, power8, and power9) enhancements.

1.1.3.4. Examples implemented using other intrinsics

Some intrinsic implementations are defined in terms of other intrinsics. For example.

```
/* Create a vector with element [0] as F and the rest zero.  */
extern __inline __m128d __attribute__((__gnu_inline__, __always_inline__,
 __artificial__))
_mm_set_sd (double __F)
{
  return __extension__ (__m128d){ __F, 0.0 };
}

/* Create a vector with element [0] as *P and the rest zero.  */
extern __inline __m128d __attribute__((__gnu_inline__, __always_inline__,
 __artificial__))
_mm_load_sd (double const *__P)
{
  return _mm_set_sd (*__P);
}
```

This notion of using part (one fourth or half) of the SSE XMM register and leaving the rest unchanged (or forced to zero) is specific to SSE scalar operations and can generate some complicated (sub-optimal) PowerISA code. In this case **`_mm_load_sd`** passes the dereferenced double value to **`_mm_set_sd`** which uses C vector initializer notation to combine (merge) that double scalar value with a scalar 0.0 constant into a vector double.

While code like this should work as-is for PPC64LE, you should look at the generated code and assess if it is reasonable. In this case the code is not awful (a load double splat, vector xor to generate 0.0s, then a `xxmrghd` to combine __F and 0.0). Other examples may generate sub-optimal code and justify a rewrite to PowerISA scalar or vector code (*GCC PowerPC AltiVec Built-in Functions* or inline assembler).

Note

Try using the existing C code if you can, but check on what the compiler generates. If the generated code is horrendous, it may be worth the effort to write a PowerISA specific equivalent. For codes making extensive use of MMX or SSE scalar intrinsics you will be better off rewriting to use standard C scalar types and letting the GCC compiler handle the details (see Section 2.1, “Preferred methods” [13]).

2. How do we work this?

The working assumption is to start with the existing GCC headers from `./gcc/config/i386/`, then convert them to PowerISA and add them to `./gcc/config/rs6000/`. I assume we will replicate the existing header structure and retain the existing header file and intrinsic names. This also allows us to reuse existing DejaGNU test cases from `./gcc/testsuite/gcc.target/i386`, modify them as needed for the POWER target, and add them to `./gcc/testsuite/gcc.target/powerpc`.

We can be flexible on the sequence that headers/intrinsics and test cases are ported. This should be based on customer need and resolving internal dependencies. This implies an oldest-to-newest / bottoms-up (MMX, SSE, SSE2, …) strategy. The assumption is, existing community and user application codes, are more likely to have optimized code for previous generation ubiquitous (SSE, SSE2, ...) processors than the latest (and rare) SkyLake AVX512.

I would start with an existing header from the current GCC ./gcc/config/i386/ and copy the header comment (including FSF copyright) down to any vector typedefs used in the API or implementation. Skip the Intel intrinsic implementation code for now, but add the ending #end if matching the headers conditional guard against multiple inclusion. You can add additional #include's as needed. For example:

```
/* Copyright (C) 2003-2017 Free Software Foundation, Inc
...
/* This header provides a best effort implementation of the Intel X86
 * SSE2 intrinsics for the PowerPC target.  This implementation is a
 * combination of compiled C vector codes or equivalent sequences of
 * GCC vector builtins from the GCC PowerPC Altivec target.
 *
 * However some details of this implementation will differ from
 * the X86 due to differences in the underlying hardware or GCC
 * implementation. For example the PowerPC target only uses unordered
 * floating point compares. */

#ifndef EMMINTRIN_H_
#define EMMINTRIN_H_

#include <altivec.h>
#include <assert.h>

/* We need definitions from the SSE header files.  */
#include <xmmintrin.h>

/* The Intel API is flexible enough that we must allow aliasing with other
   vector types, and their scalar components.  */
typedef float __m128 __attribute__ ((__vector_size__ (16), __may_alias__));

/* Internal data types for implementing the intrinsics.  */
typedef __vector float __v4sf;
/* more typedefs.  */

/* The intrinsic implmentations go here.  */

#endif /* EMMINTRIN_H_ */
```

Note

The interface typedef (__m128) uses the GCC vector builtin extension syntax, while the internal typedef (__v4sf) uses the altivec vector extension syntax. This allows the internal typedefs to work correctly with the PowerPC overloaded vector builtins. Also we use the __vector (vs vector) type prefix to avoid name space conflicts with C++.

Then you can start adding small groups of related intrinsic implementations to the header to be compiled and examine the generated code. Once you have what looks like reasonable code you can grep through ./gcc/testsuite/gcc.target/i386 for examples using the intrinsic names you just added. You should be able to find functional tests for most X86 intrinsics.

The *GCC testsuite* uses the DejaGNU test framework as documented in the *GNU Compiler Collection (GCC) Internals* manual. GCC adds its own DejaGNU directives and extensions, that are embedded in the testsuite source as comments. Some are platform specific and will need to be adjusted for tests that are ported to our platform. For example

```
/* { dg-do run } */
/* { dg-options "-O2 -msse2" } */
/* { dg-require-effective-target sse2 } */
```

should become something like

```
/* { dg-do run } */
/* { dg-options "-O3 -mpower8-vector" } */
/* { dg-require-effective-target lp64 } */
/* { dg-require-effective-target p8vector_hw { target powerpc*-*-* } } */
```

Repeat this process until you have equivalent DejaGNU test implementations for all the intrinsics in that header and associated test cases that execute without error.

2.1. Preferred methods

As we will see there are multiple ways to implement the logic of these intrinsics. Some implementation methods are preferred because they allow the compiler to select instructions and provided the most flexibility for optimization across the whole sequence. Other methods may be required to deliver a specific semantic or to deliver better optimization than the current compiler is capable of. Some methods are more portable across multiple compilers (GCC, LLVM, ...). All of this should be taken into consideration for each intrinsic implementation. In general we should use the following list as a guide to these decisions:

1. Use C vector arithmetic, logical, dereference, etc., operators in preference to intrinsics.
2. Use the bi-endian interfaces from Appendix A of the ABI in preference to other intrinsics when available, as these are designed for portability among compilers.
3. Use other, less well documented intrinsics (such as `__builtin_vsx_*`) when no better facility is available, in preference to assembly.
4. If necessary, use inline assembly, but know what you're doing.

2.2. Prepare yourself

To port Intel intrinsics to POWER you will need to prepare yourself with knowledge of PowerISA vector facilities and how to access the associated documentation.

- *GCC vector extension* syntax and usage. This is one of a set of GCC "*Extensions to the C language Family*" that the intrinsic header implementation depends on. As many of the GCC intrinsics for x86 are implemented via C vector extensions, reading and understanding of this code is an important part of the porting process.
- Intel (x86) intrinsic and type naming conventions and how to find more information. The intrinsic name encodes some information about the vector size and type of the data, but the pattern is not always obvious. Using the online Intel Intrinsic Guide to look up the intrinsic by name is a good first step.
- PowerISA Vector facilities. The Vector facilities of POWER8 are extensive and cover the usual types and usual operations. However it has a different history and organization from Intel. Both (Intel and PowerISA) have their quirks and in some cases the mapping may not be obvious. So familiarizing yourself with the PowerISA Vector (VMX) and Vector Scalar Extensions (VSX) is important.

2.2.1. GCC Vector Extensions

The GCC vector extensions are common syntax but implemented in a target specific way. Using the C vector extensions requires the `__gnu_inline__` attribute to avoid syntax errors in case the user specified C standard compliance (`-std=c90`, `-std=c11`, etc) that would normally disallow such extensions.

The GCC implementation for PowerPC64 Little Endian is (mostly) functionally compatible with x86_64 vector extension usage. We can use the same type definitions (at least for vector_size (16)), operations, syntax <**`{...}`**> for vector initializers and constants, and array syntax <**`[]`**> for vector element access. So simple arithmetic / logical operations on whole vectors should work as is.

The caveat is that the interface data type of the Intel Intrinsic may not match the data types of the operation, so it may be necessary to cast the operands to the specific type for the operation. This also applies to vector initializers and accessing vector elements. You need to use the appropriate type to get the expected results. Of course this applies to X86_64 as well. For example:

```
/* Perform the respective operation on the four SPFP values in A and B.  */
extern __inline __m128 __attribute__((__gnu_inline__, __always_inline__,
 __artificial__))
_mm_add_ps (__m128 __A, __m128 __B)
{
  return (__m128) ((__v4sf)__A + (__v4sf)__B);
}

/* Stores the lower SPFP value.  */
extern __inline void __attribute__((__gnu_inline__, __always_inline__, __artificial__))
_mm_store_ss (float *__P, __m128 __A)
{
  *__P = ((__v4sf)__A)[0];
}
```

Note the cast from the interface type (`__m128`} to the implementation type (`__v4sf`, defined in the intrinsic header) for the vector float add (+) operation. This is enough for the compiler to select the appropriate vector add instruction for the float type. Then the result (which is `__v4sf`) needs to be cast back to the expected interface type (`__m128`).

[1]Here we are using logical left and logical right which will not match the PowerISA register view in Little endian. Logical left is the left most element for initializers {left, … , right}, storage order and array order where the left most element is [0].

Note also the use of *array syntax* (`__A)[0]`) to extract the lowest (left most[1]) element of a vector. The cast (`__v4sf`) insures that the compiler knows we are extracting the left most 32-bit float. The compiler insures the code generated matches the Intel behavior for PowerPC64 Little Endian.

The code generation is complicated by the fact that PowerISA vector registers are Big Endian (element 0 is the left most word of the vector) and scalar loads / stores are also to / from the right most word / dword. X86 scalar loads / stores are to / from the right most element for the XMM vector register. The PowerPC64 ELF V2 ABI mimics the X86 Little Endian behavior by placing logical element [0] in the right most element of the vector register.

This may require the compiler to generate additional instructions to place the scalar value in the expected position. Application code with extensive use of scalar (vs packed) intrinsic loads / stores should be flagged for rewrite to C code using existing scalar types (float, double, int, long, etc.). The compiler may be able the vectorize this scalar code using the native vector SIMD instruction set.

Another example is the set reverse order:

```
/* Create the vector [Z Y X W].  */
extern __inline __m128 __attribute__((__gnu_inline__, __always_inline__,
 __artificial__))
_mm_set_ps (const float __Z, const float __Y, const float __X, const float __W)
{
  return __extension__ (__m128)(__v4sf){ __W, __X, __Y, __Z };
}

/* Create the vector [W X Y Z].  */
extern __inline __m128 __attribute__((__gnu_inline__, __always_inline__,
 __artificial__))
_mm_setr_ps (float __Z, float __Y, float __X, float __W)
{
  return __extension__ (__m128)(__v4sf){ __Z, __Y, __X, __W };
}
```

Note the use of *initializer syntax* used to collect a set of scalars into a vector. Code with constant initializer values will generate a vector constant of the appropriate endian. However code with variables in the initializer can get complicated as it often requires transfers between register sets and perhaps format conversions. We can assume that the compiler will generate the correct code, but if this class of intrinsics shows up as a hot spot, a rewrite to native PPC vector built-ins may be appropriate. For example initializer of a variable replicated to all the vector fields might not be recognized as a “load and splat” and making this explicit may help the compiler generate better code.

2.2.2. Intel Intrinsic functions

So what is an intrinsic function? From Wikipedia:

> In compiler theory, an **intrinsic function** is a function available for use in a given programming language whose implementation is handled specially by the compiler. Typically, it substitutes a sequence of automatically generated instructions for the original function call, similar to an inline function. Unlike an inline function though, the compiler has an intimate knowledge of the intrinsic function and can therefore better integrate it and optimize it for the situation. This is also called builtin function in many languages.

The “Intel Intrinsics” API provides access to the many instruction set extensions (Intel Technologies) that Intel has added (and continues to add) over the years. The intrinsics provided access to new instruction capabilities before the compilers could exploit them directly. Initially these intrinsic functions where defined for the Intel and Microsoft compiler and where eventually implemented and contributed to GCC.

The Intel Intrinsics have a specific type and naming structure. In this naming structure, functions starts with a common prefix (MMX and SSE use '_mm' prefix, while AVX added the '_mm256' '_mm512' prefixes), then a short functional name ('set', 'load', 'store', 'add', 'mul', 'blend', 'shuffle', '…') and a suffix ('_pd', '_sd', '_pi32'...) with type and packing information. See Appendix B, Intel Intrinsic suffixes [36] for the list of common intrisic suffixes.

Oddly many of the MMX/SSE operations are not vectors at all. There are a lot of scalar operations on a single float, double, or long long type. In effect these are scalars that can take advantage of the larger (xmm) register space. Also in the Intel 32-bit architecture they provided IEEE754 float and double types, and 64-bit integers that did not exist or were hard to implement in the base i386/387 instruction set. These scalar operations use a suffix starting with '_s' (`_sd` for scalar double float, `_ss` scalar float, and `_si64` for scalar long long).

True vector operations use the packed or extended packed suffixes, starting with '_p' or '_ep' (`_pd` for vector double, `_ps` for vector float, and `_epi32` for vector int). The use of '_ep' seems to be reserved to disambiguate intrinsics that existed in the (64-bit vector) MMX extension from the extended (128-bit vector) SSE equivalent. For example **`_mm_add_pi32`** is a MMX operation on a pair of 32-bit integers, while **`_mm_add_epi32`** is an SSE2 operation on vector of 4 32-bit integers.

The GCC builtins for the i386.target (includes x86 and x86_64) are not the same as the Intel Intrinsics. While they have similar intent and cover most of the same functions, they use a different naming (prefixed with `__builtin_ia32_`, then function name with type suffix) and uses GCC vector type modes for operand types. For example:

```
v8qi __builtin_ia32_paddb (v8qi, v8qi)
v4hi __builtin_ia32_paddw (v4hi, v4hi)
v2si __builtin_ia32_paddd (v2si, v2si)
v2di __builtin_ia32_paddq (v2di, v2di)
```

Note

A key difference between GCC built-ins for i386 and PowerPC is that the x86 built-ins have different names of each operation and type while the PowerPC altivec built-ins tend to have a single overloaded built-in for each operation, across a set of compatible operand types.

In GCC the Intel Intrinsic header (*intrin.h) files are implemented as a set of inline functions using the Intel Intrinsic API names and types. These functions are implemented as either GCC C vector extension code or via one or more GCC builtins for the i386 target. So lets take a look at some examples from GCC's SSE2 intrinsic header emmintrin.h:

```
extern __inline __m128d __attribute__((__gnu_inline__, __always_inline__,
 __artificial__))
_mm_add_pd (__m128d __A, __m128d __B)
{
  return (__m128d) ((__v2df)__A + (__v2df)__B);
}

extern __inline __m128d __attribute__((__gnu_inline__, __always_inline__,
 __artificial__))
_mm_add_sd (__m128d __A, __m128d __B)
{
  return (__m128d)__builtin_ia32_addsd ((__v2df)__A, (__v2df)__B);
}
```

Note that the **`_mm_add_pd`** is implemented direct as GCC C vector extension code., while **`_mm_add_sd`** is implemented via the GCC builtin **`__builtin_ia32_addsd`**. From the discussion above we know the _pd suffix indicates a packed vector double while the `_sd` suffix indicates a scalar double in a XMM register.

2.2.2.1. Packed vs scalar intrinsics

So what is actually going on here? The vector code is clear enough if you know that the '+' operator is applied to each vector element. The intent of the X86 built-in is a little less clear, as the GCC documentation for `__builtin_ia32_addsd` is not very helpful (nonexistent). So perhaps the Intel Intrinsic Guide will be more enlightening. To paraphrase:

> From the `_mm_add_dp` description ; for each double float element ([0] and [1] or bits [63:0] and [128:64]) for operands a and b are added and resulting vector is returned.
>
> From the `_mm_add_sd` description ; Add element 0 of first operand (a[0]) to element 0 of the second operand (b[0]) and return the packed vector double {(a[0] + b[0]), a[1]}. Or said differently the sum of the logical left most half of the the operands are returned in the logical left most half (element [0]) of the result, along with the logical right half (element [1]) of the first operand (unchanged) in the logical right half of the result.

So the packed double is easy enough but the scalar double details are more complicated. One source of complication is that while both Instruction Set Architectures (SSE vs VSX) support scalar floating point operations in vector registers the semantics are different.

- The vector bit and field numbering is different (reversed).
  - For Intel the scalar is always placed in the low order (right most) bits of the XMM register (and the low order address for load and store).
  - For PowerISA and VSX, scalar floating point operations and Floating Point Registers (FPRs) are in the low numbered bits which is the left hand side of the vector / scalar register (VSR).
  - For the PowerPC64 ELF V2 little endian ABI we also make a point of making the GCC vector extensions and vector built-ins, appear to be little endian. So vector element 0 corresponds to the low order address and low order (right hand) bits of the vector register (VSR).
- The handling of the non-scalar part of the register for scalar operations are different.
  - For Intel ISA the scalar operations either leaves the high order part of the XMM vector unchanged or in some cases force it to 0.0.

– For PowerISA scalar operations on the combined FPR/VSR register leaves the remainder (right half of the VSR) **undefined**.

To minimize confusion and use consistent nomenclature, I will try to use the terms logical left and logical right elements based on the order they apprear in a C vector initializers and element index order. So in the vector `(__v2df){1.0, 2.0}`, The value 1.0 is the in the logical left element [0] and the value 2.0 is logical right element [1].

So lets look at how to implement these intrinsics for the PowerISA. For example in this case we can use the GCC vector extension, like so:

```
extern __inline __m128d __attribute__((__gnu_inline__, __always_inline__,
 __artificial__))
_mm_add_pd (__m128d __A, __m128d __B)
{
  return (__m128d) ((__v2df)__A + (__v2df)__B);
}

extern __inline __m128d __attribute__((__gnu_inline__, __always_inline__,
 __artificial__))
_mm_add_sd (__m128d __A, __m128d __B)
{
  __A[0] = __A[0] + __B[0];
  return (__A);
}
```

The packed double implementation operates on the vector as a whole. The scalar double implementation operates on and updates only [0] element of the vector and leaves the `__A[1]` element unchanged.   Form this source the GCC compiler generates the following code for PPC64LE target.:

The packed vector double generated the corresponding VSX vector double add (xvadddp). But the scalar implementation is a bit more complicated.

```
0000000000000720 <test_add_pd>:
 720: 07 1b 42 f0  xvadddp vs34,vs34,vs35
 ...

0000000000000740 <test_add_sd>:
 740: 56 13 02 f0  xxspltd vs0,vs34,1
 744: 57 1b 63 f0  xxspltd vs35,vs35,1
 748: 03 19 60 f0  xsadddp vs35,vs0,vs35
 74c: 57 18 42 f0  xxmrghd vs34,vs34,vs35
 ...
```

First the PPC64LE vector format, element [0] is not in the correct position for the scalar operations. So the compiler generates vector splat double (`xxspltd`) instructions to copy elements `__A[0]` and `__B[0]` into position for the VSX scalar add double (xsadddp) that follows. However the VSX scalar operation leaves the other half of the VSR undefined (which does not match the expected Intel semantics). So the compiler must generates a vector merge high double (`xxmrghd`) instruction to combine the original `__A[1]` element (from `vs34`) with the scalar add result from `vs35` element [1]. This merge swings the scalar result from `vs35[1]` element into the `vs34[0]` position, while preserving the original `vs34[1]` (from `__A[1]`) element (copied to itself).[2]

[2]Fun fact: The vector registers in PowerISA are decidedly Big Endian. But we decided to make the PPC64LE ABI behave like a Little Endian system to make application porting easier. This requires the compiler to manipulate the PowerISA vector instrinsic behind the the scenes to get the correct Little Endian results. For example the element selector [0|1] for `vec_splat` and the generation of `vec_mergeh` vs `vec_mergel` are reversed for the Little Endian.

This technique applies to packed and scalar intrinsics for the the usual arithmetic operators (add, subtract, multiply, divide). Using GCC vector extensions in these intrinsic implementations provides the compiler more opportunity to optimize the whole function.

Now we can look at a slightly more interesting (complicated) case. Square root (`sqrt`) is not an arithmetic operator in C and is usually handled with a library call or a compiler builtin. We really want to avoid a library call and want to avoid any unexpected side effects. As you see below the implementation of `_mm_sqrt_pd` and `_mm_sqrt_sd` intrinsics are based on GCC x86 built ins.

```
extern __inline __m128d __attribute__((__gnu_inline__, __always_inline__,
 __artificial__))
_mm_sqrt_pd (__m128d __A)
{
  return (__m128d)__builtin_ia32_sqrtpd ((__v2df)__A);
}

/* Return pair {sqrt (B[0]), A[1]}.  */
extern __inline __m128d __attribute__((__gnu_inline__, __always_inline__,
 __artificial__))
_mm_sqrt_sd (__m128d __A, __m128d __B)
{
  __v2df __tmp = __builtin_ia32_movsd ((__v2df)__A, (__v2df)__B);
  return (__m128d)__builtin_ia32_sqrtsd ((__v2df)__tmp);
}
```

For the packed vector sqrt, the PowerISA VSX has an equivalent vector double square root instruction and GCC provides the `vec_sqrt` builtin. But the scalar implementation involves an additional parameter and an extra move. This seems intended to mimick the propagation of the `__A[1]` input to the logical right half of the XMM result that we saw with `_mm_add_sd above`.

The instinct is to extract the low scalar (`__B[0]`) from operand `__B` and pass this to the GCC `__builtin_sqrt ()` before recombining that scalar result with `__A[1]` for the vector result. Unfortunately C language standards force the compiler to call the libm sqrt function unless `-ffast-math` is specified. The `-ffast-math` option is not commonly used and we want to avoid the external library dependency for what should be only a few inline instructions. So this is not a good option.

Thinking outside the box: we do have an inline intrinsic for a (packed) vector double sqrt that we just implemented. However we need to insure the other half of `__B` (`__B[1]`) does not cause any harmful side effects (like raising exceptions for NAN or negative values). The simplest solution is to vector splat `__B[0]` to both halves of a temporary value before taking the `vec_sqrt`. Then this result can be combined with `__A[1]` to return the final result. For example:

```
extern __inline __m128d __attribute__((__gnu_inline__, __always_inline__,
 __artificial__))
_mm_sqrt_pd (__m128d __A)
{
   return (vec_sqrt (__A));
}

extern __inline __m128d __attribute__((__gnu_inline__, __always_inline__,
 __artificial__))
_mm_sqrt_sd (__m128d __A, __m128d __B)
{
   __m128d c;
   c = _mm_sqrt_pd(_mm_set1_pd (__B[0]));
   return (_mm_setr_pd (c[0], __A[1]));
}
```

In this example we use `_mm_set1_pd` to splat the scalar `__B[0]`, before passing that vector to our `_mm_sqrt_pd` implementation, then pass the sqrt result (`c[0]`) with `__A[1]` to `_mm_setr_pd` to combine the final result. You could also use the `{c[0], __A[1]}` initializer instead of `_mm_setr_pd`.

Now we can look at vector and scalar compares that add their own complications: For example, the Intel Intrinsic Guide for `_mm_cmpeq_pd` describes comparing double elements [0|1] and returning either 0s for not equal and 1s (`0xFFFFFFFFFFFFFFFF` or long long -1) for equal. The comparison result is intended as a select mask (predicates) for selecting or ignoring specific elements in later operations. The scalar version `_mm_cmpeq_sd` is similar except for the quirk of only comparing element [0] and combining the result with `__A[1]` to return the final vector result.

The packed vector implementation for PowerISA is simple as VSX provides the equivalent instruction and GCC provides the builtin `vec_cmpeq` supporting the vector double type. However the technique of using scalar comparison operators on the `__A[0]` and `__B[0]` does not work as the C comparison operators return 0 or 1 results while we need the vector select mask (effectively 0 or -1). Also we need to watch for sequences that mix scalar floats and integers, generating if/then/else logic or requiring expensive transfers across register banks.

In this case we are better off using explicit vector built-ins for `_mm_add_sd` and `_mm_sqrt_sd` as examples. We can use `vec_splat` from element [0] to temporaries where we can safely use `vec_cmpeq` to generate the expected selector mask. Note that the `vec_cmpeq` returns a bool long type so we need to cast the result back to `__v2df`. Then use the `(__m128d){c[0], __A[1]}` initializer to combine the comparison result with the original `__A[1]` input and cast to the require interface type. So we have this example:

```
extern __inline __m128d __attribute__((__gnu_inline__, __always_inline__,
 __artificial__))
_mm_cmpeq_pd (__m128d __A, __m128d __B)
{
  return ((__m128d)vec_cmpeq (__A, __B));
}

extern __inline  __m128d  __attribute__((__gnu_inline__, __always_inline__,
 __artificial__))
_mm_cmpeq_sd(__m128d  __A, __m128d  __B)
{
 __v2df a, b, c;
 /* PowerISA VSX does not allow partial (for just left double)
  * results. So to insure we don't generate spurious exceptions
  * (from the right double values) we splat the left double
  * before we to the operation. */
 a = vec_splat(__A, 0);
 b = vec_splat(__B, 0);
 c = (__v2df)vec_cmpeq(a, b);
 /* Then we merge the left double result with the original right
  * double from __A.  */
 return ((__m128d){c[0], __A[1]});
}
```

2.2.2.2. To vec_not or not

Now lets look at a similar example that adds some surprising complexity. When we look at the negated compare forms we can not find exact matches in the PowerISA. But a little knowledge of boolean algebra can show the way to the equivalent functions.

First the X86 compare not equal case where we might expect to find the equivalent vec_cmpne builtins for PowerISA:

```
extern __inline __m128d __attribute__((__gnu_inline__, __always_inline__,
 __artificial__))
_mm_cmpneq_pd (__m128d __A, __m128d __B)
{
  return (__m128d)__builtin_ia32_cmpneqpd ((__v2df)__A, (__v2df)__B);
}

extern __inline __m128d __attribute__((__gnu_inline__, __always_inline__,
 __artificial__))
_mm_cmpneq_sd (__m128d __A, __m128d __B)
{
  return (__m128d)__builtin_ia32_cmpneqsd ((__v2df)__A, (__v2df)__B);
}
```

Well not exactly. Looking at the OpenPOWER ABI document we see a reference to `vec_cmpne` for all numeric types. But when we look in the current GCC 6 documentation we find that `vec_cmpne` is not on the list. So it is planned in the ABI, but not implemented yet.

Looking at the PowerISA 2.07B we find a VSX Vector Compare Equal to Double-Precision but no Not Equal. In fact we see only vector double compare instructions for greater than and greater than or equal in addition to the equal compare. Not only can't we find a not equal, there is no less than or less than or equal compares either.

So what is going on here? Partially this is the Reduced Instruction Set Computer (RISC) design philosophy. In this case the compiler can generate all the required compares using the existing vector instructions and simple transforms based on Boolean algebra. So `vec_cmpne(A,B)` is simply `vec_not (vec_cmpeq(A,B))`. And `vec_cmplt(A,B)` is simply `vec_cmpgt(B,A)` based on the identity A < B ***iff*** B > A. Similarly `vec_cmple(A,B)` is implemented as `vec_cmpge(B,A)`.

What a minute, there is no `vec_not()` either. Can not find it in the PowerISA, the OpenPOWER ABI, or the GCC PowerPC Altivec Built-in documentation. There is no `vec_move()` either! How can this possibly work?

This is RISC philosophy again. We can always use a logical instruction (like bit wise **and** or **or**) to effect a move, given that we also have nondestructive 3 register instruction forms. In the PowerISA most instruction have two input registers and a separate result register. So if the result register number is different from either input register then the inputs are not clobbered (nondestructive). Of course nothing prevents you from specifying the same register for both inputs or even all three registers (result and both inputs). And some times it is useful.

The statement `B = vec_or (A,A)` is is effectively a vector move/copy from `A` to `B`. And `A = vec_or (A,A)` is obviously a **nop** (no operation). In fact the PowerISA defines the preferred `nop` and register move for vector registers in this way.

The PowerISA implements the logical operators **nor** (**not or**) and **nand** (**not and**). The PowerISA provides these instruction for fixed point and vector logical operations. So `vec_not(A)` can be implemented as `vec_nor(A,A)`. So for the implementation of _mm_cmpne we propose the following:

```
extern __inline __m128d __attribute__((__gnu_inline__, __always_inline__,
 __artificial__))
_mm_cmpneq_pd (__m128d __A, __m128d __B)
```

```
{
  __v2df temp = (__v2df ) vec_cmpeq ((__v2df) __A, (__v2df)__B);
  return ((__m128d)vec_nor (temp, temp));
}
extern __inline __m128d __attribute__((__gnu_inline__, __always_inline__,
 __artificial__))
_mm_cmpneq_sd (__m128d __A, __m128d __B)
{
 __v2df a, b, c;
 a = vec_splat(__A, 0);
 b = vec_splat(__B, 0);
 c = (__v2df)vec_cmpeq(a, b);
 c = (__v2df)vec_nor(c, c);
 return ((__m128d){c[0], __A[1]});
}
```

The Intel Intrinsics also include the not forms of the relational compares:

```
extern __inline __m128d __attribute__((__gnu_inline__, __always_inline__,
 __artificial__))
_mm_cmpnlt_pd (__m128d __A, __m128d __B)
{
  return (__m128d)__builtin_ia32_cmpnltpd ((__v2df)__A, (__v2df)__B);
}

extern __inline __m128d __attribute__((__gnu_inline__, __always_inline__,
 __artificial__))
_mm_cmpnle_pd (__m128d __A, __m128d __B)
{
  return (__m128d)__builtin_ia32_cmpnlepd ((__v2df)__A, (__v2df)__B);
}
```

The PowerISA and OpenPOWER ABI, or GCC PowerPC Altivec Built-in documentation do not provide any direct equivalents to the not greater than class of compares. Again you don't really need them if you know Boolean algebra. We can use identities like {**not** (A < B) iff A >= B} and {**not** (A <= B) iff A > B}. So the PPC64LE implementation follows:

```
extern __inline __m128d __attribute__((__gnu_inline__, __always_inline__,
 __artificial__))
_mm_cmpnlt_pd (__m128d __A, __m128d __B)
{
  return ((__m128d)vec_cmpge (__A, __B));
}

extern __inline __m128d __attribute__((__gnu_inline__, __always_inline__,
 __artificial__))
_mm_cmpnle_pd (__m128d __A, __m128d __B)
{
  return ((__m128d)vec_cmpgt (__A, __B));
}
```

These patterns repeat for the scalar version of the **not** compares. And in general the larger pattern described in this chapter applies to the other float and integer types with similar interfaces.

2.2.2.3. Crossing lanes

Vector SIMD units prefer to keep computations in the same “lane” (element number) as the input elements. The only exception in the examples so far are the occasional vector splat (copy one element to all the other elements of the vector) operations. Splat is an example of the general category of “permute” operations (Intel would call this a “shuffle” or “blend”).

Permutes select and rearrange the elements of an input vector (or a concatenated pair of vectors) and deliver those selected elements, in a specific order, to a result vector. The selection and order of elements in the result is controlled by a third operand, either as a 3rd input vector or as an immediate field of the instruction.

For example, consider the Intel intrisics for Horizontal Add / Subtract added with SSE3. These instrinsics add (subtract) adjacent element pairs across a pair of input vectors, placing the sum of the adjacent elements in the result vector. For example _mm_hadd_ps which implements the operation on float:

```
 result[0] = __A[1] + __A[0];
 result[1] = __A[3] + __A[2];
 result[2] = __B[1] + __B[0];
 result[3] = __B[3] + __B[2];
```

Horizontal Add (hadd) provides an incremental vector “sum across” operation commonly needed in matrix and vector transform math. Horizontal Add is incremental as you need three hadd instructions to sum across 4 vectors of 4 elements (7 for 8 x 8, 15 for 16 x 16, …).

The PowerISA does not have a sum-across operation for float or double. We can user the vector float add instruction after we rearrange the inputs so that element pairs line up for the horizontal add. For example we would need to permute the input vectors {1, 2, 3, 4} and {101, 102, 103, 104} into vectors {2, 4, 102, 104} and {1, 3, 101, 103} before the `vec_add`. This requires two vector permutes to align the elements into the correct lanes for the vector add (to implement Horizontal Add).

The PowerISA provides generalized byte-level vector permute (vperm) based on a vector register pair (32 bytes) source as input and a (16-byte) control vector. The control vector provides 16 indexes (0-31) to select bytes from the concatenated input vector register pair (VRA, VRB). There are also predefined permutes (splat, pack, unpack, merge) operations (across element sizes) that are encoded as separate instruction op-codes or instruction immediate fields.

Unfortunately only the general `vec_perm` can provide the realignment we need for the _mm_hadd_ps operation or any of the int, short variants of hadd. For example:

```
extern __inline __m128 __attribute__((__gnu_inline__, __always_inline__,
 __artificial__))
_mm_hadd_ps (__m128 __X, __m128 __Y)
{
    __vector unsigned char xform2 = {
        0x00, 0x01, 0x02, 0x03,  0x08, 0x09, 0x0A, 0x0B,
        0x10, 0x11, 0x12, 0x13,  0x18, 0x19, 0x1A, 0x1B
      };
    __vector unsigned char xform1 = {
        0x04, 0x05, 0x06, 0x07,  0x0C, 0x0D, 0x0E, 0x0F,
        0x14, 0x15, 0x16, 0x17,  0x1C, 0x1D, 0x1E, 0x1F
      };
    return (__m128) vec_add (vec_perm ((__v4sf) __X, (__v4sf) __Y, xform1),
                             vec_perm ((__v4sf) __X, (__v4sf) __Y, xform2));
}
```

This requires two permute control vectors; one to select the even word elements across __X and __Y, and another to select the odd word elements across __X and __Y. The results of these permutes (`vec_perm`) are inputs to the `vec_add` that completes the horizontal add operation.

Fortunately the permute required for the double (64-bit) case (`_mm_hadd_pd`) reduces to the equivalent of `vec_mergeh` / `vec_mergel` doubleword (which are variants of VSX Permute Doubleword Immediate). So the implementation of _mm_hadd_pd can be simplified to this:

```
extern __inline __m128d __attribute__((__gnu_inline__, __always_inline__,
 __artificial__))
_mm_hadd_pd (__m128d __X, __m128d __Y)
{
 return (__m128d) vec_add (vec_mergeh ((__v2df) __X, (__v2df)__Y),
                     vec_mergel ((__v2df) __X, (__v2df)__Y));
}
```

This eliminates the load of the control vectors required by the previous example.

2.2.3. PowerISA Vector facilities

The PowerISA vector facilities (VMX and VSX) are extensive, but do not always provide a direct or obvious functional equivalent to the Intel Intrinsics. However not being obvious is not the same as impossible. It just requires some basic programing skills.

It is a good idea to have an overall understanding of the vector capabilities of the PowerISA. You do not need to memorize every instruction but it helps to know where to look. Both the PowerISA and OpenPOWER ABI have a specific structure and organization that can help you find what you are looking for.

It also helps to understand the relationship between the PowerISA's low level instructions and the higher abstraction of the vector intrinsics as defined by the OpenPOWER ABI's Vector Programming Interfaces and the de facto standard of GCC's PowerPC AltiVec Builtin Functions.

2.2.3.1. The PowerISA

The PowerISA Vector facilities, for historical reasons are organized at the top level by the distinction between older Vector Facility (Altivec / VMX) and the newer Vector-Scalar Floating-Point Operations (VSX).

2.2.3.1.1. The Vector Facility (VMX)

The original VMX supported SIMD integer byte, halfword, and word, and single float data types within a separate (from GPR and FPR) bank of 32 x 128-bit vector registers. The arithmetic operations like to stay within their (SIMD) lanes except where the operation changes the element data size (integer multiply) or the generalized permute operations (splat, permute, pack, unpack merge).

This is complemented by bit logical and shift / rotate instructions that operate on the vector as a whole. Some operations (permute, pack, merge, shift double, select) will select 128 bits from a pair of vectors (256-bits) and delivers a 128-bit vector result. These instructions will cross lanes or multiple registers to grab fields and assemble them into the single register result.

The PowerISA 2.07B Chapter 6. Vector Facility is organised starting with an overview (chapters 6.1-6.6):

```

6.1 Vector Facility Overview . . . . . . . . . . . . . . . . . . . . 227
```

```
6.2 Chapter Conventions. . . . . . . . . . . . . . . . . . . . . . . . . 227
6.2.1 Description of Instruction Operation . . . . . . . . . . . . . . 227
6.3 Vector Facility Registers  . . . . . . . . . . . . . . . . . . . . . 234
6.3.1 Vector Registers . . . . . . . . . . . . . . . . . . . . . . . . 234
6.3.2 Vector Status and Control Register . . . . . . . . . . . . . . . 234
6.3.3 VR Save Register . . . . . . . . . . . . . . . . . . . . . . . . 235
6.4 Vector Storage Access Operations . . . . . . . . . . . . . . . . . . 235
6.4.1 Accessing Unaligned Storage Operands . . . . . . . . . . . . . . 237
6.5 Vector Integer Operations  . . . . . . . . . . . . . . . . . . . . . 238
6.5.1 Integer Saturation . . . . . . . . . . . . . . . . . . . . . . . 238
6.6 Vector Floating-Point Operations . . . . . . . . . . . . . . . . . . 240
6.6.1 Floating-Point Overview  . . . . . . . . . . . . . . . . . . . . 240
6.6.2 Floating-Point Exceptions  . . . . . . . . . . . . . . . . . . . 240
```

Then a chapter on storage (load/store) access for vector and vector elements:

```
6.7 Vector Storage Access Instructions . . . . . . . . . . . . . . . . 242
6.7.1 Storage Access Exceptions  . . . . . . . . . . . . . . . . . . . 242
6.7.2 Vector Load Instructions . . . . . . . . . . . . . . . . . . . . 243
6.7.3 Vector Store Instructions. . . . . . . . . . . . . . . . . . . . 246
6.7.4 Vector Alignment Support Instructions. . . . . . . . . . . . . . 248
```

2.2.3.1.1.1. Vector permute and formatting instructions

The vector Permute and formatting chapter follows and is an important one to study. These operate on the byte, halfword, word (and with PowerISA 2.07 doubleword) integer types, plus special pixel type.

The shift instructions in this chapter operate on the vector as a whole at either the bit or the byte (octet) level. This is an important chapter to study for moving PowerISA vector results into the vector elements that Intel Intrinsics expect:

```
6.8 Vector Permute and Formatting Instructions . . . . . . . . . . . . 249
6.8.1 Vector Pack and Unpack Instructions  . . . . . . . . . . . . . . 249
6.8.2 Vector Merge Instructions  . . . . . . . . . . . . . . . . . . . 256
6.8.3 Vector Splat Instructions  . . . . . . . . . . . . . . . . . . . 259
6.8.4 Vector Permute Instruction . . . . . . . . . . . . . . . . . . . 260
6.8.5 Vector Select Instruction  . . . . . . . . . . . . . . . . . . . 261
6.8.6 Vector Shift Instructions  . . . . . . . . . . . . . . . . . . . 262
```

The Vector Integer instructions include the add / subtract / Multiply / Multiply Add/Sum / (no divide) operations for the standard integer types. There are instruction forms that provide signed, unsigned, modulo, and saturate results for most operations. PowerISA 2.07 extends vector integer operations to add / subtract quadword (128-bit) integers with carry and extend. This supports extended binary integer arithmetic to 256, 512-bit and beyond. There are signed / unsigned compares across the standard integer types (byte, .. doubleword); the usual bit-wise logical operations; and the SIMD shift / rotate instructions that operate on the vector elements for various integer types.

```
6.9 Vector Integer Instructions  . . . . . . . . . . . . . . . . . . . 264
6.9.1 Vector Integer Arithmetic Instructions . . . . . . . . . . . . . 264
```

```
6.9.2 Vector Integer Compare Instructions. . . . . . . . . . . . . . . 294
6.9.3 Vector Logical Instructions  . . . . . . . . . . . . . . . . . . . 300
6.9.4 Vector Integer Rotate and Shift Instructions . . . . . . . . . . 302
```

The vector [single] float instructions are grouped into this chapter. This chapter does not include the double float instructions, which are described in the VSX chapter. VSX also includes additional float instructions that operate on the whole 64 register vector-scalar set.

```
6.10 Vector Floating-Point Instruction Set . . . . . . . . . . . . . . 306
6.10.1 Vector Floating-Point Arithmetic Instructions . . . . . . . . . 306
6.10.2 Vector Floating-Point Maximum and Minimum Instructions  . . . . 308
6.10.3 Vector Floating-Point Rounding and Conversion Instructions. . . 309
6.10.4 Vector Floating-Point Compare Instructions  . . . . . . . . . . 313
6.10.5 Vector Floating-Point Estimate Instructions . . . . . . . . . . 316
```

The vector XOR based instructions are new with PowerISA 2.07 (POWER8) and provide vector-crypto and check-sum operations:

```
6.11 Vector Exclusive-OR-based Instructions  . . . . . . . . . . . . . 318
6.11.1 Vector AES Instructions . . . . . . . . . . . . . . . . . . . . 318
6.11.2 Vector SHA-256 and SHA-512 Sigma Instructions . . . . . . . . . 320
6.11.3 Vector Binary Polynomial Multiplication Instructions. . . . . . 321
6.11.4 Vector Permute and Exclusive-OR Instruction . . . . . . . . . . 323
```

The vector gather and bit permute instructions support bit-level rearrangement of bits with in the vector, while the vector versions of the count leading zeros and population count instructions are useful to accelerate specific algorithms.

```
6.12 Vector Gather Instruction . . . . . . . . . . . . . . . . . . . . 324
6.13 Vector Count Leading Zeros Instructions . . . . . . . . . . . . . 325
6.14 Vector Population Count Instructions. . . . . . . . . . . . . . . 326
6.15 Vector Bit Permute Instruction  . . . . . . . . . . . . . . . . . 327
```

The Decimal Integer add / subtract (fixed point) instructions complement the Decimal Floating-Point instructions. They can also be used to accelerate some binary to/from decimal conversions. The VSCR instructions provide access to the Non-Java mode floating-point control and the saturation status. These instructions are not normally of interest in porting Intel intrinsics.

```
6.16 Decimal Integer Arithmetic Instructions . . . . . . . . . . . . . 328
6.17 Vector Status and Control Register Instructions . . . . . . . . . 331
```

With PowerISA 2.07B (Power8) several major extensions were added to the Vector Facility:

- Vector Crypto: Under “Vector Exclusive-OR-based Instructions Vector Exclusive-OR-based Instructions”, AES [inverse] Cipher, SHA 256 / 512 Sigma, Polynomial Multiplication, and Permute and XOR instructions.
- 64-bit Integer; signed and unsigned add / subtract, signed and unsigned compare, Even / Odd 32 x 32 multiple with 64-bit product, signed / unsigned max / min, rotate and shift left/right.
- Direct Move between GPRs and the FPRs / Left half of Vector Registers.
- 128-bit integer add / subtract with carry / extend, direct support for vector `__int128` and multiple precision arithmetic.
- Decimal Integer add / subtract for 31 digit Binary Coded Decimal (BCD).

- Miscellaneous SIMD extensions: Count leading Zeros, Population count, bit gather / permute, and vector forms of eqv, nand, orc.

The rationale for these being included in the Vector Facilities (VMX) (vs Vector-Scalar Floating-Point Operations (VSX)) has more to do with how the instructions were encoded than with the type of operations or the ISA version of introduction. This is primarily a trade-off between the bits required for register selection versus the bits for extended op-code space within a fixed 32-bit instruction.

Basically accessing 32 vector registers requires 5 bits per register, while accessing all 64 vector-scalar registers require 6 bits per register. When you consider that most vector instructions require 3 and some (select, fused multiply-add) require 4 register operand forms, the impact on op-code space is significant. The larger register set of VSX was justified by queueing theory of larger HPC matrix codes using double float, while 32 registers are sufficient for most applications.

So by definition the VMX instructions are restricted to the original 32 vector registers while VSX instructions are encoded to access all 64 floating-point scalar and vector double registers. This distinction can be troublesome when programming at the assembler level, but the compiler and compiler built-ins can hide most of this detail from the programmer.

2.2.3.1.2. Vector-Scalar Floating-Point Operations (VSX)

With PowerISA 2.06 (POWER7) we extended the vector SIMD capabilities of the PowerISA:

- Extend the available vector and floating-point scalar register sets from 32 registers each to a combined register set of 64 x 64-bit scalar floating-point and 64 x 128-bit vector registers.
- Enable scalar double float operations on all 64 scalar registers.
- Enable vector double and vector float operations for all 64 vector registers.
- Enable super-scalar execution of vector instructions and support 2 independent vector floating point pipelines for parallel execution of 4 x 64-bit Floating point Fused Multiply Adds (FMAs) and 8 x 32-bit FMAs per cycle.

With PowerISA 2.07 (POWER8) we added single-precision scalar floating-point instructions to VSX. This completes the floating-point computational set for VSX. This ISA release also clarified how these operate in the Little Endian storage model.

While the focus was on enhanced floating-point computation (for High Performance Computing), VSX also extended the ISA with additional storage access, logical, and permute (merge, splat, shift) instructions. This was necessary to extend these operations to cover 64 VSX registers, and improves unaligned storage access for vectors (not available in VMX).

The PowerISA 2.07B Chapter 7. Vector-Scalar Floating-Point Operations is organized starting with an introduction and overview (chapters 7.1- 7.5) . The early sections (7.1 and 7.2) describe the layout of the 64 VSX registers and how they relate (overlap and inter-operate) to the existing floating point scalar (FPRs) and vector (VMX VRs) registers.

```
7.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . 317
7.1.1 Overview of the Vector-Scalar Extension  . . . . . . . . . . 317
7.2 VSX Registers  . . . . . . . . . . . . . . . . . . . . . . . . 318
7.2.1 Vector-Scalar Registers  . . . . . . . . . . . . . . . . . . 318
7.2.2 Floating-Point Status and Control Register . . . . . . . . . 321
```

The definitions given in “7.1.1.1 Compatibility with Category Floating-Point and Category Decimal Floating-Point Operations”, and “7.1.1.2 Compatibility with Category Vector Operations”

The instruction sets defined in Chapter 4. Floating-Point Facility and Chapter 5. Decimal Floating-Point retain their definition with one primary difference. The FPRs are mapped to doubleword element 0 of VSRs 0-31. The contents of doubleword 1 of the VSR corresponding to a source FPR specified by an instruction are ignored. The contents of doubleword 1 of a VSR corresponding to the target FPR specified by an instruction are undefined.

The instruction set defined in Chapter 6. Vector Facility [Category: Vector], retains its definition with one primary difference. The VRs are mapped to VSRs 32-63.

Note

The reference to scalar element 0 above is from the big endian register perspective of the ISA. In the PPC64LE ABI implementation, and for the purpose of porting Intel intrinsics, this is logical doubleword element 1. Intel SSE scalar intrinsics operated on logical element [0], which is in the wrong position for PowerISA FPU and VSX scalar floating-point operations. Another important note is what happens to the other half of the VSR when you execute a scalar floating-point instruction (*The contents of doubleword 1 of a VSR … are undefined.*)

The compiler will hide some of this detail when generating code for little endian vector element [] notation and most vector built-ins. For example `vec_splat (A, 0)` is transformed for PPC64LE to `xxspltd VRT,VRA,1`. What the compiler ***can not*** hide is the different placement of scalars within vector registers.

Vector registers (VRs) 0-31 overlay and can be accessed from vector scalar registers (VSRs) 32-63. The ABI also specifies that VR2-13 are used to pass parameter and return values. In some cases the same (similar) operations exist in both VMX and VSX instruction forms, while in the other cases operations only exist for VMX (byte level permute and shift) or VSX (Vector double).

So register selection that avoids unnecessary vector moves and follows the ABI while maintaining the correct instruction specific register numbering, can be tricky. The GCC register constraint annotations for Inline assembler using vector instructions are challenging, even for experts. So only experts should be writing assembler and then only in extraordinary circumstances. You should leave these details to the compiler (using vector extensions and vector built-ins) when ever possible.

The next sections gets into the details of floating point representation, operations, and exceptions. They describe the implementation details for the IEEE-754R and C/C++ language standards that most developers only access via higher level APIs. Most programmers will not need this level of detail, but it is there if needed.

```
7.3 VSX Operations . . . . . . . . . . . . . . . . . . . . . . . . 326
7.3.1 VSX Floating-Point Arithmetic Overview . . . . . . . . . . . 326
7.3.2 VSX Floating-Point Data  . . . . . . . . . . . . . . . . . . 327
7.3.3 VSX Floating-Point Execution Models  . . . . . . . . . . . . 335
7.4 VSX Floating-Point Exceptions  . . . . . . . . . . . . . . . . 338
7.4.1 Floating-Point Invalid Operation Exception . . . . . . . . . 341
7.4.2 Floating-Point Zero Divide Exception . . . . . . . . . . . . 347
7.4.3 Floating-Point Overflow Exception. . . . . . . . . . . . . . 349
7.4.4 Floating-Point Underflow Exception . . . . . . . . . . . . . 351
```

Next comes an overview of the VSX storage access instructions for big and little endian and for aligned and unaligned data addresses. This included diagrams that illuminate the differences.

```
7.5 VSX Storage Access Operations  . . . . . . . . . . . . . . . . . . 356
7.5.1 Accessing Aligned Storage Operands . . . . . . . . . . . . . . . 356
7.5.2 Accessing Unaligned Storage Operands . . . . . . . . . . . . . . 357
7.5.3 Storage Access Exceptions  . . . . . . . . . . . . . . . . . . . 358
```

Section 7.6 starts with a VSX instruction Set Summary which is the place to start to get a feel for the types and operations supported. The emphasis on floating-point, both scalar and vector (especially vector double), is pronounced. Many of the scalar and single-precision vector instructions look like duplicates of what we have seen in the Chapter 4 Floating-Point and Chapter 6 Vector facilities. The difference here is new instruction encodings to access the full 64 VSX register space.

In addition there are a small number of logical instructions included to support predication (selecting / masking vector elements based on comparison results), and a set of permute, merge, shift, and splat instructions that operate on VSX word (float) and doubleword (double) elements. As mentioned about VMX section 6.8 these instructions are good to study as they are useful for realigning elements from PowerISA vector results to the form required for Intel Intrinsics.

```
7.6 VSX Instruction Set . . . . . . . . . . . . . . . . . . . . . . .  359
7.6.1 VSX Instruction Set Summary . . . . . . . . . . . . . . . . . .  359
7.6.1.1 VSX Storage Access Instructions . . . . . . . . . . . . . . .  359
7.6.1.2 VSX Move Instructions . . . . . . . . . . . . . . . . . . . .  360
7.6.1.3 VSX Floating-Point Arithmetic Instructions  . . . . . . . . .  360
7.6.1.4 VSX Floating-Point Compare Instructions . . . . . . . . . . .  363
7.6.1.5 VSX DP-SP Conversion Instructions . . . . . . . . . . . . . .  364
7.6.1.6 VSX Integer Conversion Instructions . . . . . . . . . . . . .  364
7.6.1.7 VSX Round to Floating-Point Integer Instructions  . . . . . .  366
7.6.1.8 VSX Logical Instructions. . . . . . . . . . . . . . . . . . .  366
7.6.1.9 VSX Permute Instructions. . . . . . . . . . . . . . . . . . .  367
7.6.2 VSX Instruction Description Conventions . . . . . . . . . . . .  368
7.6.3 VSX Instruction Descriptions  . . . . . . . . . . . . . . . . .  392
```

The VSX Instruction Descriptions section contains the detail description for each VSX category instruction. The table entries from the Instruction Set Summary are formatted in the document as hyperlinks to corresponding instruction descriptions.

2.2.3.2. PowerISA Vector Intrinsics

The **OpenPOWER ELF V2 application binary interface (ABI): Chapter 6. *Vector Programming Interfaces*** and ***Appendix A. Predefined Functions for Vector Programming*** document the current and proposed vector built-ins we expect all C/C++ compilers implement.

Some of these operations are endian sensitive and the compiler needs to make corresponding adjustments as it generates code for endian sensitive built-ins. There is a good overview for this in the **OpenPOWER ABI Section *6.4. Vector Built-in Functions***.

Appendix A is organized (sorted) by built-in name, output type, then parameter types. Most built-ins are overloaded as the named operation (vec_add, vec_sub, vec_mul, vec_cmpeq, ...) applies to multiple types.

So the `vec_add` built-in applies to all the signed and unsigned integer types (char, short, in, and long) plus float and double floating-point types. The compiler looks at the parameter type to select the vector instruction (or instruction sequence) that implements the add operation on that type. The compiler infers the output result type from the operation and input parameters and will complain if the target variable type is not compatible. Some examples:

```
vector signed char vec_add (vector signed char, vector signed char);
vector unsigned char vec_add (vector unsigned char, vector unsigned char);
vector signed short vec_add (vector signed short, vector signed short);
vector unsigned short vec_add (vector unsigned short, vector unsigned short);
vector signed int vec_add (vector signed int, vector signed int);
vector unsigned int vec_add (vector unsigned int, vector unsigned int);
vector signed int vec_add (vector signed long, vector signed long);
vector unsigned int vec_add (vector unsigned long, vector unsigned long);
vector float vec_add (vector float, vector float);
vector double vec_add (vector double, vector double);
```

This is one key difference between PowerISA built-ins and Intel Intrinsics (Intel Intrinsics are not overloaded and include type information in the name). This is why it is so important to understand the vector element types and to add the appropriate type casts to get the correct results.

The de facto standard implementation in GCC is defined in the include file <`altivec.h`> and documented in the GCC online documentation in 6.59.20 PowerPC AltiVec Built-in Functions. The header file name and section title reflect the origin of the Vector Facility, but recent versions of GCC altivec.h include built-ins for newer PowerISA 2.06 and 2.07 VMX plus VSX extensions. This is a work in progress where your (older) distro GCC compiler may not include built-ins for the latest PowerISA 3.0 or ABI edition. So before you use a built-in you find in the ABI Appendix A, check the specific GCC online documentation for the GCC version you are using.

2.2.3.3. How vector elements change size and type

Most vector built ins return the same vector type as the (first) input parameters, but there are exceptions. Examples include conversions between types, compares, pack, unpack, merge, and integer multiply operations.

Converting floats to / from integer types will change the type and sometimes change the element size as well (double ↔ int and float ↔ long). For VMX the conversions are always the same size (float ↔ [unsigned] int). But VSX allows conversion of 64-bit (long or double) to from 32-bit (float or int) with the inherent size changes. The PowerISA VSX defines a 4-element vector layout where little endian elements 0, 2 are used for input/output and elements 1,3 are undefined. The OpenPOWER ABI Appendix A defines `vec_double` and `vec_float` with even/odd and high/low extensions as program aids. These are not included in GCC 7 or earlier but are planned for GCC 8.

Compare operations produce either `vector bool` <input element type> (effectively bit masks) or predicates (the condition code for all and any are represented as an int truth variable). When a predicate compare (i.e. `vec_all_eq`, `vec_any_gt`) is used in an if statement, the condition code is used directly in the conditional branch and the int truth value is not generated.

Pack operations pack integer elements into the next smaller (half) integer sized elements. Pack operations include signed and unsigned saturate and unsigned modulo forms. As the packed result will be half the size (in bits), pack instructions require 2 vectors (256-bits) as input and generate a single 128-bit vector result.

```
vec_vpkudum ({1, 2}, {101, 102}) result={1, 2, 101, 102}
```

Unpack operations expand integer elements into the next larger size elements. The integers are always treated as signed values and sign-extended. The processor design avoids instructions that return multiple register values. So the PowerISA defines unpack-high and unpack low forms where instruction takes (the high or low) half of vector elements and extends them to fill the vector output. Element order is maintained and an unpack high / low sequence with the same input vector has the effect of unpacking to a 256-bit result in two vector registers.

```
vec_vupkhsw ({1, 2, 3, 4}) result={1, 2}
vec_vupkhsw ({-1, 2, -3, 4}) result={-1, 2}
vec_vupklsw ({1, 2, 3, 4}) result={3, 4}
vec_vupklsw ({-1, 2, -3, 4}) result={-3, 4}
```

Merge operations resemble shuffling two (vectors) card decks together, alternating (elements) cards in the result. As we are merging from 2 vectors (256-bits) into 1 vector (128-bits) and the elements do not change size, we have merge high and merge low instruction forms for each (byte, halfword and word) integer type. The merge high operations alternate elements from the (vector register left) high half of the two input vectors. The merge low operation alternate elements from the (vector register right) low half of the two input vectors.

For PowerISA 2.07 we added vector merge word even / odd instructions. Instead of high or low elements the shuffle is from the even or odd number elements of the two input vectors. Passing the same vector to both inputs to merge produces splat-like results for each doubleword half, which is handy in some convert operations.

```
vec_mrghd ({1, 2}, {101, 102}) result={1, 101}
vec_mrgld ({1, 2}, {101, 102}) result={2, 102}

vec_vmrghw ({1, 2, 3, 4}, {101, 102, 103, 104}) result={1, 101, 2, 102}
vec_vmrghw ({1, 2, 3, 4}, {1, 2, 3, 4}) result={1, 1, 2, 2}
vec_vmrglw ({1, 2, 3, 4}, {101, 102, 103, 104}) result={3, 103, 4, 104}
vec_vmrglw ({1, 2, 3, 4}, {1, 2, 3, 4}) result={3, 3, 4, 4}


vec_mergee ({1, 2, 3, 4}, {101, 102, 103, 104}) result={1, 101, 3, 103}
vec_mergee ({1, 2, 3, 4}, {1, 2, 3, 4}) result={1, 1, 3, 3}
vec_mergeo ({1, 2, 3, 4}, {101, 102, 103, 104}) result={2, 102, 4, 104}
vec_mergeo ({1, 2, 3, 4}, {1, 2, 3, 4}) result={2, 2, 4, 4}
```

Integer multiply has the potential to generate twice as many bits in the product as input. A multiply of 2 int (32-bit) values produces a long (64-bits). Normal C language * operations ignore this and discard the top 32-bits of the result. However in some computations it useful to preserve the double product precision for intermediate computation before reducing the final result back to the original precision.

The PowerISA VMX instruction set took the later approach, i.e., keep all the product bits until the programmer explicitly asks for the truncated result (via the pack operation). So the vector integer multiple are split into even/odd forms across signed and unsigned byte, halfword and word inputs. This requires two instructions (given the same inputs) to generate the full vector multiply across 2 vector registers and 256-bits. Again as POWER processors are super-scalar this pair of instructions should execute in parallel.

The set of expanded product values can either be used directly in further (doubled precision) computation or merged/packed into the single single vector at the smaller bit size. This is what the compiler will generate for C vector extension multiply of vector integer types.

2.2.4. Some more intrinsic examples

The intrinsic _mm_cvtpd_ps converts a packed vector double into a packed vector single float. Since only 2 doubles fit into a 128-bit vector only 2 floats are returned and occupy only half (64-bits) of the

XMM register. For this intrinsic the 64 bits are packed into the logical left half of the result register and the logical right half of the register is set to zero (as per the Intel `cvtpd2ps` instruction).

The PowerISA provides the VSX Vector round and Convert Double-Precision to Single-Precision format (xvcvdpsp) instruction. In the ABI this is `vec_floato` (vector double). This instruction converts each double element, then transfers converted element 0 to float element 1, and converted element 1 to float element 3. Float elements 0 and 2 are undefined (the hardware can do whatever). This does not match the expected results for `_mm_cvtpd_ps`.

```
vec_floato ({1.0, 2.0}) result = {<undefined>, 1.0, <undefined>, 2.0}
_mm_cvtpd_ps  ({1.0, 2.0}) result = {1.0, 2.0, 0.0, 0.0}
```

So we need to re-position the results to word elements 0 and 2, which allows a pack operation to deliver the correct format. Here the merge-odd splats element 1 to 0 and element 3 to 2. The Pack operation combines the low half of each doubleword from the vector result and vector of zeros to generate the require format.

```
extern __inline __m128 __attribute__((__gnu_inline__, __always_inline__,
 __artificial__))
_mm_cvtpd_ps (__m128d __A)
{
 __v4sf result;
 __v4si temp;
 const __v4si vzero = {0,0,0,0};
    __asm__(
  "xvcvdpsp %x0,%x1;\n"
  : "=wa" (temp)
  : "wa" (__A)
        : );

 temp = vec_mergeo (temp, temp);
 result = (__v4sf)vec_vpkudum ((vector long)temp, (vector long)vzero);
  return (result);
}
```

This technique is also used to implement _mm_cvttpd_epi32 which converts a packed vector double into a packed vector int. The PowerISA instruction `xvcvdpsxws` uses a similar layout for the result as `xvcvdpsp` and requires the same fix up.

2.3. Profound differences

We have already mentioned above a number of architectural differences that affect porting of codes containing Intel intrinsics to POWER. The fact that Intel supports multiple vector extensions with different vector widths (64, 128, 256, and 512 bits) while the PowerISA only supports vectors of 128 bits is one issue. Another is the difference in how the respective ISAs support scalars in vector registers. In the text above we propose workable alternatives for the PowerPC port. There are also differences in the handling of floating point exceptions and rounding modes that may impact the application's performance or behavior.

2.3.1. Floating Point Exceptions

Nominally both ISAs support the IEEE-754 specifications, but there are some subtle differences. Both architectures define a status and control register to record exceptions and enable / disable floating point exceptions for program interrupt or default action. Intel has a MXCSR and PowerISA has a FPSCR which basically do the same thing but with different bit layout.

Intel provides `_mm_setcsr` / `_mm_getcsr` intrinsic functions to allow direct access to the MXCSR. This might have been useful in the early days before the OS run-times were updated to manage the MXCSR via the POSIX APIs. Today this would be highly discouraged with a strong preference to use the POSIX APIs (`feclearexceptflag`, `fegetexceptflag`, `fesetexceptflag`, ...) instead.

If we implement `_mm_setcsr` / `_mm_getcs` at all, we should simply redirect the implementation to use the POSIX APIs from `<fenv.h>`. But it might be simpler just to replace these intrinsics with macros that generate #error.

The Intel MXCSR does have some non- (POSIX/IEEE754) standard quirks: The Flush-To-Zero and Denormals-Are-Zeros flags. This simplifies the hardware response to what should be a rare condition (underflows where the result can not be represented in the exponent range and precision of the format) by simply returning a signed 0.0 value. The intrinsic header implementation does provide constant masks for `_MM_DENORMALS_ZERO_ON` (`<pmmintrin.h>`) and `_MM_FLUSH_ZERO_ON` (`<xmmintrin.h>`), so technically it is available to users of the Intel Intrinsics API.

The VMX Vector facility provides a separate Vector Status and Control register (VSCR) with a Non-Java Mode control bit. This control combines the flush-to-zero semantics for floating point underflow and denormal values. But this control only applies to VMX vector float instructions and does not apply to VSX scalar floating Point or vector double instructions. The FPSCR does define a Floating-Point non-IEEE mode which is optional in the architecture. This would apply to Scalar and VSX floating-point operations if it were implemented. This was largely intended for embedded processors and is not implemented in the POWER processor line.

As the flush-to-zero is primarily a performance enhancement and is clearly outside the IEEE-754 standard, it may be best to simply ignore this option for the intrinsic port.

2.3.2. Floating-point rounding modes

The Intel (x86 / x86_64) and PowerISA architectures both support the 4 IEEE-754 rounding modes. Again while the Intel Intrinsic API allows the application to change rounding modes via updates to the `MXCSR` it is a bad idea and should be replaced with the POSIX APIs (`fegetround` and `fesetround`).

2.3.3. Performance

The performance of a ported intrinsic depends on the specifics of the intrinsic and the context it is used in. Many of the SIMD operations have equivalent instructions in both architectures. For example the vector float and vector double match very closely. However the SSE and VSX scalars have subtle differences of how the scalar is positioned with the vector registers and what happens to the rest (non-scalar part) of the register (previously discussed in Section 2.2.2.1, “Packed vs scalar intrinsics” [17]). This requires additional PowerISA instructions to preserve the non-scalar portion of the vector registers. This may or may not be important to the logic of the program being ported, but we have to handle the case where it is.

This is where the context of how the intrinsic is used starts to matter. If the scalar intrinsics are used within a larger program the compiler may be able to eliminate the redundant register moves as the results are never used. In other cases common set up (like permute vectors or bit masks) can be common-ed up and hoisted out of the loop. So it is very important to let the compiler do its job with higher optimization levels (`-O3`, `-funroll-loops`).

2.3.3.1. Using SSE float and double scalars

For SSE scalar float / double intrinsics, "hand" optimization is no longer necessary. This was important, when SSE was initially introduced, and compiler support was limited or nonexistent. Also SSE scalar float / double provided additional (16) registers and IEEE-754 compliance, not available from the 8087 floating point architecture that preceded it. So application developers where motivated to use SSE instructions versus what the compiler was generating at the time.

Modern compilers can now generate and optimize these (SSE scalar) instructions for Intel from C standard scalar code. Of course PowerISA supported IEEE-754 float and double and had 32 dedicated floating point registers from the start (and now 64 with VSX). So replacing Intel specific scalar intrinsic implementation with the equivalent C language scalar implementation is usually a win; it allows the compiler to apply the latest optimization and tuning for the latest generation processor, and is portable to other platforms where the compiler can also apply the latest optimization and tuning for that processor's latest generation.

2.3.3.2. Using MMX intrinsics

MMX was the first and oldest SIMD extension and initially filled a need for wider (64-bit) integer and additional registers. This is back when processors were 32-bit and 8 x 32-bit registers was starting to cramp our programming style. Now 64-bit processors, larger register sets, and 128-bit (or larger) vector SIMD extensions are common. There is simply no good reason to write new code using the (now) very limited MMX capabilities.

We recommend that existing MMX codes be rewritten to use the newer SSE and VMX/VSX intrinsics or using the more portable GCC builtin vector support or in the case of si64 operations use C scalar code. The MMX si64 scalars are just (64-bit) operations on long long int types and any modern C compiler can handle this type. The char / short in SIMD operations should all be promoted to 128-bit SIMD operations on GCC builtin vectors. Both will improve cross platform portability and performance.

Appendix A. Document references

A.1. OpenPOWER and Power documents

OpenPOWER™ Technical Specifications

Power ISA™ Version 2.07 B

Power ISA™ Version 3.0

Power Architecture 64-bit ELF ABI Specification (AKA OpenPower ABI for Linux Supplement)

AltiVec™ Technology Programming Environments Manual

A.2. A.2 Intel documents

Intel® 64 and IA-32 Architectures Software Developer’s Manual

Intel™ Intrinsics Guide

A.3. A.3 GNU Compiler Collection (GCC) documents

GCC online documentation

GCC Manual

GCC Internals Manual

Appendix B. Intel Intrinsic suffixes

B.1. MMX

`_pi16`	4 x packed short int
`_pi32`	2 x packed int
`_pi8`	8 x packed signed char
`_pu16`	4 x packed unsigned short int
`_pu8`	8 x packed unsigned char
`_si64`	single 64-bit binary (logical)

B.2. SSE

`_ps`	4 x packed float
`_ss`	single scalar float
`_si32`	single 32-bit int
`_si64`	single 64-bit long int

B.3. SSE2

`_epi16`	8 x packed short int
`_epi32`	4 x packed int
`_epi64`	2 x packed long int
`_epi8`	16 x packed signed char
`_epu16`	8 x packed unsigned short int
`_epu32`	4 x packed unsigned int
`_epu8`	16 x packed unsigned char
`_pd`	2 x packed double
`_sd`	single scalar double
`_pi64`	single long int
`_si128`	single 128-bit binary (logical)

B.4. AVX/AVX2 __m256_*

`_ps`	8 x packed float
`_pd`	4 x packed double
`_epi16`	16 x packed short int
`_epi32`	8 x packed int
`_epi64`	4 x packed long int
`_epi8`	32 x packed signed char
`_epu16`	16 x packed unsigned short int
`_epu32`	8 x packed unsigned int
`_epu8`	32 x packed unsigned char
`_ss`	single scalar float (broadcast/splat)
`_sd`	single scalar double
`_si256`	single 256-bit binary (logical)
`_pd256`	cast / zero extend
`_ps256`	cast / zero extend
`_pd128`	cast
`_ps128`	cast

B.5. AVX512 __m512_*

`_ps`	16 x packed float
`_pd`	8 x packed double
`_epi16`	32 x packed short int
`_epi32`	16 x packed int
`_epi64`	8 x packed long int
`_epi8`	64 x packed signed char
`_epu16`	32 x packed unsigned short int
`_epu32`	16 x packed unsigned int
`_epu64`	8 x packed unsigned long int

`_epu8`	64 x packed unsigned char
`_ss`	single scalar float
`_sd`	single scalar double
`_si512`	single 512-bit binary (logical)
`_pd512`	cast / zero extend
`_ps512`	cast / zero extend

Appendix C. OpenPOWER Foundation overview

The OpenPOWER Foundation was founded in 2013 as an open technical membership organization that will enable data centers to rethink their approach to technology. Member companies are enabled to customize POWER CPU processors and system platforms for optimization and innovation for their business needs. These innovations include custom systems for large or warehouse scale data centers, workload acceleration through GPU, FPGA or advanced I/O, platform optimization for SW appliances, or advanced hardware technology exploitation. OpenPOWER members are actively pursing all of these innovations and more and welcome all parties to join in moving the state of the art of OpenPOWER systems design forward.

To learn more about the OpenPOWER Foundation, visit the organization website at openpowerfoundation.org.

C.1. Foundation documentation

Key foundation documents include:

- *Bylaws of OpenPOWER Foundation*
- *OpenPOWER Foundation Intellectual Property Rights (IPR) Policy*
- *OpenPOWER Foundation Membership Agreement*
- *OpenPOWER Anti-Trust Guidelines*

More information about the foundation governance can be found at openpowerfoundation.org/about-us/governance.

C.2. Technical resources

Development resouces fall into the following general categories:

- Foundation work groups
- Remote development environments (VMs)
- Development systems
- Technical specifications
- Software
- Developer tools

The complete list of technical resources are maintained on the foundation Technical Resources web page.

C.3. Contact the foundation

To learn more about the OpenPOWER Foundation, please use the following contact points:

- General information -- <`info@openpowerfoundation.org`>
- Membership -- <`membership@openpowerfoundation.org`>
- Technical Work Groups and projects -- <`tsc-chair@openpowerfoundation.org`>
- Events and other activities -- <`admin@openpowerfoundation.org`>
- Press/Analysts -- <`press@openpowerfoundation.org`>

More contact information can be found at openpowerfoundation.org/get-involved/contact-us.