POWER Vector Library Manual

1.0.4

Generated by Doxygen 1.8.17

Chapter 1

POWER Vector Library (pveclib)

A library of useful vector functions for POWER. This library fills in the gap between the instructions defined in the POWER Instruction Set Architecture (**PowerISA**) and higher level library APIs. The intent is to improve the productivity of application developers who need to optimize their applications or dependent libraries for POWER.

Authors

Steven Munroe

Copyright

1.1 Notices

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at “Copyright and trademark information” at http:www.ibm.com/legal/copytrade.shtml.

The following terms are trademarks or registered trademarks licensed by Power.org in the United States and/or other countries: Power ISA^{TM}, Power Architecture^{TM}. Information on the list of U.S. trademarks licensed by Power.org may be found at http:www.power.org/about/brand-center/.

The following terms are trademarks or registered trademarks of Freescale Semiconductor in the United States and/or other countries: AltiVec^{TM}. Information on the list of U.S. trademarks owned by Freescale Semiconductor may be found at http://www.freescale.com/files/abstract/help_page/TERMSOFUSE.html.

1.1.1 Reference Documentation

- Power Instruction Set Architecture, Versions 2.07B, 3.0B, and 3.1, IBM, 2013-2020. Available from the IBM Portal for OpenPOWER under the **Public Documents** tab.
  - Publicly available PowerISA docs for older processors are hard to find. But here is a link to PowerISA-2.06B for POWER7.
- ALTIVEC PIM: AltiVecTM Technology Programming Interface Manual, Freescale Semiconductor, 1999.
- 64-bit PowerPC ELF Application Binary Interface (ABI) Supplement 1.9.
- OpenPOWER ELF V2 application binary interface (ABI), OpenPOWER Foundation, 2017.
- Using the GNU Compiler Collection (GCC), Free Software Foundation, 1988-2018.
- What is an indirect function (IFUNC)?, glibc wiki.
- POWER8 Processor User's Manual for the Single-Chip Module.
- POWER9 Processor User's Manual.
- Warren, Henry S. Jr, Hacker's Delight, 2nd Edition, Upper Saddle River, NJ: Addison Wesley, 2013.

1.1.2 Release history

1.1.2.1 Next Release

Proposed features:

- Enable and exploit Power10 ISA instructions for both new operations and optimizations for existing operations.
  - Quadword integer shift/rotate.
  - Quadword integer signed/unsigned compare.
  - Expand mask byte/halfword/word/doubleword/quadword.
  - Extract/Insert exponent/significand for single/double/quad-precision
- Configure and build Power10 specific runtime libraries.
- Provide Vector Gather/Scatter operations.
- Provide access to the Quad-Precision operations from POWER9/10 vector implementations for POWER8

1.1.2.2 Release 1.0.4

Tagged v1.0.4 Release. This version is included in Fedora 33 and EPEL 7/8.

- Operations Implemented: 452
- Runtime library Symbols: 14
- POWER9 Specific cases: 122
- POWER8 Specific cases: 119
- GCC version specific cases: 63
- Clang specific cases: 26
- Endian Specific cases: 121

This version adds run-time libraries for large order integer multiplies (512x512, 1024x1024, and 2048x2048) with interfaces defined in vec_int512_ppc.h. These libraries support static linkage (libpvecstatic.a) to platform specific implementations using platform suffixes (ie vec_mul2048x2048_PWR9) and dynamic linkage (libpvec.so) with IFUNC platform binding (simply vec_mul2048x2048).

1.1.2.2.1 Using the CLANG compiler Application can compile with CLANG and use (most of the) the PVECLIB APIs. The APIs for vec_f128_ppc.h and vec_bcd_ppc.h are disabled or limited as CLANG does not support _Float128 nor Decimal Float types. Also CLANG can not be used to build the PVECLIB runtime libraries as it is missing the source attributes associated with the **STTGNUIFUNC symbol type extension**. But CLANG compiled applications can still link to and use these runtime functions.

1.1.2.3 Release 1.0.3

Tagged v1.0.3 for release. This version is included as package in Fedora 31.

- Operations Implemented: 386
- POWER9 Specific cases: 112
- POWER8 Specific cases: 112
- GCC version specific cases: 59
- Endian Specific cases: 87

Includes updates for vector BCD arithmetic and conversions. Also vector quadword divide/modulo by 10∗∗31 and 10∗∗32.

1.2 Rationale

The C/C++ language compilers (that support PowerISA) may implement vector intrinsic functions (compiler built-ins as embodied by altivec.h). These vector intrinsics offer an alternative to assembler programming, but do little to reduce the complexity of the underlying PowerISA. Higher level vector intrinsic operations are needed to improve productivity and encourage developers to optimize their applications for PowerISA. Another key goal is to smooth over the complexity of the evolving PowerISA and compiler support.

For example: the PowerISA 2.07 (POWER8) provides population count and count leading zero operations on vectors of byte, halfword, word, and doubleword elements but not on the whole vector as a __int128 value. Before PowerISA 2.07, neither operation was supported, for any element size.

Another example: The original **Altivec** (AKA Vector Multimedia Extension (**VMX**)) provided Vector Multiply Odd / Even operations for signed / unsigned byte and halfword elements. The PowerISA 2.07 added Vector Multiply Even/Odd operations for signed / unsigned word elements. This release also added a Vector Multiply Unsigned Word Modulo operation. This was important to allow auto vectorization of C loops using 32-bit (int) multiply.

But PowerISA 2.07 did not add support for doubleword or quadword (__int128) multiply directly. Nor did it fill in the missing multiply modulo operations for byte and halfword. However it did add support for doubleword and quadword add / subtract modulo, This can be helpful, if you are willing to apply grade school arithmetic (add, carry the 1) to vector elements.

PowerISA 3.0 (POWER9) adds a Vector Multiply-Sum Unsigned Doubleword Modulo operation. With this instruction (and a generated vector of zeros as input) you can effectively implement the simple doubleword integer multiply modulo operation in a few instructions. Similarly for Vector Multiply-Sum Unsigned Halfword Modulo. But this may not be obvious.

PowerISA 3.1 (POWER10) adds SIMD-equivalent forms of the FXU multiply, divide, and modulo instructions. Also additional 128-bit divide, modulo, rotate, shift, and conversion operations.

This history embodies a set of trade-offs negotiated between the Software and Processor design architects at specific points in time. But most programmers would prefer to use a set of operators applied across the supported element types/sizes while letting the compiler/runtime deal with the instruction level details.

1.2.1 POWER Vector Library Goals

Obviously many useful operations can be constructed from existing PowerISA operations and GCC <altivec.h> built-ins but the implementation may not be obvious. The optimum sequence will vary across the PowerISA levels as new instructions are added. And finally the compiler's built-in support for new PowerISA instructions evolves with the compiler's release cycle.

So the goal of this project is to provide well crafted implementations of useful vector and large number operations.

- Provide equivalent functions across versions of the PowerISA. This includes some of the most useful vector instructions added to POWER9 (PowerISA 3.0B) and POWER10 (PowerISA 3.1). Many of these operations can be implemented as inline function in a few vector instructions on earlier PowerISA versions.
- Provide equivalent functions across versions of the compiler. For example built-ins provided in later versions of the compiler can be implemented as inline functions with inline asm in earlier compiler versions.
- Provide complete arithmetic operations across supported C types. For example multiply modulo and even/odd for int, long, and __int128.

- Provide complete extended arithmetic (carry / extend / multiple high) operations across supported C types. For example add / subtract with carry and extend for int, long, and __int128.
- Provide higher order functions not provided directly by the PowerISA. For example:
  - Vector SIMD implementation for ASCII __isalpha, etc.
  - Vector Binary Code Decimal (BCD) Multiply/Divide/Convert.
  - Vector __int128 implementations of Count Leading/Trailing Zeros, Population Count, Shift left/right immediate.
  - Large integer (128-bit and greater) multiply/divide.
  - Vector Gather/Scatter.
- Most implementations should be small enough to inline and allow the compiler opportunity to apply common optimization techniques.
- Larger Implementations should be built into platform specific object archives and dynamic shared objects. Shared objects should use **IFUNC resolvers** to bind the dynamic symbol to best implementation for the platform (see Putting the Library into PVECLIB).

1.2.1.1 POWER Vector Library Intrinsic headers

The POWER Vector Library will be primarily delivered as C language inline functions in headers files.

- vec_common_ppc.h Typedefs and helper macros
- vec_int512_ppc.h Operations on multiple precision integer values
- vec_int128_ppc.h Operations on vector __int128 values
- vec_int64_ppc.h Operations on vector long int (64-bit) values
- vec_int32_ppc.h Operations on vector int (32-bit) values
- vec_int16_ppc.h Operations on vector short int (16-bit) values
- vec_char_ppc.h Operations on vector char (values) values
- vec_bcd_ppc.h Operations on vectors of Binary Code Decimal and Zoned Decimal values
- vec_f128_ppc.h Operations on vector _Float128 values
- vec_f64_ppc.h Operations on vector double values
- vec_f32_ppc.h Operations on vector float values

Note

> The list above is complete in the current public github as a first pass. A backlog of functions remain to be implemented across these headers. Development continues while we work on the backlog listed in: `Issue #13 TODOs`

The goal is to provide high quality implementations that adapt to the specifics of the compile target (-mcpu=) and compiler (<altivec.h>) version you are using. Initially pveclib will focus on the GCC compiler and -mcpu=[power7|power8|power9] for Linux. Testing will focus on Little Endian (**powerpc64le** for power8 and power9 targets. Any testing for Big Endian (**powerpc64** will be initially restricted to power7 and power8 targets.

Expanding pveclib support beyond this list to include:

- additional compilers (ie Clang)
- additional PPC platforms (970, power6, ...)
- Larger functions that just happen to use vector registers (Checksum, Crypto, compress/decompress, lower precision neural networks, ...)

will largely depend on additional skilled practitioners joining this project and contributing (code and platform testing) on a sustained basis.

1.2.2 How pveclib is different from compiler vector built-ins

The PowerPC vector built-ins evolved from the original `AltiVec (TM) Technology Programming Interface Manual` (PIM). The PIM defined the minimal extensions to the application binary interface (ABI) required to support the Vector Facility. This included new keywords (vector, pixel, bool) for defining new vector types, and new operators (built-in functions) required for any supporting and compliant C language compiler.

The vector built-in function support included:

- generic AltiVec operations, like vec_add()
- specific AltiVec operations (instructions, like vec_vaddubm())
- predicates computed from AltiVec operations, like vec_all_eq() which are also generic

See Background on the evolution of <altivec.h> for more details.

There are clear advantages with the compiler implementing the vector operations as built-ins:

- The compiler can access the C language type information and vector extensions to implement the function overloading required to process generic operations.
- Built-ins can be generated inline, which eliminates function call overhead and allows more compact code generation.
- The compiler can then apply higher order optimization across built-ins including: Local and global register allocation. Global common subexpression elimination. Loop-invariant code motion.
- The compiler can automatically select the best instructions for the *target* processor ISA level (from the -mcpu compiler option).

While this is an improvement over writing assembler code, it does not provide much function beyond the specific operations specified in the PowerISA. As a result the generic operations were not uniformly applied across vector element types. And this situation often persisted long after the PowerISA added instructions for wider elements. Some examples:

- Initially vec_add / vec_sub applied to float, int, short and char.
- Later compilers added support for double (with POWER7 and the Vector Scalar Extensions (VSX) facility)
- Later still, integer long (64-bit) and __int128 support (with POWER8 and PowerISA 2.07B).

But vec_mul / vec_div did not:

- Initially vec_mul applied to vector float only. Later vector double was supported for POWER7 VSX. Much later integer multiply modulo under the generic vec_mul intrinsic.
- vec_mule / vec_mulo (Multiply even / odd elements) applied to [signed | unsigned] integer short and char. Later compilers added support for vector int after POWER8 added vector multiply word instructions.
- vec_div was not included in the original PIM as Altivec (VMX) only included vector reciprocal estimate for float and no vector integer divide for any size. Later compilers added support for vec_div float / double after POWER7 (VSX) added vector divide single/double-precision instructions.

Note

> While the processor you (plan to) use, may support the specific instructions you want to exploit, the compiler you are using may not support, the generic or specific vector operations, for the element size/types, you want to use. This is common for GCC versions installed by "Enterprise Linux" distributions. They tend to freeze the GCC version early and maintain that GCC version for long term stability. One solution is to use the `IBM Advance toolchain for Linux on Power` (AT). AT is free for download and new AT versions are released yearly (usually in August) with the latest stable GCC from that spring.

This can be a frustrating situation unless you are familiar with:

- the PowerISA and how it has evolved.
- the history and philosophy behind the implementation of <altivec.h>.
- The specific level of support provided by the compiler(s) you are using.

And to be fair, this author believes, this too much to ask from your average library or application developer. A higher level and more intuitive API is needed.

1.2.2.1 What can we do about this?

A lot can be done to improve this situation. For older compilers we substitute inline assembler for missing <altivec.h> operations. For older processors we can substitute short instruction sequences as equivalents for new instructions. And useful higher level (and more intuitive) operations can be written and shared. All can be collected and provided in headers and libraries.

1.2.2.1.1 Use inline assembler carefully First the Binutils assembler is usually updated within weeks of the public release of the PowerISA document. So while your compiler may not support the latest vector operations as built-in operations, an older compiler with an updated assembler, may support the instructions as inline assembler.

Sequences of inline assembler instructions can be wrapped within C language static inline functions and placed in a header files for shared use. If you are careful with the input / output register *constraints* the GCC compiler can provide local register allocation and minimize parameter marshaling overhead. This is very close (in function) to a specific Altivec (built-in) operation.

Note

> Using GCC's inline assembler can be challenging even for the experienced programmer. The register constraints have grown in complexity as new facilities and categories were added. The fact that some (VMX) instructions are restricted to the original 32 Vector Registers (**VRs**) (the high half of the Vector-Scalar Registers **VSRs**), while others (Binary and Decimal Floating-Point) are restricted to the original 32 Floating-Point Registers (**FPRs** (overlapping the low half of the VSRs), and the new VSX instructions can access all 64 VSRs, is just one source of complexity. So it is very important to get your input/output constraints correct if you want inline assembler code to work correctly.

In-line assembler should be reserved for the first implementation using the latest PowerISA. Where possible you should use existing vector built-ins to implement specific operations for wider element types, support older hardware, or higher order operations. Again wrapping these implementations in static inline functions for collection in header files for reuse and distribution is recommended.

1.2.2.1.2 Define multi-instruction sequences to fill in gaps The PowerISA vector facility has all the instructions you need to implement extended precision operations for add, subtract, and multiply. Add / subtract with carry-out and permute or double vector shift and grade-school arithmetic is all you need.

For example the Vector Add Unsigned Quadword Modulo introduced in POWER8 (PowerISA 2.07B) can be implemented for POWER7 and earlier machines in 10-11 instructions. This uses a combination of Vector Add Unsigned Word Modulo (vadduwm), Vector Add and Write Carry-Out Unsigned Word (vaddcuw), and Vector Shift Left Double by Octet Immediate (vsldoi), to propagate the word carries through the quadword.

For POWER8 and later, C vector integer (modulo) multiply can be implemented in a single Vector Unsigned Word Modulo (**vmuluwm**) instruction. This was added explicitly to address vectorizing loops using int multiply in C language code. And some newer compilers do support generic vec_mul() for vector int. But this is not documented. Similarly for char (byte) and short (halfword) elements.

POWER8 also introduced Vector Multiply Even Signed|Unsigned Word (**vmulesw**|**vmuleuw**) and Vector Multiply Odd Signed|Unsigned Word (**vmulosw**|**vmulouw**) instructions. So you would expect the generic vec_mule and vec_mulo operations to be extended to support *vector int*, as these operations have long been supported for char and short. Sadly this is not supported as of GCC 7.3 and inline assembler is required for this case. This support was added for GCC 8.

So what will the compiler do for vector multiply int (modulo, even, or odd) for targeting power7? Older compilers will reject this as a *invalid parameter combination ....* A newer compiler may implement the equivalent function in a short sequence of VMX instructions from PowerISA 2.06 or earlier. And GCC 7.3 does support vec_mul (modulo) for element types char, short, and int. These sequences are in the 2-7 instruction range depending on the operation and element type. This includes some constant loads and permute control vectors that can be factored and reused across operations. See vec_muluwm() code for details.

Once the pattern is understood it is not hard to write equivalent sequences using operations from the original <altivec.↩ h>. With a little care these sequences will be compatible with older compilers and older PowerISA versions.

1.2.2.1.3 Define new and useful operations These concepts can be extended to operations that PowerISA and the compiler does not support yet. For example; a processor that may not have multiply even/odd/modulo of the required width (word, doubleword, or quadword). This might take 10-12 instructions to implement the next element size bigger then the current processor. A full 128-bit by 128-bit multiply with 256-bit result only requires 36 instructions on POWER8 (using multiple word even/odd) and 15 instructions on POWER9 (using vmsumudm).

Other examples include Vector Scatter/Gather operations. The PowerISA does not provide Scatter/Gather instructions. It does provide instructions to directly store/load single vector elements to/from storage. For example; vec_vlxsfdx() and vec_vstxsfdx(). Batches (in groups of 2-4) of these, combined with appropriate vector splat/merge operations, provide the effective Scatter/Gather operations:

- Storing multiple vector elements to disjoint storage locations.
- Loading multiple vector elements from disjoint storage locations.

The PowerISA does not provide for effective address computation from vector registers or elements. All Load/store instructions require scalar GPRs for Base Address and Index (offset). For 64-bit PowerISA, effective address (EA) calculations use 64-bit two's compliment addition.

This is not a serious limitation as often the element offsets are scalar constants or variables. So using multiple integer scalars as offsets for Scatter/Gather operation is reasonable (and highest performing) option. For example; vec_vglfdso() and vec_vsstfdso().

However there are times when it is useful to use vector elements as load/store offsets or array indexes. This requires a transfer of elements from a vector to scalar GPRs. When using smaller (than doubleword) elements, they are extended (signed or unsigned) to 64-bit (doubleword) before use in storage EA calculates. For example; vec_vglfddo(), and vec_vsstfddo().

Note

> This behavior is defined by PowerISA section 1.10.3 Effective Address Calculation.

If left shifts are required (to convert array indexes to offsets), 64-bit shifts are applied after the element is extended. For example; vec_vglfddsx(), vec_vglfddx(), vec_vsstfddsx(), and vec_vsstfddx().

Note

> Similar gather/scatter operations are provided for doubleword integer elements (vec_int64_ppc.h) and word integer/float elements (vec_int64_ppc.h, vec_f32_ppc.h).

These integer extension and left shift operations can be on vector elements (before transfer) or scalar values (after transfer). The best (performing) sequence will depend on the compile target's PowerISA version and micro-architecture.

Starting with Power8 the ISA provides for direct transfers from vector elements to GRPs (**Move From VSR Doubleword**). Power9 adds **Move From VSR Lower Doubleword** simplifying access to the whole (both doublewords of the) 128-bit VSR.

1.2.2.1.4 Leverage other PowerISA facilities Also many of the operations missing from the vector facility, exist in the Fixed-point, Floating-point, or Decimal Floating-point scalar facilities. There will be some loss of efficiency in the data transfer but compared to a complex operation like divide or decimal conversions, this can be a workable solution. On older POWER processors (before power7/8) transfers between register banks (GPR, FPR, VR) had to go through memory. But with the VSX facility (POWER7) FPRs and VRs overlap with the lower and upper halves of the 64 VSR registers. So FPR <-> VSR transfer are 0-2 cycles latency. And with power8 we have direct transfer (GPR <-> FPR | VR | VSR) instructions in the 4-5 cycle latency range.

For example POWER8 added Decimal (**BCD**) Add/Subtract Modulo (**bcdadd**, **bcdsub**) instructions for signed 31 digit vector values. POWER9 added Decimal Convert From/To Signed Quadword (**bcdcfsq**, **bcdctsq**) instructions. So far vector unit does not support BCD multiply / divide. But the Decimal Floating-Point (**DFP**) facility (introduced with PowerISA 2.05 and Power6) supports up to 34-digit (__Decimal128) precision and all the expected (add/subtract/multiply/divide/...) arithmetic operations. DFP also supports conversion to/from 31-digit BCD and __↩ Decimal128 precision. This is all supported with a hardware Decimal Floating-Point Unit (**DFU**).

So we can implement vec_bcdadd() and vec_bcdsub() with single instructions on POWER8, and 10-11 instructions for Power6/7. This count include the VSR <-> FPRp transfers, BCD <-> DFP conversions, and DFP add/sub. Similarly for vec_bcdcfsq() and vec_bcdctsq(). The POWER8 and earlier implementations are a bit bigger (83 and 32 instruction respectively) but even the POWER9 hardware implementation runs 37 and 23 cycles (respectively).

The vec_bcddiv() and vec_bcdmul() operations are implement by transfer/conversion to __Decimal128 and execute in the DFU. This is slightly complicated by the requirement to preserve correct fix-point alignment/truncation in the floating-point format. The operation timing runs ~100-200 cycles mostly driven the DFP multiply/divide and the number of digits involved.

Note

> So why does anybody care about BCD and DFP? Sometimes you get large numbers in decimal that you need converted to binary for extended computation. Sometimes you need to display the results of your extended binary computation in decimal. The multiply by 10 and BCD vector operations help simplify and speed-up these conversions.

1.2.2.1.5 Use clever tricks

And finally: Henry S. Warren's wonderful book Hacker's Delight provides inspiration for SIMD versions of; count leading zeros, population count, parity, etc.

1.2.2.2 So what can the Power Vector Library project do?

Clearly the PowerISA provides multiple, extensive, and powerful computational facilities that continue to evolve and grow. But the best instruction sequence for a specific computation depends on which POWER processor(s) you have or plan to support. It can also depend on the specific compiler version you use, unless you are willing to write some of your application code in assembler. Even then you need to be aware of the PowerISA versions and when specific instructions where introduced. This can be frustrating if you just want to port your application to POWER for a quick evaluation.

So you would like to start evaluating how to leverage this power for key algorithms at the heart of your application.

- But you are working with an older POWER processor (until the latest POWER box is delivered).
- Or the latest POWER machine just arrived at your site (or cloud) but you are stuck using an older/stable Linux distro version (with an older distro compiler).
- Or you need extended precision multiply for your crypto code but you are not really an assembler level programmer (or don't want to be).
- Or you would like to program with higher level operations to improve your own productivity.

Someone with the right background (knowledge of the PowerISA, assembler level programming, compilers and the vector built-ins, ...) can solve any of the issues described above. But you don't have time for this.

There should be an easier way to exploit the POWER vector hardware without getting lost in the details. And this extends beyond classical vector (Single Instruction Multiple Data (SIMD)) programming to exploiting larger data width (128-bit and beyond), and larger register space (64 x 128 Vector Scalar Registers)

1.2.2.2.1 Vector Add Unsigned Quadword Modulo example

Here is an example of what can be done:

```
static inline vui128_t
vec_adduqm (vui128_t a, vui128_t b)
{
  vui32_t t;
#ifdef _ARCH_PWR8
#ifndef vec_vadduqm
  __asm__(
      "vadduqm %0,%1,%2;"
      : "=v" (t)
      : "v" (a),
      "v" (b)
      : );
#else
  t = (vui32_t) vec_vadduqm (a, b);
#endif
#else
  vui32_t c, c2;
  vui32_t z= { 0,0,0,0};
  c = vec_vaddcuw ((vui32_t)a, (vui32_t)b);
  t = vec_vadduwm ((vui32_t)a, (vui32_t)b);
  c = vec_sld (c, z, 4);
  c2 = vec_vaddcuw (t, c);
  t = vec_vadduwm (t, c);
  c = vec_sld (c2, z, 4);
  c2 = vec_vaddcuw (t, c);
  t = vec_vadduwm (t, c);
  c = vec_sld (c2, z, 4);
  t = vec_vadduwm (t, c);
#endif
  return ((vui128_t) t);
}
```

The **_ARCH_PWR8** macro is defined by the compiler when it targets POWER8 (PowerISA 2.07) or later. This is the first processor and PowerISA level to support vector quadword add/subtract. Otherwise we need to use the vector word add modulo and vector word add and write carry-out word, to add 32-bit chunks and propagate the carries through the quadword.

One little detail remains. Support for vec_vadduqm was added to GCC in March of 2014, after GCC 4.8 was released and GCC 4.9's feature freeze. So the only guarantee is that this feature is in GCC 5.0 and later. At some point this change was backported to GCC 4.8 and 4.9 as it is included in the current GCC 4.8/4.9 documentation. When or if these backports where propagated to a specific Linux Distro version or update is difficult to determine. So support for this vector built-in dependes on the specific version of the GCC compiler, or if specific Distro update includes these specific backports for the GCC 4.8/4.9 compiler they support. The:

```
#ifndef vec_vadduqm
```

C preprocessor conditional checks if the **vec_vadduqm** is defined in <altivec.h>. If defined we can assume that the compiler implements **__builtin_vec_vadduqm** and that <altivec.h> includes the macro definition:

```
#define vec_vadduqm __builtin_vec_vadduqm
```

For **_ARCH_PWR7** and earlier we need a little grade school arithmetic using Vector Add Unsigned Word Modulo (**vadduwm**) and Vector Add and Write Carry-Out Unsigned Word (**vaddcuw**). This treats the vector __int128 as 4 32-bit binary digits. The first instruction sums each (32-bit digit) column and the second records the carry out of the high order bit of each word. This leaves the carry bit in the original (word) column, so a shift left 32-bits is needed to line up the carries with the next higher word.

To propagate any carries across all 4 (word) digits, repeat this (add / carry / shift) sequence three times. Then a final add modulo word to complete the 128-bit add. This sequence requires 10-11 instructions. The 11th instruction is a vector splat word 0 immediate, which in needed in the shift left (vsldoi) instructions. This is common in vector codes and the compiler can usually reuse this register across several blocks of code and inline functions.

For POWER7/8 these instructions are all 2 cycle latency and 2 per cycle throughput. The vadduwm / vaddcuw instruction pairs should issue in the same cycle and execute in parallel. So the expected latency for this sequence is 14 cycles. For POWER8 the vadduqm instruction has a 4 cycle latency.

Similarly for the carry / extend forms which can be combined to support wider (256, 512, 1024, ...) extended arithmetic.

See also

vec_addcuq, vec_addeuqm, and vec_addecuq

1.2.2.2.2 Vector Multiply-by-10 Unsigned Quadword example

PowerISA 3.0 (POWER9) added this instruction and it's extend / carry forms to speed up decimal to binary conversion for large numbers. But this operation is generally useful and not that hard to implement for earlier processors.

```
static inline vui128_t
vec_mul10uq (vui128_t a)
{
  vui32_t t;
#ifdef _ARCH_PWR9
  __asm__(
      "vmul10uq %0,%1;\n"
      : "=v" (t)
      : "v" (a)
      : );
#else
  vui16_t ts = (vui16_t) a;
  vui16_t t10;
  vui32_t t_odd, t_even;
  vui32_t z = { 0, 0, 0, 0 };
  t10 = vec_splat_u16(10);
#if __BYTE_ORDER__ == __ORDER_LITTLE_ENDIAN__
  t_even = vec_vmulouh (ts, t10);
  t_odd = vec_vmuleuh (ts, t10);
#else
  t_even = vec_vmuleuh(ts, t10);
  t_odd = vec_vmulouh(ts, t10);
#endif
  t_even = vec_sld (t_even, z, 2);
#ifdef _ARCH_PWR8
  t = (vui32_t) vec_vadduqm ((vui128_t) t_even, (vui128_t) t_odd);
#else
  t = (vui32_t) vec_adduqm ((vui128_t) t_even, (vui128_t) t_odd);
#endif
#endif
  return ((vui128_t) t);
}
```

Notice that under the **_ARCH_PWR9** conditional, there is no check for the specific **vec_vmul10uq** built-in. As of this writing **vec_vmul10uq** is not included in the *OpenPOWER ELF2 ABI* documentation nor in the latest GCC trunk source code.

Note

> The *OpenPOWER ELF2 ABI* does define **bcd_mul10** which (from the description) will actually generate Decimal Shift (**bcds**). This instruction shifts 4-bit nibbles (BCD digits) left or right while preserving the BCD sign nibble in bits 124-127, While this is a handy instruction to have, it is not the same operation as **vec_vmul10uq**, which is a true 128-bit binary multiply by 10. As of this writing **bcd_mul10** support is not included in the latest GCC trunk source code.

For **_ARCH_PWR8** and earlier we need a little grade school arithmetic using **Vector Multiply Even/Odd Unsigned Halfword**. This treats the vector __int128 as 8 16-bit binary digits. We multiply each of these 16-bit digits by 10, which is done in two (even and odd) parts. The result is 4 32-bit (2 16-bit digits) partial products for the even digits and 4 32-bit products for the odd digits. The vector register (independent of endian); the even product elements are higher order and odd product elements are lower order.

The even digit partial products are offset right by 16-bits in the register. If we shift the even products left 1 (16-bit) digit, the even digits are lined up in columns with the odd digits. Now we can sum across partial products to get the final 128 bit product.

Notice also the conditional code for endian around the **vec_vmulouh** and **vec_vmuleuh** built-ins:

```
#if __BYTE_ORDER__ == __ORDER_LITTLE_ENDIAN__
```

Little endian (**LE**) changes the element numbering. This also changes the meaning of even / odd and this effects the code generated by compilers. But the relationship of high and low order bytes, within multiplication products, is defined by the hardware and does not change. (See: General Endian Issues) So the pveclib implementation needs to pre-swap the even/odd partial product multiplies for LE. This in effect nullifies the even / odd swap hidden in the compilers **LE** code generation and the resulting code gives the correct results.

Now we are ready to sum the partial product *digits* while propagating the digit carries across the 128-bit product. For **_ARCH_PWR8** we can use **Vector Add Unsigned Quadword Modulo** which handles all the internal carries in hardware. Before **_ARCH_PWR8** we only have **Vector Add Unsigned Word Modulo** and **Vector Add and Write Carry-Out Unsigned Word**.

We see these instructions used in the **else** leg of the pveclib **vec_adduqm** implementation above. We can assume that this implementation is correct and tested for supported platforms. So here we use another pveclib function to complete the implementation of **Vector Multiply-by-10 Unsigned Quadword**.

Again similarly for the carry / extend forms which can be combined to support wider (256, 512, 1024, ...) extended decimal to binary conversions.

See also

vec_mul10cuq, vec_mul10euq, and vec_mul10ecuq

And similarly for full 128-bit x 128-bit multiply which combined with the add quadword carry / extended forms above can be used to implement wider (256, 512, 1024, ...) multiply operations.

See also

vec_mulluq and vec_muludq

Vector Merge Algebraic High Word example

Vector Multiply High Unsigned Word example

1.2.2.3 pveclib is not a matrix math library

The pveclib does not implement general purpose matrix math operations. These should continue to be developed and improved within existing projects (ie LAPACK, OpenBLAS, ATLAS, etc). We believe that pveclib will be helpful to implementors of matrix math libraries by providing a higher level, more portable, and more consistent vector interface for the PowerISA.

The decision is still pending on: extended arithmetic, cryptographic, compression/decompression, pattern matching / search and small vector libraries (libmvec). This author believes that the small vector math implementation should be part of GLIBC (libmvec). But the lack of optimized implementations or even good documentation and examples for these topics is a concern. This may be something that PVECLIB can address by providing enabling kernels or examples.

1.2.3 Practical considerations.

1.2.3.1 General Endian Issues

For POWER8, IBM made the explicit decision to support Little Endian (**LE**) data format in the Linux ecosystem. The goal was to enhance application code portability across Linux platforms. This goal was integrated into the OpenPOWER ELF V2 Application Binary Interface **ABI** specification.

The POWER8 processor architecturally supports an *Endian Mode* and supports both BE and LE storage access in hardware. However, register to register operations are not effected by endian mode. The ABI extends the LE storage format to vector register (logical) element numbering. See OpenPOWER ABI specification `Chapter 6. Vector Programming Interfaces` for details.

This has no effect for most altivec.h operations where the input elements and the results "stay in their lanes". For operations of the form (T[n] = A[n] op B[n]), it does not matter if elements are numbered [0, 1, 2, 3] or [3, 2, 1, 0].

But there are cases where element renumbering can change the results. Changing element numbering does change the even / odd relationship for merge and integer multiply. For **LE** targets, operations accessing even vector elements are implemented using the equivalent odd instruction (and visa versa) and inputs are swapped. Similarly for high and low merges. Inputs are also swapped for Pack, Unpack, and Permute operations and the permute select vector is inverted. The above is just a sampling of a larger list of *LE transforms*. The OpenPOWER ABI specification provides a helpful table of `Endian-Sensitive Operations`.

Note

> This means that the vector built-ins provided by altivec.h may not generate the instructions you expect.

This does matter when doing extended precision arithmetic. Here we need to maintain most-to-least significant byte order and align "digit" columns for summing partial products. Many of these operations where defined long before Little Endian was seriously considered and are decidedly Big Endian in register format. Basically, any operation where the element changes size (truncated, extended, converted, subsetted) from input to output is suspect for **LE** targets.

The coding for these higher level operations is complicated by *Little Endian* (LE) support as specified in the OpenPO↩ WER ABI and as implemented in the compilers. Little Endian changes the effective vector element numbering and the location of even and odd elements.

This is a general problem for using vectors to implement extended precision arithmetic. The multiply even/odd operations being the primary example. The products are double-wide and in BE order in the vector register. This is reinforced by the Vector Add/Subtract Unsigned Doubleword/Quadword instructions. And the products from multiply even instructions are always *numerically* higher digits then multiply odd products. The pack, unpack, and sum operations have similar issues.

This matters when you need to align (shift) the partial products or select the *numerically* high or lower portion of the products. The (high to low) order of elements for the multiply has to match the order of the largest element size used in accumulating partial sums. This is normally a quadword (vadduqm instruction).

So the element order is fixed while the element numbering and the partial products (between even and odd) will change between BE and LE. This effects splatting and octet shift operations required to align partial product for summing. These are the places where careful programming is required, to nullify the compiler's LE transforms, so we will get the correct numerical answer.

So what can the Power Vector Library do to help?

- Be aware of these mandated LE transforms and if required provide compliant inline assembler implementations for LE.
- Where required for correctness provide LE specific implementations that have the effect of nullifying the unwanted transforms.
- Provide higher level operations that help pveclib and applications code in an endian neutral way and get correct results.

See also

Endian problems with word operations

Vector Multiply-by-10 Unsigned Quadword example

1.2.3.2 Returning extended quadword results.

Extended quadword add, subtract and multiply results can exceed the width of a single 128-bit vector. A 128-bit add can produce 129-bit results. A unsigned 128-bit by 128-bit multiply result can produce 256-bit results. This is simplified for the *modulo* case where any result bits above the low order 128 can be discarded. But extended arithmetic requires returning the full precision result. Returning double wide quadword results are a complication for both RISC processor and C language library design.

1.2.3.2.1 PowerISA and Implementation. For a RISC processor, encoding multiple return registers forces hard trade-offs in a fixed sized instruction format. Also building a vector register file that can support at least one (or more) double wide register writes per cycle is challenging. For a super-scalar machine with multiple vector execution pipelines, the processor can issue and complete multiple instructions per cycle. As most operations return single vector results, this is a higher priority than optimizing for double wide results.

The PowerISA addresses this by splitting these operations into two instructions that execute independently. Here independent means that given the same inputs, one instruction does not depend on the result of the other. Independent instructions can execute out-of-order, or if the processor has multiple vector execution pipelines, can execute (issue and complete) concurrently.

The original VMX implementation had Vector Add/Subtract Unsigned Word Modulo (**vadduwm** / **vsubuwm**), paired with Vector Add/Subtract and Write Carry-out Unsigned Word (**vaddcuw** / **vsubcuw**). Most usage ignores the carry-out and only uses the add/sub modulo instructions. Applications requiring extended precision, pair the add/sub modulo with add/sub write carry-out, to capture the carry and propagate it to higher order bits.

The (four word) carries are generated into the same *word lane* as the source addends and modulo result. Propagating the carries require a separate shift (to align the carry-out with the low order (carry-in) bit of the next higher word) and another add word modulo.

POWER8 (PowerISA 2.07B) added full Vector Add/Subtract Unsigned Quadword Modulo (**vadduqm** / **vsubuqm**) instructions, paired with corresponding Write Carry-out instructions. (**vaddcuq** / **vsubcuq**). A further improvement over the word instructions was the addition of three operand *Extend* forms which combine add/subtract with carry-in (**vaddeuqm**, **vsubeuqm**, **vaddecuq** and **vsubecuq**). This simplifies propagating the carry-out into higher quadword operations.

See also

vec_adduqm, vec_addcuq, vec_addeuqm, vec_addecuq

POWER9 (PowerISA 3.0B) added Vector Multiply-by-10 Unsigned Quadword (Modulo is implied), paired with Vector Multiply-by-10 and Write Carry-out Unsigned Quadword (**vmul10uq** / **vmul10cuq**). And the *Extend* forms (**vmul10euq** / **vmul10ecuq**) simplifies the digit (0-9) carry-in for extended precision decimal to binary conversions.

See also

vec_mul10uq, vec_mul10cuq, vec_mul10euq, vec_mul10ecuq

The VMX integer multiply operations are split into multiply even/odd instructions by element size. The product requires the next larger element size (twice as many bits). So a vector multiply byte would generate 16 halfword products (256-bits in total). Requiring separate even and odd multiply instructions cuts the total generated product bits (per instruction) in half. It also simplifies the hardware design by keeping the generated product in adjacent element lanes. So each vector multiply even or odd byte operation generates 8 halfword products (128-bits) per instruction.

This multiply even/odd technique applies to most element sizes from byte up to doubleword. The original VMX supports multiply even/odd byte and halfword operations. In the original VMX, arithmetic operations where restricted to byte, halfword, and word elements. Multiply halfword products fit within the integer word element. No multiply byte/halfword modulo instructions were provided, but could be implemented via a vmule, vmulo, vperm sequence.

POWER8 (PowerISA 2.07B) added multiply even/odd word and multiply modulo word instructions.

See also

vec_muleuw, vec_mulouw, vec_muluwm

The latest PowerISA (3.0B for POWER9) does add a doubleword integer multiply via **Vector Multiply-Sum unsigned Doubleword Modulo**. This is a departure from the Multiply even/odd byte/halfword/word instructions available in earlier Power processors. But careful conditioning of the inputs can generate the equivalent of multiply even/odd unsigned doubleword.

See also

vec_msumudm, vec_muleud, vec_muloud

This (multiply even/odd) technique breaks down when the input element size is quadword or larger. A quadword integer multiply forces a different split. The easiest next step would be a high/low split (like the Fixed-point integer multiply). A multiply low (modulo) quadword would be a useful function. Paired with multiply high quadword provides the double quadword product. This would provide the basis for higher (multi-quadword) precision multiplies.

See also

vec_mulluq, vec_muludq

1.2.3.2.2 C Language restrictions. The Power Vector Library is implemented using C language (inline) functions and this imposes its own restrictions. Standard C language allows an arbitrary number of formal parameters and one return value per function. Parameters and return values with simple C types are normally transfered (passed / returned) efficiently in local (high performance) hardware registers. Aggregate types (struct, union, and arrays of arbitrary size) are normally handled by pointer indirection. The details are defined in the appropriate Application Binary Interface (ABI) documentation.

The POWER processor provides lots of registers (96) so we want to use registers wherever possible. Especially when our application is composed of collections of small functions. And more especially when these functions are small enough to inline and we want the compiler to perform local register allocation and common subexpression elimination optimizations across these functions. The PowerISA defines 3 kinds of registers;

- General Purpose Registers (GPRs),
- Floating-point Registers (FPRs),
- Vector registers (VRs),

with 32 of each kind. We will ignore the various special registers for now.

The PowerPC64 64-bit ELF (and OpenPOWER ELF V2) ABIs normally pass simple arguments and return values in a single register (of the appropriate kind) per value. Arguments of aggregate types are passed as storage pointers in General Purpose Registers (GPRs).

The language specification, the language implementation, and the ABI provide some exceptions. The C99 language adds _Complex floating types which are composed of real and imaginary parts. GCC adds _Complex integer types. For PowerPC ABIs complex values are held in a pair of registers of the appropriate kind. C99 also adds double word integers as the *long long int* type. This only matters for PowerPC 32-bit ABIs. For PowerPC64 ABIs *long long* and *long* are both 64-bit integers and are held in 64-bit GPRs.

GCC also adds the __int128 type for some targets including the PowerPC64 ABIs. Values of __int128 type are held (for operations, parameter passing and function return) in 64-bit GPR pairs. Starting with version 4.9 GCC supports the vector signed/unsigned __int128 type. This is passed and returned as a single vector register and should be used for all 128-bit integer types (bool/signed/unsigned).

GCC supports __ibm128 and _Decimal128 floating point types which are held in Floating-point Registers pairs. These are distinct types from vector double and oriented differently in the VXS register file. But the doubleword halves can be moved between types using the VSX permute double word immediate instructions (xxpermdi). This useful for type conversions and implementing some vector BCD operations.

GCC recently added the __float128 floating point type which are held in single vector register. The compiler considers this to be floating scalar and is not cast compatible with any vector type. To access the __float128 value as a vector it must be passed through a union.

Note

The implementation will need to provide transfer functions between vectors and other 128-bit types.

GCC defines Generic Vector Extensions that allow typedefs for vectors of various element sizes/types and generic SIMD (arithmetic, logical, and element indexing) operations. For PowerPC64 ABIs this is currently restricted to 16-byte vectors as defined in <altivec.h>. For currently available compilers attempts to define vector types with larger (32 or 64 byte) *vector_size* values are treated as arrays of scalar elements. Only vector_size(16) variables are passed and returned in vector registers.

The OpenPOWER 64-Bit ELF V2 ABI Specification makes specific provisions for passing/returning *homogeneous aggregates* of multiple like (scalar/vector) data types. Such aggregates can be passed/returned as up to eight floating-point or vector registers. A parameter list may include multiple *homogeneous aggregates* with up to a total of twelve parameter registers.

This is defined for the Little Endian ELF V2 ABI and is not applicable to Big Endian ELF V1 targets. Also GCC versions before GCC8, do not fully implement this ABI feature, and revert to old ABI structure passing (passing through storage).

Passing large *homogeneous aggregates* becomes the preferred solution as PVECLIB starts to address wider (256 and 512-bit) vector operations. For example the ABI allows passing up to 3 512-bit parameters and return a 1024-bit result in vector registers (as in vec_madd512x512a512_inline()). For large multi-quadword precision operations the only practical solution uses reference parameters to arrays or structs in storage (as in vec_mul2048x2048()). See vec_int512_ppc.h for more examples.

So we have shown that there are mechanisms for functions to return multiple vector register values.

1.2.3.2.3 Subsetting the problem. We can simplify this problem by remembering that:

- Only a subset of the pveclib functions need to return more than one 128-bit vector.
- The PowerISA normally splits these cases into multiple instructions anyway.
- Most of these functions are small and fully inlined.
- The exception will be the multiple quadword precision arithmetic operations.

So we have two (or three) options given the current state of GCC compilers in common use:

- Mimic the PowerISA and split the operation into two functions, where each function only returns (up to) 128-bits of the result.
- Use pointer parameters to return a second vector value in addition to the function return.
- Support both options above and let the user decide which works best.
- With a availability of GCC 8/9 compilers, pass/return 256, 512 and 1024-bit vectors as *homogeneous aggregates*.

The add/subtract quadword operations provide good examples. For exmaple adding two 256-bit unsigned integer values and returning the 257-bit (the high / low sum and the carry)result looks like this:

```
s1 = vec_vadduqm (a1, b1); // sum low 128-bits a1+b1
c1 = vec_vaddcuq (a1, b1); // write-carry from low a1+b1
s0 = vec_vaddeuqm (a0, b0, c1); // Add-extend high 128-bits a0+b0+c1
c0 = vec_vaddecuq (a0, b0, c1); // write-carry from high a0+b0+c1
```

This sequence uses the built-ins from <altivec.h> and generates instructions that will execute on POWER8 and POWER9. The compiler must target POWER8 (-mcpu=power8) or higher. In fact the compile will fail if the target is POWER7.

Now let's look at the pveclib version of these operations from <vec_int128_ppc.h>:

```
s1 = vec_adduqm (a1, b1); // sum low 128-bits a1+b1
c1 = vec_addcuq (a1, b1); // write-carry from low a1+b1
s0 = vec_addeuqm (a0, b0, c1); // Add-extend high 128-bits a0+b0+c1
c0 = vec_addecuq (a0, b0, c1); // write-carry from high a0+b0+c1
```

Looks almost the same but the operations do not use the 'v' prefix on the operation name. This sequence generates the same instructions for (-mcpu=power8) as the <altivec.h> version above. It will also generate a different (slightly longer) instruction sequence for (-mcpu=power7) which is functionally equivalent.

The pveclib <vec_int128_ppc.h> header also provides a coding style alternative:

```
s1 = vec_addcq (&c1, a1, b1);
s0 = vec_addeq (&c0, a0, b0, c1);
```

Here vec_addcq combines the adduqm/addcuq operations into a *add and carry quadword* operation. The first parameter is a pointer to vector to receive the carry-out while the 128-bit modulo sum is the function return value. Similarly vec_addeq combines the addeuqm/addecuq operations into a *add with extend and carry quadword* operation.

As these functions are inlined by the compiler the implied store / reload of the carry can be converted into a simple register assignment. For (-mcpu=power8) the compiler should generate the same instruction sequence as the two previous examples.

For (-mcpu=power7) these functions will expand into a different (slightly longer) instruction sequence which is functionally equivalent to the instruction sequence generated for (-mcpu=power8).

For older processors (power7 and earlier) and under some circumstances instructions generated for this "combined form" may perform better than the "split form" equivalent from the second example. Here the compiler may not recognize all the common subexpressions, as the "split forms" are expanded before optimization.

1.3 Background on the evolution of <altivec.h>

The original `AltiVec (TM) Technology Programming Interface Manual` defined the minimal vector extensions to the application binary interface (ABI), new keywords (vector, pixel, bool) for defining new vector types, and new operators (built-in functions).

- generic AltiVec operations, like vec_add()
- specific AltiVec operations (instructions, like vec_addubm())
- predicates computed from a AltiVec operation like vec_all_eq()

A generic operation generates specific instructions based on the types of the actual parameters. So a generic vec_add operation, with vector char parameters, will generate the (specific) vector add unsigned byte modulo (vaddubm) instruction. Predicates are used within if statement conditional clauses to access the condition code from vector operations that set Condition Register 6 (vector SIMD compares and Decimal Integer arithmetic and format conversions).

The PIM defined a set of compiler built-ins for vector instructions (see section "4.4 Generic and Specific AltiVec Operations") that compilers should support. The document suggests that any required typedefs and supporting macro definitions be collected into an include file named <altivec.h>.

The built-ins defined by the PIM closely match the vector instructions of the underlying PowerISA. For example: vec_mul, vec_mule / vec_mulo, and vec_muleub / vec_muloub.

- vec_mul is defined for float and double and will (usually) generate a single instruction for the type. This is a simpler case as floating point operations usually stay in their lanes (result elements are the same size as the input operand elements).
- vec_mule / vec_mulo (multiply even / odd) are defined for integer multiply as integer products require twice as many bits as the inputs (the results don't stay in their lane).

The RISC philosophy resists and POWER Architecture avoids instructions that write to more than one register. So the hardware and PowerISA vector integer multiply generate even and odd product results (from even and odd input elements) from two instructions executing separately. The PIM defines and compiler supports these operations as overloaded built-ins and selects the specific instructions based on the operand (char or short) type.

As the PowerISA evolved adding new vector (VMX) instructions, new facilities (Vector Scalar Extended (VSX)), and specialized vector categories (little endian, AES, SHA2, RAID), some of these new operators were added to <altivec.↩ h>. This included some new specific and generic operations and additional vector element types (long (64-bit) int, __int128, double and quad precision (__Float128) float). This support was *staged* across multiple compiler releases in response to perceived need and stake-holder requests.

The result was a patchwork of <altivec.h> built-ins support versus new instructions in the PowerISA and shipped hardware. The original Altivec (VMX) provided Vector Multiply (Even / Odd) operations for byte (char) and halfword (short) integers. Vector Multiply Even / Odd Word (int) instructions were not introduced until PowerISA V2.07 (POWE↩ R8) under the generic built-ins vec_mule, vec_mulo. PowerISA 2.07 also introduced Vector Multiply Word Modulo under the generic built-in vec_mul. Both where first available in GCC 8. Specific built-in forms (vec_vmuleuw, vec_vmulouw, vec_vmuluwm) where not provided. PowerISA V3.0 (POWER9) added Multiply-Sum Unsigned Doubleword Modulo but neither generic (vec_msum) or specific (vec_msumudm) forms have been provided (so far as of GCC 9).

However the original PIM documents were primarily focused on embedded processors and were not updated to include the vector extensions implemented by the server processors. So any documentation for new vector operations were relegated to the various compilers. This was a haphazard process and some divergence in operation naming did occur between compilers.

In the run up to the POWER8 launch and the OpenPOWER initiative it was recognized that switching to Little Endian would require and new and well documented Application Binary Interface (**ABI**). It was also recognized that new <altivec.h> extensions needed to be documented in a common place so the various compilers could implement a common vector built-in API. So ...

1.3.1 The ABI is evolving

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 to implement for the PowerISA.

The ABI defined generic operations as overloaded built-in functions. Here the ABI suggests a specific PowerISA implementation based on the operand (vector element) types. The ABI also defines the (big/little) endian behavior and the ABI may suggests different instructions based on the endianness of the target.

This is an important point as the vector element numbering changes between big and little endian, and so does the meaning of even and odd. Both affect what the compiler supports and the instruction sequence generated.

- **vec_mule** and **vec_mulo** (multiply even / odd are examples of generic built-ins defined by the ABI. One would assume these built-ins will generate the matching instruction based only on the input vector type, however the GCC compiler will adjust the generated instruction based on the target endianness (reversing even / odd for little endian).
- Similarly for the merge (even/odd high/low) operations. For little endian the compiler reverses even/odd (high/low) and swaps operands as well.
- See **Table 6.1. Endian-Sensitive Operations** for details.

The many existing specific built-ins (where the name includes explicit type and signed/unsigned notation) are included in the ABI but listed as deprecated. Specifically the Appendix **A.6. Deprecated Compatibility Functions** and **Table A.8. Functions Provided for Compatibility**.

This reflects an explicit decision by the ABI and compiler maintainers that a generic only interface would be smaller/easier to implement and document as the PowewrISA evolves.

Certainly the addition of VSX to POWER7 and the many vector extensions added to POWER8 and POWER9 added hundreds of vector instructions. Many of these new instructions needed build-ins to:

- Enable early library exploitations. For example new floating point element sizes (double and Float128).
- Support specialized operations not generally supported in the language. For example detecting Not-a-Number and Infinities without triggering exceptions. These are needed in the POSIX library implementation.
- Supporting wider integer element sizes can result in large multiples of specific built-ins if you include variants for:
  - signed and unsigned
  - saturated
  - even, odd, modulo, write-carry, and extend
  - high and low
  - and additional associated merge, pack, unpack, splat, operations

So implementing new instructions as generic built-ins first, and delaying the specific built-in permutations, is a wonderful simplification. This moves naturally from tactical to strategy to plan quickly. Dropping the specific built-ins for new instructions and deprecating the existing specific built-ins saves a lot of work.

As the ABI places more emphasis on generic built-in operations, we are seeing more cases where the compiler generates multiple instruction sequences. The first example was vec_abs (vector absolute value) from the original Altivec PIM. There was no vector absolute instruction for any of the supported types (including vector float at the time). But this could be implemented in a 3 instruction sequence. This generic operation was extended to vector double for VSX (PowerISA 2.06) which introduced hardware instructions for absolute value of single and double precision vectors. But vec_abs remains a multiple instruction sequence for integer elements.

Another example is vec_mul. POWER8 (PowerISA 2.07) introduced Vector Multiply Unsigned Word Modulo (vmuluwm). This was included in the ISA as it simplified vectorizing C language (int) loops. This also allowed a single instruction implementation for vec_mul for vector (signed/unsigned) int. The PowerISA does not provide direct vector multiply modulo instructions for char, short, or long. Again this requires a multiple-instruction sequence to implement.

1.3.2 The current <altivec.h> is a mixture

The current vector ABI implementation in the compiler and <altivec.h> is mixture of old and new.

- Many new instruction (since PowerISA 2.06) are supported only under existing built-ins (with new element types; vec_mul, vec_mule, vec_mulo). Or as newly defined generic built-ins (vec_eqv. vec_nand, vec_orc).
  - Specific types/element sizes under these generic built-ins may be marked *phased in*.
- Some new instructions are supported with both generic (vec_popcnt) and specific built-ins (vec_vpopcntb, vec_vpopcntd, vec_vpopcnth, vec_vpopcntw).
- Other new instructions are only supported with specific built-ins (vec_vaddcuq, vec_vaddecuq, vec_vaddeuqm, vec_vsubcuq, vec_vsubecuq, vec_vsubeuqm). To be fair only the quadword element supports the write-carry and extend variants.
- Endian sensitivity may be applied in surprising ways.
  - **vec_muleub** and **vec_muloub** (multiply even / odd unsigned byte) are examples of non-overloaded built-ins provided by the GCC compiler but not defined in the ABI. One would assume these built-ins will generate the matching instruction, however the GCC compiler will adjust the generated instruction based on the target endianness (even / odd is reversed for little endian).
  - **vec_sld**, **vec_sldw**, **vec_sll**, and **vec_slo** (vector shift left) are **not** endian sensitive. Historically, these built-ins are often used to shift by amounts not a multiple of the element size, across types.
- A number of built-ins are defined in the ABI and marked (all or in part) as *phased in*. This implies that compilers **shall** implement these built-ins (eventually) in <altivec.h>. However the specific compiler version you are using many not have implemented it yet.

1.3.3 Best practices

This is a small sample of the complexity we encounter programming at this low level (vector intrinsic) API. This is also an opportunity for a project like the Power Vector Library (PVECLIB) to smooth off the rough edges and simplify software development for the OpenPOWER ecosystem.

If the generic vector built-in operation you need:

- is defined in the ABI, and
- defined in the PowerISA across the processor versions you need to support, and
- defined in <altivec.h> for the compilers and compiler versions you expect to use, and
- implemented for the vector types/element sizes you need for the compilers and compiler versions you expect to use.

Then use the generic vector built-in from <altivec.h> in your application/library.

Otherwise if the specific vector built-in operation you need is defined in <altivec.h>:

- For the vector types/element sizes you need, and
- defined in the PowerISA across the processor versions you need to support, and

- implemented for the compilers and compiler versions you expect to use.

Then use the specific vector built-in from <altivec.h> in your application/library.

Otherwise if the vector operation you need is defined in PVECLIB.

- For the vector types/element sizes you need.

Then use the vector operation from PVECLIB in your application/library.

Otherwise

- Check on `https://github.com/open-power-sdk/pveclib` and see if there is newer version of PVECLIB.
- Open an issue on `https://github.com/open-power-sdk/pveclib/issues` for the operation you would like to see.
- Look at source for PVECLIB for examples similar to what you are trying to do.

1.4 Putting the Library into PVECLIB

Until recently (as of v1.0.3) PVECLIB operations were **static inline** only. This was reasonable as most operations were small (one to a few vector instructions). This offered the compiler opportunity for:

- Better register allocation.
- Identifying common subexpressions and factoring them across operation instances.
- Better instruction scheduling across operations.

Even then, a few operations (quadword multiply, BCD multiply, BCD <-> binary conversions, and some POWER8/7 implementations of POWER9 instructions) were getting uncomfortably large (10s of instructions). But it was the multiple quadword precision operations that forced the issue as they can run to 100s and sometimes 1000s of instructions. So, we need to build some functions from pveclib into a static archive and/or a dynamic library (DSO).

1.4.1 Building Multi-target Libraries

Building libraries of compiled binaries is not that difficult. The challenge is effectively supporting multiple processor (POWER7/8/9) targets, as many PVECLIB operations have different implementations for each target. This is especially evident on the multiply integer word, doubleword, and quadword operations (see; vec_muludq(), vec_mulhuq(), vec_mulluq(), vec_vmuleud(), vec_vmuloud(), vec_msumudm(), vec_muleuw(), vec_mulouw()).

This is dictated by both changes in the PowerISA and in the micro-architecture as it evolved across processor generations. So an implementation to run on a POWER7 is necessarily restricted to the instructions of PowerISA 2.06. But if we are running on a POWER9, leveraging new instructions from PowerISA 3.0 can yield better performance than the POWER7 compatible implementation. When we are dealing with larger operations (10s and 100s of instructions) the compiler can schedule instruction sequences based on the platform (-mtune=) for better performance.

So, we need to deliver multiple implementations for some operations and we need to provide mechanisms to select a specific target implementation statically at compile/build or dynamically at runtime. First we need to compile multiple version of these operations, as unique functions, each with a different effective compile target (-mcpu= options).

Obviously, creating multiple source files implementing the same large operation, each supporting a different specific target platform, is a possibility. However, this could cause maintenance problems where changes to a operation must be coordinated across multiple source files. This is also inconsistent with the current PVECLIB coding style where a file contains an operation's complete implementation, including documentation and target specific implementation variants.

The current PVECLIB implementation makes extensive use of C Preprocessor (**CPP**) conditional source code. These includes testing for; compiler version, target endianness, and current target processor, then selects the appropriate source code snippet (So what can the Power Vector Library project do?). This was intended to simplify the application/library developer's life were they could use the PVECLIB API and not worry about these details.

So far, this works as intended (single vector source for multiple PowerISA VMX/VSX targets) when the entire application is compiled for a single target. However, this dependence on CPP conditionals is mixed blessing then the application needs to support multiple platforms in a single package.

1.4.1.1 The mechanisms available

The compiler and ABI offer options that at first glance seem to allow multiple target specific binaries from a single source. Besides the compiler's command level target options a number of source level mechanisms to change the target. These include:

- __ attribute __ (target ("cpu=power8"))
- __ attribute __ (target_clones ("cpu=power9,default"))
- #pragma GCC target ("cpu=power8")
- multiple compiles with different command line options (i.e. -mcpu=)

The target and target_clones attributes are function attributes (apply to single function). The target attribute overrides the command line -mcpu= option. However it is not clear which version of GCC added explicit support for (target ("cpu="). This was not explicitly documented until GCC 5. The target_clones attribute will cause GCC will create two (or more) function clones, one (or more) compiled with the specified cpu= target and another with the default (or command line -mcpu=) target. It also creates a resolver function that dynamically selects a clone implementation suitable for current platform architecture. This PowerPC specific variant was not explicitly documented until GCC 8.

There are a few issues with function attributes:

- The Doxygen preprocessor can not parse function attributes without a lot of intervention.
- The availability of these attributes seems to be limited to the latest GCC compilers.

Note

> The Clang/LLVM compilers don't provide equivalents to attribute (target) or #pragma target.

But there is a deeper problem related to the usage of CPP conditionals. Many PVECLIB operation implementations depend on GCC/compiler predefined macros including:

- __ GNUC __
- __ GNUC_MINOR __
- __ BYTE_ORDER __
- __ ORDER_LITTLE_ENDIAN __
- __ ORDER_BIG_ENDIAN __

PVECLIB also depends on many system-specific predefined macros including:

- __ ALTIVEC __
- __ VSX __
- __ FLOAT128 __
- _ARCH_PWR9
- _ARCH_PWR8
- _ARCH_PWR7

PVECLIB also depends on the <altivec.h> include file which provides the mapping between the ABI defined intrinsics and compiler defined built-ins. In some places PVECLIB conditionally tests if specific built-in is defined and substitutes an in-line assembler implementation if not. Altivec.h also depends on system-specific predefined macros to enable/disable blocks of intrinsic built-ins based on PowerISA level of the compile target.

1.4.1.2 Some things just do not work

This issue is the compiler (GCC at least) only expands the compiler and system-specific predefined macros once per source file. The preprocessed source does not change due to embedded function attributes that change the target. So the following does not work as expected.

```
#include <altivec.h>
#include <pveclib/vec_int128_ppc.h>
#include <pveclib/vec_int512_ppc.h>
// Defined in vec_int512_ppc.h but included here for clarity.
static inline __VEC_U_256
vec_mul128x128_inline (vui128_t a, vui128_t b)
{
  __VEC_U_256 result;
  // vec_muludq is defined in vec_int128_ppc.h
  result.vx0 = vec_muludq (&result.vx1, a, b);
  return result;
}
__VEC_U_256 __attribute__(target ("cpu=power7"))
vec_mul128x128_PWR7 (vui128_t m1l, vui128_t m2l)
{
  return vec_mul128x128_inline (m1l, m2l);
}
__VEC_U_256 __attribute__(target ("cpu=power8"))
vec_mul128x128_PWR8 (vui128_t m1l, vui128_t m2l)
```

```
{
  return vec_mul128x128_inline (m1l, m2l);
}
__VEC_U_256 __attribute__(target ("cpu=power9"))
vec_mul128x128_PWR9 (vui128_t m1l, vui128_t m2l)
{
  return vec_mul128x128_inline (m1l, m2l);
}
```

For example if we assume that the compiler default is (or the command line specifies) -mcpu=power8 the compiler will use this to generate the system-specific predefined macros. This is done before the first include file is processed. In this case <altivec.h>, vec_int128_ppc.h, and vec_int512_ppc.h source will be expanded for power8 (PowerISA-2.07). The result is the vec_muludq and vec_muludq inline source implementations will be the power8 specific version.

This will all be established before the compiler starts to parse and generate code for vec_mul128x128_PWR7. This compile is likely to fail because we are trying to compile code containing power8 instructions for a -mcpu=power7 target.

The compilation of vec_mul128x128_PWR8 should work as we are compiling power8 code with a -mcpu=power8 target. The compilation of vec_mul128x128_PWR9 will compile without error but will generate essentially the same code as vec_mul128x128_PWR8. The target("cpu=power9") allows that compiler to use power9 instructions but the expanded source coded from vec_muludq and vec_mul128x128_inline will not contain any power9 intrinsic built-ins.

Note

> The GCC attribute **target_clone** has the same issue.

Pragma GCC target has a similar issue if you try to change the target multiple times within the same source file.

```
#include <altivec.h>
#include <pveclib/vec_int128_ppc.h>
#include <pveclib/vec_int512_ppc.h>
// Defined in vec_int512_ppc.h but included here for clarity.
static inline __VEC_U_256
vec_mul128x128_inline (vui128_t a, vui128_t b)
{
  __VEC_U_256 result;
  // vec_muludq is defined in vec_int128_ppc.h
  result.vx0 = vec_muludq (&result.vx1, a, b);
  return result;
}
#pragma GCC push_options
#pragma GCC target ("cpu=power7")
__VEC_U_256
vec_mul128x128_PWR7 (vui128_t m1l, vui128_t m2l)
{
  return vec_mul128x128_inline (m1l, m2l);
}
#pragma GCC pop_options
#pragma GCC push_options
#pragma GCC target ("cpu=power8")
__VEC_U_256
vec_mul128x128_PWR8 (vui128_t m1l, vui128_t m2l)
{
  return vec_mul128x128_inline (m1l, m2l);
}
#pragma GCC pop_options
#pragma GCC push_options
#pragma GCC target ("cpu=power9")
__VEC_U_256
vec_mul128x128_PWR9 (vui128_t m1l, vui128_t m2l)
{
  return vec_mul128x128_inline (m1l, m2l);
}
```

This has the same issues as the target attribute example above. However you can use #pragma GCC target if;

- it proceeds the first #include in the source file.

- there is only one target #pragma in the file.

For example:

```
#pragma GCC target ("cpu=power9")
#include <altivec.h>
#include <pveclib/vec_int128_ppc.h>
#include <pveclib/vec_int512_ppc.h>
// vec_mul128x128_inline is defined in vec_int512_ppc.h
__VEC_U_256
vec_mul128x128_PWR9 (vui128_t m1l, vui128_t m2l)
{
  return vec_mul128x128_inline (m1l, m2l);
}
```

In this case the cpu=power9 option is applied before the compiler reads the first include file and initializes the system-specific predefined macros. So the CPP source expansion reflects the power9 target.

Note

> So far the techniques described only work reliably for C/C++ codes, compiled with GCC, that don't use <altivec.h> intrinsics or use CPP conditionals.

The implication is we need a build system that allows source files to be compiled multiple times, each with different compile targets.

1.4.1.3 Some tricks to build targeted runtime objects.

We need a unique compiled object implementation for each target processor. We still prefer a single file implementation for each function to improve maintenance. So we need a way to separate setting the platform target from the implementation source. Also we need to provide a unique external symbol for each target specific implementation of a function.

This can be handled with a simple macro to append a suffix based on system-specific predefined macro settings.

```
#ifdef _ARCH_PWR9
#define __VEC_PWR_IMP(FNAME) FNAME ## _PWR9
#else
#ifdef _ARCH_PWR8
#define __VEC_PWR_IMP(FNAME) FNAME ## _PWR8
#else
#define __VEC_PWR_IMP(FNAME) FNAME ## _PWR7
#endif
#endif
```

Then use __VEC_PWR_IMP() as function name wrapper in the implementation source file.

```
 //
 //  \file  vec_int512_runtime.c
 //
#include <altivec.h>
#include <pveclib/vec_int128_ppc.h>
#include <pveclib/vec_int512_ppc.h>
// vec_mul128x128_inline is defined in vec_int512_ppc.h
__VEC_U_256
__VEC_PWR_IMP (vec_mul128x128) (vui128_t m1l, vui128_t m2l)
{
  return vec_mul128x128_inline (m1l, m2l);
}
```

Then the use __VEC_PWR_IMP() function wrapper for any calling function that is linked statically to that library function.

```
__VEC_U_1024
__VEC_PWR_IMP (vec_mul512x512) (__VEC_U_512 m1, __VEC_U_512 m2)
{
  __VEC_U_1024 result;
  __VEC_U_512x1 mp3, mp2, mp1, mp0;
  mp0.x640 = __VEC_PWR_IMP(vec_mul512x128) (m1, m2.vx0);
  result.vx0 = mp0.x3.v1x128;
  mp1.x640 = __VEC_PWR_IMP(vec_madd512x128a512) (m1, m2.vx1, mp0.x3.v0x512);
  result.vx1 = mp1.x3.v1x128;
  mp2.x640 = __VEC_PWR_IMP(vec_madd512x128a512) (m1, m2.vx2, mp1.x3.v0x512);
  result.vx2 = mp2.x3.v1x128;
  mp3.x640 = __VEC_PWR_IMP(vec_madd512x128a512) (m1, m2.vx3, mp2.x3.v0x512);
  result.vx3 = mp3.x3.v1x128;
  result.vx4 = mp3.x3.v0x512.vx0;
  result.vx5 = mp3.x3.v0x512.vx1;
  result.vx6 = mp3.x3.v0x512.vx2;
  result.vx7 = mp3.x3.v0x512.vx3;
  return result;
}
```

The **runtime** library implementation is in a separate file from the **inline** implementation. The vec_int512_ppc.h file contains:

- static inline implementations and associated doxygen interface descriptions. These are still small enough to used directly by application codes and as building blocks for larger library implementations.
- extern function declarations and associated doxygen interface descriptions. These names are for the dynamic shared object (**DSO**) function implementations. The functions are not qualified with inline or target suffixes. The expectation is the dynamic linker mechanism with bind to the appropriate implementation.
- extern function declarations qualified with a target suffix. These names are for the statically linked (**archive**) function implementations. The suffix is applied by the __VEC_PWR_IMP() macro for the current (default) target processor. These have no doxygen descriptions as using the __VEC_PWR_IMP() macro interferes with the doxygen scanner. But the interface is the same as the unqualified extern for the DSO implementation of the same name.

The runtime source file (for example vec_int512_runtime.c) contains the common implementations for all the target qualified static interfaces.

- Again the function names are target qualified via the __VEC_PWR_IMP() macro.
- The runtime implementation can use any of the PVECLIB inline operations (see: vec_mul128x128() and vec_mul256x256()) as well as other function implementations from the same file (see: vec_mul512x512() and vec_mul2048x2048()).
- At the -O3 optimization level the compiler will attempt to inline functions referenced from the same file. Compiler heuristics will limit this based on estimates for the final generated object size. GCC also supports the function __ attribute __ ((flatten)) which overrides the in-lining size heuristics.
- These implementations can also use target specific CPP conditional codes to manually tweak code optimization or generated code size for specific targets.

This simple strategy allows the collection of the larger function implementations into a single source file and build object files for multiple platform targets. For example collect all the multiple precision quadword implementations into a source file named **vec_int512_runtime.c**.

1.4.2 Building static runtime libraries

This source file can be compiled multiple times for different platform targets. The resulting object files have unique function symbols due to the platform specific suffix provided by the __VEC_PWR_IMP() macro. There are a number of build strategies for this.

For example, create a small source file named **vec_runtime_PWR8.c** that starts with the target pragma and includes the multi-platform source file.

```
//  \file  vec_runtime_PWR8.c
#pragma GCC target ("cpu=power8")
#include "vec_int512_runtime.c"
```

Similarly for **vec_runtime_PWR7.c**, **vec_runtime_PWR9.c** with appropriate changes for "cpu='. Additional runtime source files can be included as needed. Other multiple precision functions supporting BCD and BCD <-> binary conversions are likely candidates.

Note

> Current Clang compilers silently ignore "#pragme GCC target". This causes all such targeted runtimes to revert to the compiler default target or configure CFLAGS "-mcpu=". In this case the __VEC_PWR_IMP() macro will apply the same suffix to all functions across the targeted runtime builds. As a result linking these targeted runtime objects into the DSO will fail with duplicate symbols.

Projects using autotools (like PVECLIB) can use Makefile.am rules to associate rumtime source files with a library. For example:

```
libpvec_la_SOURCES = vec_runtime_PWR9.c \          vec_runtime_PWR8.c \          vec_runtime_PWR7.c
```

If compiling with GCC this is sufficient for automake to generate Makefiles to compile each of the runtime sources and combine them into a single static archive named libpvec.a. However it is not that simple, especially if the build uses a different compiler.

We would like to use Makefile.am rules to specify different -mcpu= compile options. This eliminates the #pragma GCC target and simplifies the platform source files too something like:

```
 //
 //  \file  vec_runtime_PWR8.c
 //
#include "vec_int512_runtime.c"
```

This requires splitting the target specific runtimes into distinct automake libraries.

```
libpveccommon_la_SOURCES = tipowof10.c decpowof2.c
libpvecPWR9_la_SOURCES = vec_runtime_PWR9.c
libpvecPWR8_la_SOURCES = vec_runtime_PWR8.c
libpvecPWR7_la_SOURCES = vec_runtime_PWR7.c
```

Then add the -mcpu compile option to runtime library CFLAGS

```
libpvecPWR9_la_CFLAGS = -mcpu=power9
libpvecPWR8_la_CFLAGS = -mcpu=power8
libpvecPWR7_la_CFLAGS = -mcpu=power7
```

Then use additional automake rules to combine these targeted runtimes into a single static archive library.

```
libpvecstatic_la_LIBADD = libpveccommon.la
libpvecstatic_la_LIBADD += libpvecPWR9.la
libpvecstatic_la_LIBADD += libpvecPWR8.la
libpvecstatic_la_LIBADD += libpvecPWR7.la
```

However this does not work if the user (build configure) specifies flag variables (i.e. CFLAGS) containing -mcpu= options internal use of target options.

Note

> Automake/libtool will always apply the user CFLAGS after any AM_CFLAGS or yourlib_la_CFLAGS (See: `Automake documentation: Flag Variables Ordering`) and the last -mcpu option always wins. This has the same affect as the compiler ignoring the #pragma GCC target options described above.

1.4.2.1 A deeper look at library Makefiles

This requires a deeper dive into the black arts of automake and libtools. In this case the libtool macro LTCOMPILE expands the various flag variables in a specific order (with $CFLAGS last) for all –tag=CC –mode=compile commands. In this case we need to either:

- locally edit CFLAGS to eliminates any -mcpu= (or -O) options so that our internal build targets are applied.
- provide our own alternative to the LTCOMPILE macro and use our own explicit make rules. (See ./pveclib/src/↩ Makefile.am for examples.)

So lets take a look at LTCOMPILE:

```
LTCOMPILE = $(LIBTOOL) $(AM_V_lt) --tag=CC $(AM_LIBTOOLFLAGS) \
        $(LIBTOOLFLAGS) --mode=compile $(CC) $(DEFS) \
        $(DEFAULT_INCLUDES) $(INCLUDES) $(AM_CPPFLAGS) $(CPPFLAGS) \
        $(AM_CFLAGS) $(CFLAGS)
```

Note

> "$(CFLAGS)" is always applied after all other *FLAGS*.

The generated Makefile.in includes rules that depend on LTCOMPILE. For example the general rule for compile .c source to .lo objects.

```
.c.lo:
@am__fastdepCC_TRUE@    $(AM_V_CC)depbase=`echo $@ | sed 's|[^/]*$$|$(DEPDIR)/&|;s|\.lo$$||'`;\
@am__fastdepCC_TRUE@    $(LTCOMPILE) -MT $@ -MD -MP -MF $$depbase.Tpo -c -o $@ $< &&\
@am__fastdepCC_TRUE@    $(am__mv) $$depbase.Tpo $$depbase.Plo
@AMDEP_TRUE@@am__fastdepCC_FALSE@       $(AM_V_CC)source='$<' object='$@' libtool=yes @AMDEPBACKSLASH@
@AMDEP_TRUE@@am__fastdepCC_FALSE@       DEPDIR=$(DEPDIR) $(CCDEPMODE) $(depcomp) @AMDEPBACKSLASH@
@am__fastdepCC_FALSE@   $(AM_V_CC@am__nodep@)$(LTCOMPILE) -c -o $@ $<
```

Or the more specific rule to compile the vec_runtime_PWR9.c for the -mcpu=power9 target:

```
 libpvecPWR9_la-vec_runtime_PWR9.lo: vec_runtime_PWR9.c
@am__fastdepCC_TRUE@    $(AM_V_CC)$(LIBTOOL) $(AM_V_lt) --tag=CC $(AM_LIBTOOLFLAGS) \
 $(LIBTOOLFLAGS) --mode=compile $(CC) $(DEFS) $(DEFAULT_INCLUDES) $(INCLUDES) \
 $(AM_CPPFLAGS) $(CPPFLAGS) $(libpvecPWR9_la_CFLAGS) $(CFLAGS) \
 -MT libpvecPWR9_la-vec_runtime_PWR9.lo -MD -MP -MF \
 $(DEPDIR)/libpvecPWR9_la-vec_runtime_PWR9.Tpo -c -o libpvecPWR9_la-vec_runtime_PWR9.lo \
 `test -f 'vec_runtime_PWR9.c' || echo '$(srcdir)/'`vec_runtime_PWR9.c
@am__fastdepCC_TRUE@    $(AM_V_at)$(am__mv) $(DEPDIR)/libpvecPWR9_la-vec_runtime_PWR9.Tpo \
 $(DEPDIR)/libpvecPWR9_la-vec_runtime_PWR9.Plo
@AMDEP_TRUE@@am__fastdepCC_FALSE@       $(AM_V_CC)source='vec_runtime_PWR9.c' \
 object='libpvecPWR9_la-vec_runtime_PWR9.lo' libtool=yes @AMDEPBACKSLASH@
@AMDEP_TRUE@@am__fastdepCC_FALSE@       DEPDIR=$(DEPDIR) $(CCDEPMODE) \
 $(depcomp) @AMDEPBACKSLASH@
@am__fastdepCC_FALSE@   $(AM_V_CC@am__nodep@)$(LIBTOOL) $(AM_V_lt) --tag=CC \
 $(AM_LIBTOOLFLAGS) $(LIBTOOLFLAGS) --mode=compile $(CC) $(DEFS) $(DEFAULT_INCLUDES) \
 $(INCLUDES) $(AM_CPPFLAGS) $(CPPFLAGS) $(libpvecPWR9_la_CFLAGS) $(CFLAGS) -c \
 -o libpvecPWR9_la-vec_runtime_PWR9.lo `test -f 'vec_runtime_PWR9.c' \
 || echo '$(srcdir)/'`vec_runtime_PWR9.c
```

Which is eventually generated into the Makefile as:

```
libpvecPWR9_la-vec_runtime_PWR9.lo: vec_runtime_PWR9.c
        $(AM_V_CC)$(LIBTOOL) $(AM_V_lt) --tag=CC $(AM_LIBTOOLFLAGS) $(LIBTOOLFLAGS) \
        --mode=compile $(CC) $(DEFS) $(DEFAULT_INCLUDES) $(INCLUDES) $(AM_CPPFLAGS) \
        $(CPPFLAGS) $(libpvecPWR9_la_CFLAGS) $(CFLAGS) -MT libpvecPWR9_la-vec_runtime_PWR9.lo \
        -MD -MP -MF $(DEPDIR)/libpvecPWR9_la-vec_runtime_PWR9.Tpo -c -o \
        libpvecPWR9_la-vec_runtime_PWR9.lo `test -f 'vec_runtime_PWR9.c' || \
        echo '$(srcdir)/'`vec_runtime_PWR9.c
        $(AM_V_at)$(am__mv) $(DEPDIR)/libpvecPWR9_la-vec_runtime_PWR9.Tpo \
        $(DEPDIR)/libpvecPWR9_la-vec_runtime_PWR9.Plo
#       $(AM_V_CC)source='vec_runtime_PWR9.c' object='libpvecPWR9_la-vec_runtime_PWR9.lo' \
#       libtool=yes DEPDIR=$(DEPDIR) $(CCDEPMODE) $(depcomp) \
#       $(AM_V_CC_no)$(LIBTOOL) $(AM_V_lt) --tag=CC $(AM_LIBTOOLFLAGS) $(LIBTOOLFLAGS) \
#       --mode=compile $(CC) $(DEFS) $(DEFAULT_INCLUDES) $(INCLUDES) $(AM_CPPFLAGS) \
#       $(CPPFLAGS) $(libpvecPWR9_la_CFLAGS) $(CFLAGS) -c -o libpvecPWR9_la-vec_runtime_PWR9.lo \
#       `test -f 'vec_runtime_PWR9.c' || echo '$(srcdir)/'`vec_runtime_PWR9.c
```

Somehow in the internal struggle for the dark soul of automake/libtools, the *@am__fastdepCC_TRUE@* conditional wins out over *@AMDEP_TRUE@@am__fastdepCC_FALSE@* , and the alternate rule was commented out as the Makefile was generated.

However this still leaves a problem. While we see that $(libpvecPWR9_la_CFLAGS) applies the "-mcpu=power9" target option, it is immediately followed by $(CFLAGS). And it CFLAGS contains any "-mcpu=" option the last "-mcpu=" option always wins. The result will a broken library archives with duplicate symbols.

Note

> The techniques described work reliably for most codes and compilers as long as the user does not override target (-mcpu=) with CFLAGS on configure.

1.4.2.2 Adding our own Makefile magic

Todo Is there a way for automake to compile vec_int512_runtime.c with -mcpu=power9 and -o vec_runtime_PWR9.o? And similarly for PWR7/PWR8.

Once we get a glimpse of the underlying automake/libtool rule generation we have a template for how to solve this problem. However while we need to workaround some automake/libtool constraints we also want fit into overall flow.

First we need an alternative to **LTCOMPILE** where we can bypass user provided **CFLAGS**. For example:

```
PVECCOMPILE = $(LIBTOOL) $(AM_V_lt) --tag=CC $(AM_LIBTOOLFLAGS) \
        $(LIBTOOLFLAGS) --mode=compile $(CC) $(DEFS) \
        $(DEFAULT_INCLUDES) $(INCLUDES) $(AM_CPPFLAGS) $(CPPFLAGS) \
        $(AM_CFLAGS)
```

In this variant (**PVECCOMPILE**) we simply leave $(CFLAGS) off the end of the macro.

Now we can use the generated rule above as an example to provide our own Makefile rules. These rules will be passed directly to the generated Makefile. For example:

```
vec_staticrt_PWR9.lo: vec_runtime_PWR9.c $(pveclibinclude_HEADERS)
if am__fastdepCC
        $(PVECCOMPILE) $(PVECLIB_POWER9_CFLAGS) -MT $@ -MD -MP -MF \
        $(DEPDIR)/$*.Tpo -c -o $@ $(srcdir)/vec_runtime_PWR9.c
        mv -f $(DEPDIR)/$*.Tpo $(DEPDIR)/$*.Plo
else
if AMDEP
        source='vec_runtime_PWR9.c' object='$@' libtool=yes @AMDEPBACKSLASH@
        DEPDIR=$(DEPDIR) $(CCDEPMODE) $(depcomp) @AMDEPBACKSLASH@
endif
        $(PVECCOMPILE) $(PVECLIB_POWER9_CFLAGS) -c -o $@ $(srcdir)/vec_runtime_PWR9.c
endif
```

We change the target (vec_staticrt_PWR9.lo) of the rule to indicate that this object is intended for a *static* runtime archive. And we list prerequisites vec_runtime_PWR9.c and $(pveclibinclude_HEADERS)

For the recipe we expand both clauses (am__fastdepCC and AMDEP) from the example. We don't know exactly what they represent or do, but assume they both are needed for some configurations. We use the alternative PVECCOMPILE to provide all the libtool commands and options we need without the CFLAGS. We use new PVECLIB_POWER9_CFLAGS macro to provide all the platform specific target options we need. The automatic variable $@ provides the file name of the target object (vec_staticrt_PWR9.lo). And we specify the $(srcdir) qualified source file (vec_runtime_PWR9.c) as input to the compile. We can provide similar rules for the other processor targets (PWR8/PWR7).

With this technique we control the compilation of specific targets without requiring unique LTLIBRARIES. This was only required before so libtool would allow target specific CFLAGS. So we can eliminate libpvecPWR9.la, libpvecPWR8.la, and libpvecPWR7.la from lib_LTLIBRARIES.

Continuing the theme of separating the static archive elements from DSO elements we rename libpveccommon.la to libpvecstatic.la. We can add the common (none target specific) source files and CFLAGS to *libpvecstatic_la*.

```
libpvecstatic_la_SOURCES = tipowof10.c decpowof2.c
libpvecstatic_la_CFLAGS = $(AM_CPPFLAGS) $(PVECLIB_DEFAULT_CFLAGS) $(AM_CFLAGS)
```

We still need to add the target specific objects generated by the rules above to the libpvecstatic.a archive.

```
# libpvecstatic_la already includes tipowof10.c decpowof2.c.
# Now add the name qualified -mcpu= target runtimes.
libpvecstatic_la_LIBADD = vec_staticrt_PWR9.lo
libpvecstatic_la_LIBADD += vec_staticrt_PWR8.lo
libpvecstatic_la_LIBADD += vec_staticrt_PWR7.lo
```

Note

> the libpvecstatic archive will contain 2 or 3 implementations of each target specific function (i.e. the function vec_mul128x128() will have implementations vec_mul128x128_PWR7() and vec_mul128x128_PWR8(), vec_mul128x128_PWR9()). This OK because because the target suffix insures the name is unique within the archive. When an application calls function with the appropriate target suffix (using the __VEC_PWR_IMP() wrapper macro) and links to libpvecstatic, the linker will extract only the matching implementations and include them in the static program image.

1.4.3 Building dynamic runtime libraries

Building objects for dynamic runtime libraries is a bit more complicated than building static archives. For one dynamic libraries requires position independent code (**PIC**) while static code does not. Second we want to leverage the Dynamic Linker/Loader's GNU Indirect Function (See: `What is an indirect function (IFUNC)?`) binding mechanism.

PIC functions require a more complicated call linkage or function prologue. This usually requires the -fpic compiler option. This is the case for the OpenPOWER ELF V2 ABI. Any PIC function must assume that the caller may be from an different execution unit (library or main executable). So the called function needs to establish the Table of Contents (**TOC**) base address for itself. This is the case if the called function needs to reference static or const storage variables or calls to functions in other dynamic libraries. So it is normal to compile library runtime codes separately for static archives and DSOs.

Note

> The details of how the **TOC** is established differs between the ELF V1 ABI (Big Endian POWER) and the ELF V2 ABI (Little Endian POWER). This should not be an issue if compile options (-fpic) are used correctly.

There are additional differences associated with dynamic selection of function Implementations for different processor targets. The Linux dynamic linker/loader (ld64.so) provides general mechanism for target specific binding of function call linkage.

The dynamic linker employees a user supplied resolver mechanism as function calls are dynamically bound to to an implementation. The DSO exports function symbols that externally look like a normal *extern*. For example:

```
extern __VEC_U_256
vec_mul128x128 (vui128_t, vui128_t);
```

This symbol's implementation has a special **STT_GNU_IFUNC** attribute recognized by the dynamic linker which associates this symbol with the corresponding runtime resolver function. So in addition to any platform specific implementations we need to provide the resolver function referenced by the *IFUNC* symbol. For example:

```
 //
 //  \file  vec_runtime_DYN.c
 //
extern __VEC_U_256
```

```
vec_mul128x128_PWR7 (vui128_t, vui128_t);
extern __VEC_U_256
vec_mul128x128_PWR8 (vui128_t, vui128_t);
extern __VEC_U_256
vec_mul128x128_PWR9 (vui128_t, vui128_t);
static
__VEC_U_256
(*resolve_vec_mul128x128 (void))(vui128_t, vui128_t)
{
#ifdef  __BUILTIN_CPU_SUPPORTS__
  if (__builtin_cpu_is ("power9"))
    return vec_mul128x128_PWR9;
  else
    {
      if (__builtin_cpu_is ("power8"))
        return vec_mul128x128_PWR8;
      else
        return vec_mul128x128_PWR7;
    }
#else // ! __BUILTIN_CPU_SUPPORTS__
    return vec_mul128x128_PWR7;
#endif
}
__VEC_U_256
vec_mul128x128 (vui128_t, vui128_t)
__attribute__ ((ifunc ("resolve_vec_mul128x128")));
```

For convince we collect the:

- IFUNC symbols
- corresponging resolver functions
- and externs to target specific implementations

into one or more source files (For example: vec_runtime_DYN.c).

On the program's first call to an *IFUNC* symbol, the dynamic linker calls the resolver function associated with that symbol. The resolver function performs a runtime check to determine the platform, selects the (closest) matching platform specific function, then returns that function pointer to the dynamic linker.

The dynamic linker stores this function pointer in the callers Procedure Linkage Tables (PLT) before forwarding the call to the resolved implementation. Any subsequent calls to this function symbol branch (via the PLT) directly to the appropriate platform specific implementation.

Note

> The platform specific implementations we use here are compiled from the same source files we used to build the static library archive.

Like the static libraries we need to build multiple target specific implementations of the functions. So we can leverage the example of explicit Makefile rules we used for the static archive but with some minor differences. For example:

```
vec_dynrt_PWR9.lo: vec_runtime_PWR9.c $(pveclibinclude_HEADERS)
if am__fastdepCC
        $(PVECCOMPILE) -fpic $(PVECLIB_POWER9_CFLAGS) -MT $@ -MD -MP -MF \
        $(DEPDIR)/$*.Tpo -c -o $@ $(srcdir)/vec_runtime_PWR9.c
        mv -f $(DEPDIR)/$*.Tpo $(DEPDIR)/$*.Plo
else
if AMDEP
        source='vec_runtime_PWR9.c' object='$@' libtool=yes @AMDEPBACKSLASH@
        DEPDIR=$(DEPDIR) $(CCDEPMODE) $(depcomp) @AMDEPBACKSLASH@
endif
        $(PVECCOMPILE) -fpic $(PVECLIB_POWER9_CFLAGS) -c -o $@ \
        $(srcdir)/vec_runtime_PWR9.c
endif
```

Again we change the rule target (vec_dynrt_PWR9.lo) of the rule to indicate that this object is intended for a *DSO* runtime. And we list the same prerequisites vec_runtime_PWR9.c and $(pveclibinclude_HEADERS)

For the recipe we expand both clauses (am__fastdepCC and AMDEP) from the example. We use the alternative PVECCOMPILE to provide all the libtool commands and options we need without the CFLAGS. But we insert the -fpic option so the compiler will will generate position independent code. We use a new PVECLIB_POWER9_CFLAGS macro to provide all the platform specific target options we need. The automatic variable $@ provides the file name of the target object (vec_dynrt_PWR9.lo). And we specify the same $(srcdir) qualified source file (vec_runtime_PWR9.c) we used for the static library. We can provide similar rules for the other processor targets (PWR8/PWR7). We also build an -fpic version of vec_runtime_common.c.

Continuing the theme of separating the static archive elements from DSO elements, we use libpvec.la as the libtool name for libpvec.so. Here we add the source files for the IFUNC resolvers and add -fpic as library specific CFLAGS to *libpvec_la*.

```
libpvec_la_SOURCES = vec_runtime_DYN.c
libpvec_la_CFLAGS = $(AM_CPPFLAGS) -fpic $(PVECLIB_DEFAULT_CFLAGS) $(AM_CFLAGS)
```

We still need to add the target specific and common objects generated by the rules above to the libpvec library.

```
# libpvec_la already includes vec_runtime_DYN.c compiled compiled -fpic
# for IFUNC resolvers.
# Now adding the -fpic -mcpu= target built runtimes.
libpvec_la_LDFLAGS = -version-info $(PVECLIB_SO_VERSION)
libpvec_la_LIBADD = vec_dynrt_PWR9.lo
libpvec_la_LIBADD += vec_dynrt_PWR8.lo
libpvec_la_LIBADD += vec_dynrt_PWR7.lo
libpvec_la_LIBADD += vec_dynrt_common.lo
libpvec_la_LIBADD += -lc
```

1.4.4 Calling Multi-platform functions

The next step is to provide mechanisms for applications to call these functions via static or dynamic linkage. For static linkage the application needs to reference a specific platform variant of the functions name. For dynamic linkage we will use **STT_GNU_IFUNC** symbol resolution (a symbol type extension to the ELF standard).

1.4.4.1 Static linkage to platform specific functions

For static linkage the application is compiled for a specific platform target (via -mcpu=). So function calls should be bound to the matching platform specific implementations. The application may select the platform specific function directly by defining a *extern* and invoking the platform qualified function.

Or simply use the __VEC_PWR_IMP() macro as wrapper for the function name in the application. This selects the appropriate platform specific implementation based on the -mcpu= specified for the application compile. For example.

```
k = __VEC_PWR_IMP (vec_mul128x128)(i, j);
```

The vec_int512_ppc.h header provides the default platform qualified *extern* declarations for this and related functions based on the -mcpu= specified for the compile of application including this header. For example.

```
extern __VEC_U_256
__VEC_PWR_IMP (vec_mul128x128) (vui128_t, vui128_t);
```

For example if the applications calling vec_mul128x128() is itself compiled with -mcpu=power8, then the __VEC_PWR_IMP() will insure that:

- The vec_int512_ppc.h header will define an extern for vec_mul128x128_PWR8.
- That application's calls to __VEC_PWR_IMP (vec_mul128x128) will reference vec_mul128x128_PWR8.

The application should then link to the libpvecstatic.a archive. Where the application references PVECLIB functions with the appropriate target suffix, the linker will extract only the matching implementations and include them in the program image.

1.4.4.2 Dynamic linkage to platform specific functions

Applications using dynamic linkage will call the unqualified function symbol. For example:

```
extern __VEC_U_256
vec_mul128x128 (vui128_t, vui128_t);
```

This symbol's implementation (in libpvec.so) has a special **STT_GNU_IFUNC** attribute recognized by the dynamic linker which associates this symbol with the corresponding runtime resolver function. The application simply calls the (unqualified) function and the dynamic linker (with the help of PVECLIB's IFUNC resolvers) handles the details.

1.5 Performance data.

It is useful to provide basic performance data for each pveclib function. This is challenging as these functions are small and intended to be in-lined within larger functions (algorithms). As such they are subject to both the compiler's instruction scheduling and common subexpression optimizations plus the processors super-scalar and out-of-order execution design features.

As pveclib functions are normally only a few instructions, the actual timing will depend on the context it is in (the instructions that it depends on for data and instructions that proceed them in the pipelines).

The simplest approach is to use the same performance metrics as the Power Processor Users Manuals Performance Profile. This is normally per instruction latency in cycles and throughput in instructions issued per cycle. There may also be additional information for special conditions that may apply.

For example the vector float absolute value function. For recent PowerISA implementations this a single (VSX **xvabssp**) instruction which we can look up in the POWER8 / POWER9 Processor User's Manuals (**UM**).

processor	Latency	Throughput
power8	6-7	2/cycle
power9	2	2/cycle

The POWER8 UM specifies a latency of *"6 cycles to FPU (+1 cycle to other VSU ops"* for this class of VSX single precision FPU instructions. So the minimum latency is 6 cycles if the register result is input to another VSX single precision FPU instruction. Otherwise if the result is input to a VSU logical or integer instruction then the latency is 7 cycles. The POWER9 UM shows the pipeline improvement of 2 cycles latency for simple FPU instructions like this. Both processors support dual pipelines for a 2/cycle throughput capability.

A more complicated example:

```
static inline vb32_t
vec_isnanf32 (vf32_t vf32)
{
vui32_t tmp2;
const vui32_t expmask = CONST_VINT128_W(0x7f800000, 0x7f800000, 0x7f800000,
                                        0x7f800000);
#if _ARCH_PWR9
// P9 has a 2 cycle xvabssp and eliminates a const load.
tmp2 = (vui32_t) vec_abs (vf32);
#else
const vui32_t signmask = CONST_VINT128_W(0x80000000, 0x80000000, 0x80000000,
                                         0x80000000);
tmp2 = vec_andc ((vui32_t)vf32, signmask);
#endif
return vec_cmpgt (tmp2, expmask);
}
```

Here we want to test for *Not A Number* without triggering any of the associate floating-point exceptions (VXSNAN or VXVC). For this test the sign bit does not effect the result so we need to zero the sign bit before the actual test. The vector abs would work for this, but we know from the example above that this instruction has a high latency as we are definitely passing the result to a non-FPU instruction (vector compare greater than unsigned word).

So the code needs to load two constant vectors masks, then vector and-compliment to clear the sign-bit, before comparing each word for greater then infinity. The generated code should look something like this:

```
addis   r9,r2,.rodata.cst16+0x10@ha
addis   r10,r2,.rodata.cst16+0x20@ha
addi    r9,r9,.rodata.cst16+0x10@l
addi    r10,r10,.rodata.cst16+0x20@l
lvx     v0,0,r10  # load vector const signmask
lvx     v12,0,r9  # load vector const expmask
xxlandc vs34,vs34,vs32
vcmpgtuw v2,v2,v12
```

So six instructions to load the const masks and two instructions for the actual vec_isnanf32 function. The first six instructions are only needed once for each containing function, can be hoisted out of loops and into the function prologue, can be *commoned* with the same constant for other pveclib functions, or executed out-of-order and early by the processor.

Most of the time, constant setup does not contribute measurably to the over all performance of vec_isnanf32. When it does it is limited by the longest (in cycles latency) of the various independent paths that load constants. In this case the const load sequence is composed of three pairs of instructions that can issue and execute in parallel. The addis/addi FXU instructions supports throughput of 6/cycle and the lvx load supports 2/cycle. So the two vector constant load sequences can execute in parallel and the latency is same as a single const load.

For POWER8 it appears to be (2+2+5=) 9 cycles latency for the const load. While the core vec_isnanf32 function (xxlandc/vcmpgtuw) is a dependent sequence and runs (2+2) 4 cycles latency. Similar analysis for POWER9 where the addis/addi/lvx sequence is still listed as (2+2+5) 9 cycles latency. While the xxlandc/vcmpgtuw sequence increases to (2+3) 5 cycles.

The next interesting question is what can we say about throughput (if anything) for this example. The thought experiment is "what would happen if?";

- two or more instances of vec_isnanf32 are used within a single function,
- in close proximity in the code,
- with independent data as input,

could the generated instructions execute in parallel and to what extent. This illustrated by the following (contrived) example:

```
int
test512_all_f32_nan (vf32_t val0, vf32_t val1, vf32_t val2, vf32_t val3)
{
const vb32_t alltrue = { -1, -1, -1, -1 };
vb32_t nan0, nan1, nan2, nan3;
nan0 = vec_isnanf32 (val0);
nan1 = vec_isnanf32 (val1);
nan2 = vec_isnanf32 (val2);
nan3 = vec_isnanf32 (val3);
nan0 = vec_and (nan0, nan1);
nan2 = vec_and (nan2, nan3);
nan0 = vec_and (nan2, nan0);
return vec_all_eq(nan0, alltrue);
}
```

which tests 4 X vector float (16 X float) values and returns true if all 16 floats are NaN. Recent compilers will generates something like the following PowerISA code:

```
   addis   r9,r2,-2
   addis   r10,r2,-2
   vspltisw v13,-1      # load vector const alltrue
   addi    r9,r9,21184
   addi    r10,r10,-13760
   lvx     v0,0,r9      # load vector const signmask
   lvx     v1,0,r10     # load vector const expmask
   xxlandc vs35,vs35,vs32
   xxlandc vs34,vs34,vs32
   xxlandc vs37,vs37,vs32
   xxlandc vs36,vs36,vs32
   vcmpgtuw v3,v3,v1    # nan1 = vec_isnanf32 (val1);
   vcmpgtuw v2,v2,v1    # nan0 = vec_isnanf32 (val0);
   vcmpgtuw v5,v5,v1    # nan3 = vec_isnanf32 (val3);
   vcmpgtuw v4,v4,v1    # nan2 = vec_isnanf32 (val2);
   xxland  vs35,vs35,vs34       # nan0 = vec_and (nan0, nan1);
   xxland  vs36,vs37,vs36       # nan2 = vec_and (nan2, nan3);
   xxland  vs36,vs35,vs36       # nan0 = vec_and (nan2, nan0);
   vcmpequw. v4,v4,v13  # vec_all_eq(nan0, alltrue);
...
```

first the generated code loading the vector constants for signmask, expmask, and alltrue. We see that the code is generated only once for each constant. Then the compiler generate the core vec_isnanf32 function four times and interleaves the instructions. This enables parallel pipeline execution where conditions allow. Finally the 16X isnan results are reduced to 8X, then 4X, then to a single condition code.

For this exercise we will ignore the constant load as in any realistic usage it will be *commoned* across several pveclib functions and hoisted out of any loops. The reduction code is not part of the vec_isnanf32 implementation and also ignored. The sequence of 4X xxlandc and 4X vcmpgtuw in the middle it the interesting part.

For POWER8 both xxlandc and vcmpgtuw are listed as 2 cycles latency and throughput of 2 per cycle. So we can assume that (only) the first two xxlandc will issue in the same cycle (assuming the input vectors are ready). The issue of the next two xxlandc instructions will be delay by 1 cycle. The following vcmpgtuw instruction are dependent on the xxlandc results and will not execute until their input vectors are ready. The first two vcmpgtuw instruction will execute 2 cycles (latency) after the first two xxlandc instructions execute. Execution of the second two vcmpgtuw instructions will be delayed 1 cycle due to the issue delay in the second pair of xxlandc instructions.

So at least for this example and this set of simplifying assumptions we suggest that the throughput metric for vec_↩ isnanf32 is 2/cycle. For latency metric we offer the range with the latency for the core function (without and constant load overhead) first. Followed by the total latency (the sum of the constant load and core function latency). For the vec_isnanf32 example the metrics are:

processor	Latency	Throughput
power8	4-13	2/cycle
power9	5-14	2/cycle

Looking at a slightly more complicated example where core functions implementation can execute more then one instruction per cycle. Consider:

```
static inline vb32_t
vec_isnormalf32 (vf32_t vf32)
{
vui32_t tmp, tmp2;
const vui32_t expmask = CONST_VINT128_W(0x7f800000, 0x7f800000, 0x7f800000,
                                        0x7f800000);
const vui32_t minnorm = CONST_VINT128_W(0x00800000, 0x00800000, 0x00800000,
                                        0x00800000);
#if _ARCH_PWR9
// P9 has a 2 cycle xvabssp and eliminates a const load.
tmp2 = (vui32_t) vec_abs (vf32);
#else
const vui32_t signmask = CONST_VINT128_W(0x80000000, 0x80000000, 0x80000000,
                                         0x80000000);
tmp2 = vec_andc ((vui32_t)vf32, signmask);
#endif
tmp = vec_and ((vui32_t) vf32, expmask);
tmp2 = (vui32_t) vec_cmplt (tmp2, minnorm);
tmp = (vui32_t) vec_cmpeq (tmp, expmask);
return (vb32_t )vec_nor (tmp, tmp2);
}
```

which requires two (independent) masking operations (sign and exponent), two (independent) compares that are dependent on the masking operations, and a final *not OR* operation dependent on the compare results.

The generated POWER8 code looks like this:

```
addis   r10,r2,-2
addis   r8,r2,-2
addi    r10,r10,21184
addi    r8,r8,-13760
addis   r9,r2,-2
lvx     v13,0,r8
addi    r9,r9,21200
lvx     v1,0,r10
lvx     v0,0,r9
xxland  vs33,vs33,vs34
xxlandc vs34,vs45,vs34
vcmpgtuw v0,v0,v1
vcmpequw v2,v2,v13
xxlnor  vs34,vs32,vs34
```

Note this this sequence needs to load 3 vector constants. In previous examples we have noted that POWER8 lvx supports 2/cycle throughput. But with good scheduling, the 3rd vector constant load, will only add 1 additional cycle to the timing (10 cycles).

Once the constant masks are loaded the xxland/xxlandc instructions can execute in parallel. The vcmpgtuw/vcmpequw can also execute in parallel but are delayed waiting for the results of masking operations. Finally the xxnor is dependent on the data from both compare instructions.

For POWER8 the latencies are 2 cycles each, and assuming parallel execution of xxland/xxlandc and vcmpgtuw/vcmpequw we can assume (2+2+2=) 6 cycles minimum latency and another 10 cycles for the constant loads (if needed).

While the POWER8 core has ample resources (10 issue ports across 16 execution units), this specific sequence is restricted to the two *issue ports and VMX execution units* for this class of (simple vector integer and logical) instructions. For vec_isnormalf32 this allows for a lower latency (6 cycles vs the expected 10, over 5 instructions), it also implies that both of the POWER8 cores *VMX execution units* are busy for 2 out of the 6 cycles.

So while the individual instructions have can have a throughput of 2/cycle, vec_isnormalf32 can not. It is plausible for two executions of vec_isnormalf32 to interleave with a delay of 1 cycle for the second sequence. To keep the table information simple for now, just say the throughput of vec_isnormalf32 is 1/cycle.

After that it gets complicated. For example after the first two instances of vec_isnormalf32 are issued, both *VMX execution units* are busy for 4 cycles. So either the first instructions of the third vec_isnormalf32 will be delayed until the fifth cycle. Or the compiler scheduler will interleave instructions across the instances of vec_isnormalf32 and the latencies of individual vec_isnormalf32 results will increase. This is too complicated to put in a simple table.

For POWER9 the sequence is slightly different

```
addis   r10,r2,-2
addis   r9,r2,-2
xvabssp vs45,vs34
addi    r10,r10,-14016
addi    r9,r9,-13920
lvx     v1,0,r10
lvx     v0,0,r9
xxland  vs34,vs34,vs33
vcmpgtuw v0,v0,v13
vcmpequw v2,v2,v1
xxlnor  vs34,vs32,vs34
```

We use vec_abs (xvabssp) to replace the sigmask and vec_andc and so only need to load two vector constants. So the constant load overhead is reduced to 9 cycles. However the the vector compares are now 3 cycles for (2+3+2=) 7 cycles for the core sequence. The final table for vec_isnormalf32:

processor	Latency	Throughput
power8	6-16	1/cycle
power9	7-16	1/cycle

1.5.1 Additional analysis and tools.

The overview above is simplified analysis based on the instruction latency and throughput numbers published in the Processor User's Manuals (see Reference Documentation). These values are *best case* (input data is ready, SMT1 mode, no cache misses, mispredicted branches, or other hazards) for each instruction in isolation.

Note

> This information is intended as a guide for compiler and application developers wishing to optimize for the platform. Any performance tables provided for pveclib functions are in this spirit.

Of course the actual performance is complicated by the overall environment and how the pveclib functions are used. It would be unusual for pveclib functions to be used in isolation. The compiler will in-line pveclib functions and look for sub-expressions it can hoist out of loops or share across pveclib function instances. The The compiler will also model the processor and schedule instructions across the larger containing function. So in actual use the instruction sequences for the examples above are likely to be interleaved with instructions from other pvevlib functions and user written code.

Larger functions that use pveclib and even some of the more complicated pveclib functions (like vec_muludq) defy simple analysis. For these cases it is better to use POWER specific analysis tools. To understand the overall pipeline flows and identify hazards the instruction trace driven performance simulator is recommended.

The `IBM Advance Toolchain` includes an updated (POWER enabled) Valgrind tool and instruction trace plug-in (itrace). The itrace tool (–tool=itrace) collects instruction traces for the whole program or specific functions (via –fnname= option).

Note

> The Valgrind package provided by the Linux Distro may not be enabled for the latest POWER processor. Nor will it include the itrace plug-in or the associated vgi2qt conversion tool.

Instruction trace files are processed by the `Performance Simulator` (sim_ppc) models. Performance simulators are specific to each processor generation (POWER7-9) and provides a cycle accurate modeling for instruction trace streams. The results of the model (a pipe file) can viewed via one the interactive display tools (scrollpv, jviewer) or passed to an analysis tool like `pipestat`.

Chapter 2

Todo List

page POWER Vector Library (pveclib)

Is there a way for automake to compile vec_int512_runtime.c with -mcpu=power9 and -o vec_runtime_PWR9.o? And similarly for PWR7/PWR8.

File vec_bcd_ppc.h

The BCD add/subtract extend/carry story is not complete. The carry extend operations based only on the **OV** condition codes only works as expected for bcdadd operands with the same sign and bcdsub with different signs. See vec_bcdaddcsq() and vec_bcdaddecsq(). Extended BCD difference (or subtract the same sign or add with different signs) is more complicated. See vec_bcdsubcsq() and vec_bcdsubecsq(). Generating a true borrow seems to require looking one (31-digit) column ahead or behind. The first attempt at generating correct borrowing is implemented in vec_cbcdaddcsq() and vec_cbcdaddecsq(). There are still cases where these operation will generate a borrow and invert (10s complement) incorrectly. The net seems to be that for BCD multiple precision difference to work correctly, the larger magnitude must be the first operand.

File vec_int128_ppc.h

The implementation above gives correct results for all the cases tested for divide by constants 10^{31} and 10^{32}). This is not a mathematical proof of correctness, just an observation. Anyone who finds a counter example or offers a mathematical proof should submit a bug report.

File vec_int512_ppc.h

Currently the dynamic resolvers and *IFUNC* symbols for vec_int512_runtime.c are contained within vec_runtime↩_DYN.c. As the list of runtime operations expands to other element sizes/types, vec_runtime_DYN.c should be refactored into multiple files.

Chapter 3

Deprecated List

Member vec_slq4 (vui128_t vra)

Vector Shift Left 4-bits Quadword. Replaced by vec_slqi with shb param = 4.

Member vec_slq5 (vui128_t vra)

Vector Shift Left 5-bits Quadword. Replaced by vec_slqi with shb param = 5.

Member vec_spltd (vui64_t vra, const int ctl)

Vector splat doubleword. Duplicate the selected doubleword element across the doubleword elements of the result.

Member vec_srq4 (vui128_t vra)

Vector Shift right 4-bits Quadword. Replaced by vec_srqi with shb param = 4.

Member vec_srq5 (vui128_t vra)

Vector Shift right 5-bits Quadword. Replaced by vec_srqi with shb param = 5.

Chapter 4

Class Index

4.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 5

File Index

5.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 6

Class Documentation

6.1 __VEC_U_1024 Struct Reference

A vector representation of a 1024-bit unsigned integer.

```
#include <vec_int512_ppc.h>
```

6.1.1 Detailed Description

A vector representation of a 1024-bit unsigned integer.

A homogeneous aggregate of 8 x 128-bit unsigned integer fields. The low order field is named vx0, progressing to the high order field vx7.

The documentation for this struct was generated from the following file:

- src/pveclib/vec_int512_ppc.h

6.2 __VEC_U_1024x512 Union Reference

A vector representation of a 1024-bit unsigned integer as two 512-bit fields.

```
#include <vec_int512_ppc.h>
```

6.2.1 Detailed Description

A vector representation of a 1024-bit unsigned integer as two 512-bit fields.

A union of:

- homogeneous aggregate of 1024-bit unsigned integer.
- struct of:
  - two x homogeneous aggregate of 512-bit unsigned integers.

Note

Useful for summing partial products based on a 512x512-bit multiply.

The documentation for this union was generated from the following file:

- src/pveclib/vec_int512_ppc.h

6.3 __VEC_U_1152 Struct Reference

A vector representation of a 1152-bit unsigned integer.

```
#include <vec_int512_ppc.h>
```

6.3.1 Detailed Description

A vector representation of a 1152-bit unsigned integer.

A homogeneous aggregate of 9 x 128-bit unsigned integer fields. The low order field is named vx0, progressing to the high order field vx8.

Note

Useful for returning the result of a 1024x128-bit multiply.

This structure does not qualify for parameter passing in registers (more than 8 registers are required) and will be passed in memory.

The documentation for this struct was generated from the following file:

- src/pveclib/vec_int512_ppc.h

6.4 __VEC_U_128 Union Reference

Union used to transfer 128-bit data between vector and non-vector types.

```
#include <vec_common_ppc.h>
```

Public Attributes

- signed __int128 i128

  Signed 128-bit integer from pair of 64-bit GPRs.

- unsigned __int128 ui128

  Unsigned 128-bit integer from pair of 64-bit GPRs.

- _Decimal128 dpd128

  128 bit Decimal Float from pair of double float registers.

- long double ldbl128

  IBM long double float from pair of double float registers.

- vui8_t vx16

  128 bit Vector of 16 unsigned char elements.

- vui16_t vx8

  128 bit Vector of 8 unsigned short int elements.

- vui32_t vx4

  128 bit Vector of 4 unsigned int elements.

- vui64_t vx2

  128 bit Vector of 2 unsigned long int (64-bit) elements.

- vui128_t vx1

  128 bit Vector of 1 unsigned __int128 element.

- vf64_t vf2

  128 bit Vector of 2 double float elements.

- struct {
    uint64_t **lower**
    uint64_t **upper**
  } ulong

  Struct of two unsigned long int (64-bit GPR) fields.

6.4.1 Detailed Description

Union used to transfer 128-bit data between vector and non-vector types.

The documentation for this union was generated from the following file:

- src/pveclib/vec_common_ppc.h

6.5 __VEC_U_2048 Struct Reference

A vector representation of a 2048-bit unsigned integer.

```
#include <vec_int512_ppc.h>
```

6.5.1 Detailed Description

A vector representation of a 2048-bit unsigned integer.

A homogeneous aggregate of 16 x 128-bit unsigned integer fields. The low order field is named vx0, progressing to the high order field vx15.

Note

> This structure does not qualify for parameter passing in registers (more than 8 registers are required) and will be passed in memory.

The documentation for this struct was generated from the following file:

- src/pveclib/vec_int512_ppc.h

6.6 __VEC_U_2048x512 Union Reference

A vector representation of a 2048-bit unsigned integer as 4 x 512-bit integer fields.

```
#include <vec_int512_ppc.h>
```

6.6.1 Detailed Description

A vector representation of a 2048-bit unsigned integer as 4 x 512-bit integer fields.

A union of:

- homogeneous aggregate of 2048-bit unsigned integer.
- struct of:
  - 4 x homogeneous aggregate of 512-bit unsigned integers.

Note

> Useful to access 512-bit blocks to pass to a 512x512-bit multiplies. These can be used as partial products in a larger 2048x2048-bit multiply.
>
> This structure does not qualify for parameter passing in registers (more than 8 registers are required) and will be passed in memory.

The documentation for this union was generated from the following file:

- src/pveclib/vec_int512_ppc.h

6.7 __VEC_U_2176 Struct Reference

A vector representation of a 2176-bit unsigned integer.

```
#include <vec_int512_ppc.h>
```

6.7.1 Detailed Description

A vector representation of a 2176-bit unsigned integer.

A homogeneous aggregate of 17 x 128-bit unsigned integer fields. The low order field is named vx0, progressing to the high order field vx16.

Note

> Useful for returning the result of a 2048x128-bit multiply.
>
> This structure does not qualify for parameter passing in registers (more than 8 registers are required) and will be passed in memory.

The documentation for this struct was generated from the following file:

- src/pveclib/vec_int512_ppc.h

6.8 __VEC_U_256 Struct Reference

A vector representation of a 256-bit unsigned integer.

```
#include <vec_int512_ppc.h>
```

6.8.1 Detailed Description

A vector representation of a 256-bit unsigned integer.

A homogeneous aggregate of 2 x 128-bit unsigned integer fields. The low order field is named vx0, progressing to the high order field vx1.

The documentation for this struct was generated from the following file:

- src/pveclib/vec_int512_ppc.h

6.9 __VEC_U_4096 Struct Reference

A vector representation of a 4096-bit unsigned integer.

```
#include <vec_int512_ppc.h>
```

6.9.1 Detailed Description

A vector representation of a 4096-bit unsigned integer.

A homogeneous aggregate of 32 x 128-bit unsigned integer fields. The low order field is named vx0, progressing to the high order field vx31.

Note

> This structure does not qualify for parameter passing in registers (more than 8 registers are required) and will be passed in memory.

The documentation for this struct was generated from the following file:

- src/pveclib/vec_int512_ppc.h

6.10 __VEC_U_4096x512 Union Reference

A vector representation of a 4096-bit unsigned integer as 8 x 512-bit integer fields.

```
#include <vec_int512_ppc.h>
```

6.10.1 Detailed Description

A vector representation of a 4096-bit unsigned integer as 8 x 512-bit integer fields.

A union of:

- homogeneous aggregate of 4096-bit unsigned integer.
- struct of:
    - 8 x homogeneous aggregate of 512-bit unsigned integers.

Note

> Useful to access 512-bit blocks to pass to a 512x512-bit multiplies. These can be used as partial products in a larger 2048x2048-bit multiply.
>
> This structure does not qualify for parameter passing in registers (more than 8 registers are required) and will be passed in memory.

The documentation for this union was generated from the following file:

- src/pveclib/vec_int512_ppc.h

6.11 __VEC_U_512 Struct Reference

A vector representation of a 512-bit unsigned integer.

```
#include <vec_int512_ppc.h>
```

6.11.1 Detailed Description

A vector representation of a 512-bit unsigned integer.

A homogeneous aggregate of 4 x 128-bit unsigned integer fields. The low order field is named vx0, progressing to the high order field vx3.

The documentation for this struct was generated from the following file:

- src/pveclib/vec_int512_ppc.h

6.12 __VEC_U_512x1 Union Reference

A vector representation of a 512-bit unsigned integer and a 128-bit carry-out.

```
#include <vec_int512_ppc.h>
```

6.12.1 Detailed Description

A vector representation of a 512-bit unsigned integer and a 128-bit carry-out.

A union of:

- homogeneous aggregate of 640-bit unsigned integer.
- struct of:
  - homogeneous aggregate of 512-bit unsigned integer.
  - vector representation of the carry out of 512-bit add.
- The Carry out vector overlays the high-order 128-bits of the 640-bit vector.

Note

Useful for passing the carry-out of a 512-bit add into the carry-in of an extended add.

The documentation for this union was generated from the following file:

- src/pveclib/vec_int512_ppc.h

6.13 __VEC_U_640 Struct Reference

A vector representation of a 640-bit unsigned integer.

```
#include <vec_int512_ppc.h>
```

6.13.1 Detailed Description

A vector representation of a 640-bit unsigned integer.

A homogeneous aggregate of 5 x 128-bit unsigned integer fields. The low order field is named vx0, progressing to the high order field vx4.

Note

Useful for returning the result of a 512x128-bit multiply.

The documentation for this struct was generated from the following file:

- src/pveclib/vec_int512_ppc.h

6.14 __VF_128 Union Reference

Union used to transfer 128-bit data between vector and __float128 types.

```
#include <vec_f128_ppc.h>
```

Public Attributes

- vui8_t vx16

  union field of vector unsigned char elements.
- vui16_t vx8

  union field of vector unsigned short elements.
- vui32_t vx4

  union field of vector unsigned int elements.
- vui64_t vx2

  union field of vector unsigned long long elements.
- vui128_t vx1

  union field of vector unsigned __int128 elements.
- vb128_t vbool1

  union field of vector __bool __int128 elements.
- __binary128 vf1

  union field of __float128 elements.
- unsigned __int128 ix1

  union field of __int128 elements.

6.14.1 Detailed Description

Union used to transfer 128-bit data between vector and __float128 types.

The documentation for this union was generated from the following file:

- src/pveclib/vec_f128_ppc.h

Chapter 7

File Documentation

7.1 src/pveclib/vec_bcd_ppc.h File Reference

Header package containing a collection of Binary Coded Decimal (**BCD**) computation and Zoned Character conversion operations on vector registers.

```
#include <pveclib/vec_common_ppc.h>
#include <pveclib/vec_char_ppc.h>
#include <pveclib/vec_int128_ppc.h>
```

Macros

- #define vBCD_t vui32_t

  vector signed BCD integer of up to 31 decimal digits.
- #define vbBCD_t vb32_t

  vector vector bool from 128-bit signed BCD integer.
- #define _BCD_CONST_PLUS_NINES ((vBCD_t) CONST_VINT128_DW128(0x9999999999999999, 0x999999999999999c))

  vector signed BCD constant +9s.
- #define _BCD_CONST_PLUS_ONE ((vBCD_t) CONST_VINT128_DW128(0, 0x1c))

  vector signed BCD constant +1.
- #define _BCD_CONST_MINUS_ONE ((vBCD_t) CONST_VINT128_DW128(0, 0x1d))

  vector signed BCD constant -1.
- #define _BCD_CONST_ZERO ((vBCD_t) CONST_VINT128_DW128(0, 0x0c))

  vector signed BCD constant +0.
- #define _BCD_CONST_SIGN_MASK ((vBCD_t) CONST_VINT128_DW128(0, 0xf))

  vector BCD sign mask in bits 124:127.

Functions

- static vui64_t vec_BCD2BIN (vBCD_t val)

  Convert vector of 2 x unsigned 16-digit BCD values to vector 2 x doubleword binary values.
- static _Decimal128 vec_BCD2DFP (vBCD_t val)

  Convert a Vector Signed BCD value to __Decimal128.
- static vBCD_t vec_BIN2BCD (vui64_t val)

  Convert vector unsigned doubleword binary values to Vector unsigned 16-digit BCD values.
- static vBCD_t vec_DFP2BCD (_Decimal128 val)

  Convert a __Decimal128 value to Vector BCD.
- static vBCD_t vec_bcdadd (vBCD_t a, vBCD_t b)

  Decimal Add Signed Modulo Quadword.
- static vBCD_t vec_bcdaddcsq (vBCD_t a, vBCD_t b)

  Decimal Add & write Carry Signed Quadword.
- static vBCD_t vec_bcdaddecsq (vBCD_t a, vBCD_t b, vBCD_t c)

  Decimal Add Extended & write Carry Signed Quadword.
- static vBCD_t vec_bcdaddesqm (vBCD_t a, vBCD_t b, vBCD_t c)

  Decimal Add Extended Signed Modulo Quadword.
- static vBCD_t vec_bcdcfsq (vi128_t vrb)

  Vector Decimal Convert From Signed Quadword returning up to 31 BCD digits.
- static vBCD_t vec_bcdcfud (vui64_t vrb)

  Vector Decimal Convert From Unsigned doubleword returning up to 2x16 BCD digits.
- static vBCD_t vec_bcdcfuq (vui128_t vra)

  Vector Decimal Convert From Unsigned Quadword returning up to 32 BCD digits.
- static vBCD_t vec_bcdcfz (vui8_t vrb)

  Vector Decimal Convert From Zoned.
- static vbBCD_t vec_bcdcmp_eqsq (vBCD_t vra, vBCD_t vrb)

  Vector Compare Signed BCD Quadword for equal.
- static vbBCD_t vec_bcdcmp_gesq (vBCD_t vra, vBCD_t vrb)

  Vector Compare Signed BCD Quadword for greater than or equal.
- static vbBCD_t vec_bcdcmp_gtsq (vBCD_t vra, vBCD_t vrb)

  Vector Compare Signed BCD Quadword for greater than.
- static vbBCD_t vec_bcdcmp_lesq (vBCD_t vra, vBCD_t vrb)

  Vector Compare Signed BCD Quadword for less than or equal.
- static vbBCD_t vec_bcdcmp_ltsq (vBCD_t vra, vBCD_t vrb)

  Vector Compare Signed BCD Quadword for less than.
- static vbBCD_t vec_bcdcmp_nesq (vBCD_t vra, vBCD_t vrb)

  Vector Compare Signed BCD Quadword for not equal.
- static int vec_bcdcmpeq (vBCD_t vra, vBCD_t vrb)

  Vector Compare Signed BCD Quadword for equal.
- static int vec_bcdcmpge (vBCD_t vra, vBCD_t vrb)

  Vector Compare Signed BCD Quadword for greater than or equal.
- static int vec_bcdcmpgt (vBCD_t vra, vBCD_t vrb)

  Vector Compare Signed BCD Quadword for greater than.
- static int vec_bcdcmple (vBCD_t vra, vBCD_t vrb)

  Vector Compare Signed BCD Quadword for less than or equal.
- static int vec_bcdcmplt (vBCD_t vra, vBCD_t vrb)

Vector Compare Signed BCD Quadword for less than.

- static int vec_bcdcmpne (vBCD_t vra, vBCD_t vrb)

  Vector Compare Signed BCD Quadword for not equal.
- static vBCD_t vec_bcdcpsgn (vBCD_t vra, vBCD_t vrb)

  Vector copy sign BCD.
- static vi128_t vec_bcdctsq (vBCD_t vra)

  Vector Decimal Convert to Signed Quadword.
- static vui8_t vec_bcdctub (vBCD_t vra)

  Vector Decimal Convert Binary Coded Decimal (BCD) digit pairs to binary unsigned bytes .
- static vui16_t vec_bcdctuh (vBCD_t vra)

  Vector Decimal Convert groups of 4 BCD digits to binary unsigned halfwords.
- static vui32_t vec_bcdctuw (vBCD_t vra)

  Vector Decimal Convert groups of 8 BCD digits to binary unsigned words.
- static vui64_t vec_bcdctud (vBCD_t vra)

  Vector Decimal Convert groups of 16 BCD digits to binary unsigned doublewords.
- static vui128_t vec_bcdctuq (vBCD_t vra)

  Vector Decimal Convert groups of 32 BCD digits to binary unsigned quadword.
- static vui8_t vec_bcdctz (vBCD_t vrb)

  Vector Decimal Convert To Zoned.
- static vBCD_t vec_bcddiv (vBCD_t a, vBCD_t b)

  Divide a Vector Signed BCD 31 digit value by another BCD value.
- static vBCD_t vec_bcddive (vBCD_t a, vBCD_t b)

  Decimal Divide Extended.
- static vBCD_t vec_bcdmul (vBCD_t a, vBCD_t b)

  Multiply two Vector Signed BCD 31 digit values.
- static vBCD_t vec_bcdmulh (vBCD_t a, vBCD_t b)

  Vector Signed BCD Multiply High.
- static vBCD_t vec_bcds (vBCD_t vra, vi8_t vrb)

  Decimal Shift. Shift a vector signed BCD value, left or right a variable amount of digits (nibbles). The sign nibble is preserved.
- static vBCD_t vec_bcdsetsgn (vBCD_t vrb)

  Vector Set preferred BCD Sign.
- static vBCD_t vec_bcdslqi (vBCD_t vra, const unsigned int _N)

  Vector BCD Shift Right Signed Quadword.
- static vBCD_t vec_bcdsluqi (vBCD_t vra, const unsigned int _N)

  Vector BCD Shift Right unsigned Quadword.
- static vBCD_t vec_bcdsr (vBCD_t vra, vi8_t vrb)

  Decimal Shift and Round. Shift a vector signed BCD value, left or right a variable amount of digits (nibbles). The sign nibble is preserved. If byte element 7 of the shift count is negative (right shift), and the last digit shifted out is greater then or equal to 5, then increment the shifted magnitude by 1.
- static vBCD_t vec_bcdsrqi (vBCD_t vra, const unsigned int _N)

  Vector BCD Shift Right Signed Quadword Immediate.
- static vBCD_t vec_bcdsrrqi (vBCD_t vra, const unsigned int _N)

  Vector BCD Shift Right and Round Signed Quadword Immediate.
- static vBCD_t vec_bcdsruqi (vBCD_t vra, const unsigned int _N)

  Vector BCD Shift Right Unsigned Quadword immediate.
- static vBCD_t vec_bcdsub (vBCD_t a, vBCD_t b)

  Subtract two Vector Signed BCD 31 digit values.

- static vBCD_t vec_bcdsubcsq (vBCD_t a, vBCD_t b)

  Decimal Sudtract & write Carry Signed Quadword.
- static vBCD_t vec_bcdsubecsq (vBCD_t a, vBCD_t b, vBCD_t c)

  Decimal Add Extended & write Carry Signed Quadword.
- static vBCD_t vec_bcdsubesqm (vBCD_t a, vBCD_t b, vBCD_t c)

  Decimal Subtract Extended Signed Modulo Quadword.
- static vBCD_t vec_bcdtrunc (vBCD_t vra, vui16_t vrb)

  Decimal Truncate. Truncate a vector signed BCD value vra to N-digits, where N is the unsigned integer value in bits 48-63 of vrb. The first 31-N digits are set to 0 and the result returned.
- static vBCD_t vec_bcdtruncqi (vBCD_t vra, const unsigned short _N)

  Decimal Truncate Quadword Immediate. Truncate a vector signed BCD value vra to N-digits, where N is a unsigned short integer constant. The first 31-N digits are set to 0 and the result returned.
- static vBCD_t vec_bcdus (vBCD_t vra, vi8_t vrb)

  Decimal Unsigned Shift. Shift a vector unsigned BCD value, left or right a variable amount of digits (nibbles).
- static vBCD_t vec_bcdutrunc (vBCD_t vra, vui16_t vrb)

  Decimal Unsigned Truncate. Truncate a vector unsigned BCD value vra to N-digits, where N is the unsigned integer value in bits 48-63 of vrb. The first 32-N digits are set to 0 and the result returned.
- static vBCD_t vec_bcdutruncqi (vBCD_t vra, const unsigned short _N)

  Decimal Unsigned Truncate Quadword Immediate. Truncate a vector unsigned BCD value vra to N-digits, where N is a unsigned short integer constant. The first 32-N digits are set to 0 and the result returned.
- static vBCD_t vec_cbcdaddcsq (vBCD_t ∗cout, vBCD_t a, vBCD_t b)

  Combined Decimal Add & Write Carry Signed Quadword.
- static vBCD_t vec_cbcdaddecsq (vBCD_t ∗cout, vBCD_t a, vBCD_t b, vBCD_t cin)

  Combined Decimal Add Extended & write Carry Signed Quadword.
- static vBCD_t vec_cbcdmul (vBCD_t ∗p_high, vBCD_t a, vBCD_t b)

  Combined Vector Signed BCD Multiply High/Low.
- static vBCD_t vec_cbcdsubcsq (vBCD_t ∗cout, vBCD_t a, vBCD_t b)

  Combined Decimal Subtract & Write Carry Signed Quadword.
- static vf64_t vec_pack_Decimal128 (_Decimal128 lval)

  Pack a FPR pair (_Decimal128) to a doubleword vector (vector double).
- static _Decimal128 vec_quantize0_Decimal128 (_Decimal128 val)

  Quantize (truncate) a _Decimal128 value before convert to BCD.
- static vui8_t vec_rdxcf100b (vui8_t vra)

  Vector Decimal Convert Binary Coded Decimal (BCD) digit pairs from radix 100 binary integer bytes.
- static vui8_t vec_rdxcf10kh (vui16_t vra)

  Vector Decimal Convert radix 10,000 Binary halfwords to pairs of radix 100 binary bytes.
- static vui16_t vec_rdxcf100mw (vui32_t vra)

  Vector Decimal Convert radix 10∗∗8 Binary words to pairs of radix 10,000 binary halfwords.
- static vui32_t vec_rdxcf10E16d (vui64_t vra)

  Vector Decimal Convert radix 10∗∗16 Binary doublewords to pairs of radix 10∗∗8 binary words.
- static vui64_t vec_rdxcf10e32q (vui128_t vra)

  Vector Decimal Convert radix 10∗∗32 Binary quadword to pairs of radix 10∗∗16 binary doublewords.
- static vui8_t vec_rdxcfzt100b (vui8_t zone00, vui8_t zone16)

  Vector Decimal Convert Zoned Decimal digit pairs to to radix 100 binary integer bytes..
- static vui8_t vec_rdxct100b (vui8_t vra)

  Vector Decimal Convert Binary Coded Decimal (BCD) digit pairs to radix 100 binary integer bytes.
- static vui16_t vec_rdxct10kh (vui8_t vra)

  Vector Decimal Convert radix 100 digit pairs to radix 10,000 binary integer halfwords.

- static vui32_t vec_rdxct100mw (vui16_t vra)

  Vector Decimal Convert radix 10,000 digit halfword pairs to radix 100,000,000 binary integer words.
- static vui64_t vec_rdxct10E16d (vui32_t vra)

  Vector Decimal Convert radix 100,000,000 digit word pairs to radix 10E16 binary integer doublewords.
- static vui128_t vec_rdxct10e32q (vui64_t vra)

  Vector Decimal Convert radix 10E16 digit pairs to radix 10E32 __int128 quadwords.
- static vb128_t vec_setbool_bcdinv (vBCD_t vra)

  Vector Set Bool from Signed BCD Quadword if invalid.
- static vb128_t vec_setbool_bcdsq (vBCD_t vra)

  Vector Set Bool from Signed BCD Quadword.
- static int vec_signbit_bcdsq (vBCD_t vra)

  Vector Sign bit from Signed BCD Quadword.
- static _Decimal128 vec_unpack_Decimal128 (vf64_t lval)

  Unpack a doubleword vector (vector double) into a FPR pair. (_Decimal128).
- static vui128_t vec_zndctuq (vui8_t zone00, vui8_t zone16)

  Vector Zoned Decimal Convert 32 digits to binary unsigned quadword.

7.1.1 Detailed Description

Header package containing a collection of Binary Coded Decimal (**BCD**) computation and Zoned Character conversion operations on vector registers.

Many of these operations are implemented in a single VMX or DFP instruction on newer (POWER8/POWER9) processors. This header serves to fill in functional gaps for older (POWER7, POWER8) processors (using existing VMX, VSX, and DFP instructions) and provides in-line assembler implementations for older compilers that do not provide the built-ins.

Starting with POWER6 introduced a Decimal Floating-point (*DFP*) Facility implementing the `IEEE 754-2008 revision` standard. This is implemented in hardware as an independent Decimal Floating-point Unit (*DFU*). This is supported with ISO C/C++ language bindings and runtime libraries.

The DFP Facility supports a different data format `Densely packed decimal` (*DPD* and a more extensive set of operations then BCD or Zoned. So DFP and the comprehensive C language and runtime library support makes it a better target for new business oriented applications. As the DFP Facility supports conversions between DPD and BCD, existing DFP operations can be used to emulate BCD operations on older processors and fill in operational gaps in the vector BCD instruction set.

As DFP is supported directly in the hardware and has extensive language and runtime support, there is little that P←VECLIB can contribute to general decimal radix computation. However the vector unit and recent BCD and Zoned extensions can still be useful in areas include large order multiple precision computation and conversions between binary and decimal radix. Both are required to convert large decimal numeric or floating-point values with extreme exponents for input or print.

So what operations are needed, what does the PowerISA provide, and what does the ABI and/or compiler provide. Some useful operations include:

- conversions between BCD and __int128
  - As intermediate step between external decimal/_Decimal128 and _Float128

- Conversions between BCD and Zoned (character)
- Conversions between BCD and DFP
- BCD add/subtract with carry/extend
- BCD compare equal, greater than, less than
- BCD copy sign and set bool from sign
- BCD digit shift left/right
- BCD multiply/divide

The original VMX (AKA Altivec) only defined a few instructions that operated on the 128-bit vector as a whole. This included the vector shifts by bit and octet, and generalized vector permute, general binary integer add, subtract and multiply for byte/halfword/word. But no BCD or decimal character operations.

POWER6 introduced the Decimal Floating-point Facility. DFP provides a robust set of operations with 7 (_Decimal32), 16 (_Decimal64), and 34 (_Decimal128) digit precision. Arithmetic operations include add, subtract, multiply, divide, and compare. Special operations insert/extract exponent, quantize, and digit shift. Conversions to and from signed (31-digits) and unsigned (32-digit) BCD. And conversions to and from binary signed long (64-bit) integer. DFP operations use the existing floating-point registers (FPRs). The 128-bit DFP (quadword) instructions operate on even/odd 64-bit Floating-point register pairs (FPRp).

POWER6 also implemented the Vector Facility (VMX) instructions. No additional vectors operations where added and the Vector Registers (VRs) where separate from the GRPs and FPRs. The only transfer data path between register sets is via storage. So while the DFP Facility could be used for BCD operations and conversions, there was little synergy with the vector unit, in POWER6.

POWER7 introduced the VSX facility providing 64x128-bit Vector Scalar Registers (VSRs) that overlaid both the FPRs (VSRs 0-31) and VRs (VSRs 32-63). It also added useful doubleword permute immediate (xxpermdi) and logical/select operations with access to all 64 VSRs. This greatly simplifies data transfers between VRs and FPRs (FPRps) (see vec_pack_Decimal128(), vec_unpack_Decimal128()). This makes it more practical to transfer vector contents to the DFP Facility for processing (see vec_BCD2DFP() and vec_DFP2BCD().

Note

> All the BCD instructions and the quadword binary add/subtract are defined as vector class and can only access vector registers (VSRs 32-63). The DFP instructions can only access FPRs (VSRs 0-31). So only a VSX instruction (like xxpermdi) can perform the transfer without going through storage.

POWER8 added vector add/subtract modulo/carry/extend unsigned quadword for binary integer (vector [unsigned] ↩ __int128). This combined with the wider (word) multiply greatly enhances multiple precision operations on large (> 128-bit) binary numbers. POWER8 also added signed BCD add/subtract instructions with up to 31-digits. While the PowerISA did not provide carry/extend forms of bcdadd/bcdsub, it does set a condition code with bits for GT/LT/EQ/O↩ VF. This allows for implementations of BCD compare and the overflow (OVF) bit supports carry/extend operations. Also the lack of BCD multiply/divide in the vector unit is not a problem because we can leverage DFP (see vec_bcdmul(), vec_bcddiv()).

POWER9 (PowerISA 3.0B) adds BCD copy sign, set sign, shift, round, and truncate instructions. There are also unsigned (32-digit) forms of the shift and truncate instructions. And instructions to convert between signed BCD and quadword (__int128) and signed BCD and Zoned. POWER9 also added quadword binary multiply 10 with carry extend forms than can also help with decimal to binary conversion.

The `OpenPOWER ABI` does have an *Appendix B. Binary-Coded Decimal Built-In Functions* and proposes that compilers provide a **bcd.h** header file. At this time no compiler provides this header. GCC does provides compiler built-ins to generate the bcdadd/bcdsub instructions and access the associated condition codes in *if* statements. GCC also provides built-ins to generate the DFP instruction encode/decode to and from BCD.

Note

> The compiler disables built-ins if the **mcpu** target does not enable the specific instruction. For example if you compile with **-mcpu=power7**, __builtin_bcdadd and __builtin_bcdsub are not supported. But vec_bcdadd() is always defined in this header, will generate the minimum code, appropriate for the target, and produce correct results.

This header covers operations that are either:

- Operations implemented in hardware instructions for later processors and useful to programmers, on slightly older processors, even if the equivalent function requires more instructions. Examples include quadword BCD add and subtract.
- Defined in the OpenPOWER ABI but *not* yet defined in <altivec.n> or <bcd.h> provided by available compilers in common use. Examples include bcd_add, bcd_cmpg and bcd_mul.
- Commonly used operations, not covered by the ABI or <altivec.h>, and require multiple instructions or are not obvious. Examples include vec_pack_Decimal128() and vec_unpack_Decimal128().

See Returning extended quadword results. for more background on extended quadword computation.

7.1.2 Endian problems with quadword implementations

Technically, operations on quadword elements should not require any endian specific transformation. There is only one element so there can be no confusion about element numbering or order. However some of the more complex quadword operations are constructed from operations on smaller elements. And those operations as provided by <altivec.h> are required by the OpenPOWER ABI to be endian sensitive. See Endian problems with doubleword operations for a more detailed discussion.

In any case, the arithmetic (high to low) order of digit nibbles in BCD or characters in Zoned are defined in the Power↩ ISA. In the vector register, high order digits are on the left while low order digits and the sign are on the right. (See vec_bcdadd() and vec_bcdsub()). So pveclib implementations will need to either:

- Nullify little endian transforms of <altivec.h> operations. The <altivec.h> built-ins vec_mule(), vec_mulo(), and vec_pack() are endian sensitive and often require nullification that restores the original operation.
- Use new operations that are specifically defined to be stable across BE/LE implementations. The pveclib operations; vec_vmuleud() and vec_mulubm() are defined to be endian stable.

7.1.3 Some details of BCD computation

`Binary-coded decimal` (Also called *packed decimal*) and the related *Zoned Decimal* are common representations of signed decimal radix (base 10) numbers. BCD is more compact and usually faster then zoned. Zoned format is more closely aligned with human readable and printable character formats. In both formats the sign indicator is associated (in the same character or byte) with the low order digit.

BCD and Zoned formats and operations were implemented for some of the earliest computers. Then circuitry was costly and arithmetic was often implemented as a digit (or bit) serial operation. Modern computers have more circuitry with wider data paths and more complex arithmetic/logic units. The current trend is for each processor core implementation to include multiple computational units that can operate in parallel.

For POWER server class processors separate and multiple Fixed-Point Units (FXU), (binary) Floating-point Units (FPU), and Vector Processing Units (VPU) are the norm. POWER6 introduced a Decimal Floating-point (*DFP*) Facility implementing the `IEEE 754-2008 revision` standard. This is implemented in hardware as an independent Decimal Floating-point Unit (*DFU*). This is supported with ISO C/C++ language bindings and runtime libraries.

The DFU supports a different data format `Densely packed decimal` (*DPD* and a more extensive set of operations then BCD or Zoned. So hardware DFP and the comprehensive C language and runtime library support makes it a better target for new business oriented applications. As DFP is supported directly in the hardware and has extensive language and runtime support, there is little that PVECLIB can contribute to general decimal radix computation.

Note

> BCD and DFP support requires at least PowerISA 2.05 (POWER6) or later server level processor support.

However the vector unit and recent BCD and Zoned extensions can still be useful in areas including large order multiple precision computation and conversions between binary and decimal radix. Both are required to convert large decimal numeric or floating-point values with extreme exponents for input or print. And conventions between _Float128 and _Decimal128 types is even more challenging. Basically both POSIX and IEEE 754-2008 require that it possible to convert floating-point values to an external character decimal representation, with the specified rounding, and back recovering the original value. This always requires more precision for the conversion then is available in the given format and size.

7.1.3.1 Preferred sign, zone, and zero.

BCD and Zoned Decimal have a long history with multiple computer manufacturers, and this is reflected as multiple encodings of the same basic concept. This is in turn reflected in the PowerISA as Preferred Sign **PS** immediate operand on BCD instructions.

This header implementation assumes that users of PVECLIB are not interested in this detail and just want access to BCD computation with consistent results. So PVECLIB does not expose preferred sign at the API and provides reasonable defaults in the implementation.

PVECLIB is targeted at the Linux ecosystem with ASCII character encoding, so the implementation defaults for:

- preferred zone nibble 0x3. ASCII encodes decimal characters as 0x30 - 0x39.
- preferred sign code nibbles 0xC and 0xD. Historically accounting refers to ***Credit*** as positive and ***Dedit*** for negative.

The PowerISA implementation is permissive of sign encoding of input values and will accept four (0xA, 0xC, 0xE, 0xF) encodings of positive and two (0xB, 0xD) for negative. But the sign code of the result is always set to the preferred sign.

The BCD encoding allows for signed zeros (-0, +0) but the PowerISA implementation prefers the positive encoding for zero results. Again the implementation is permissive of both encodings for input operands. Usually this is not an issue but can be when dealing with conversions from other formats (DFP also allows signed 0.0) and implementations of BCD operations for older (POWER7/8) processors.

This is most likely to effect user code in comparisons of BCD values for 0. One might expect the following vector binary word compare all

```
if (vec_all_eq((vui32_t) t, (vui32_t) _BCD_CONST_ZERO))
```

to give the same result as

```
if (vec_bcdcmpeq (t, _BCD_CONST_ZERO))
```

The vector binary compare is likely to have lower latency (on POWER7/8), but will miss compare on *-0*. The BCD compare operation (i.e. vec_bcdcmpeq ()) is recommended, unless the programs knows the details for the source operands generation, and have good (performance and latency) reasons to to use the alternative compare. Pveclib strives to provide correct preferred zeros results in its implementation of BCD operations.

7.1.3.2 Extended Precision computation with BCD

Extended precision requires carry and extend forms of bcdadd/sub. Also BCD multiply with multiply high and and double quadword (62-digit) forms. The vector unit does not support BCD multiply so pveclib leverages the DFP Facility to implement these operations. Finally algorithms and extended precision conversions require BCD divide and divide extended. Again leveraging the DPU to implement these operations.

7.1.3.2.1 Vector Add/Subtrace with Carry/Extend example The PowerISA does not provide the extend and write-carry forms of the bcdadd/sub instructions. But bcdadd/sub instructions do post status to CR field 6 which includes:

- Result is less than zero (CR.bit[56])
- Result is greater than zero (CR.bit[57])
- Result is equal to zero (CR.bit[58])
- Result overflowed (CR.bit[59])

which provides a basis for BCD comparison and the overflow may be used for carry/extend logic. The GCC compiler provides built-ins to generate the bcdadd/sub and test the resulting CR bits in if statements.

Unfortunately, the Overflow flag generated by bcdadd/bcdsub is not a true carry/borrow. If the operands have the same sign for bcdadd (different sign for bcdsub) and there is a carry out of the high order digit, then:

- The sum is truncated to the low order 31 digits
- The sum's sign matches the operands signs
- The overflow flag (CR.bit[59]) is set.

Note

> overflow is only set in conjunction with greater than zero (positive) or less than zero (negative) results. This implies that BCD carries are tri-state; +1, 0, or -1.

This can be used to simulate a **Add and Write-Carry** operation. However if the operands have different signs the bcdadd (same sign for bcdsub) the operation does the following:

- The smaller magnitude is subtracted from the larger magnitude.
- The sign matches the sign of the larger magnitude.
- The ox_flag (CR.bit[59]) is NOT set.

For a simple BCD add this is the desired result (overflow is avoided and the borrow is recorded in the sign). But for multiple precision BCD operation, this will delay propagation of borrows to the higher order digits and the result is a mixture of signs across elements of the larger multiple precision value. This would have to be corrected at some later stage. For example the sum of 32 digits:

```
  21000000000000000000000000000008
+ 19000000000000000000000000000008
= 40000000000000000000000000000016
```

This exceeds the 31-digit capacity of Vector signed BCD so we are forced to represent each number as two or more BCD values. For example:

```
  0000000000000000000000000000002c 1000000000000000000000000000008c
+ 0000000000000000000000000000001c 9000000000000000000000000000008c
= 0000000000000000000000000000004c 0000000000000000000000000000016c
```

The sum of the low order operands will overflow, so we need to detect this overflow and generate a carry that we can apply to sum of the high order operands. For example the following code using the GCC's __builtin_bcdadd_ov.

```
static inline vBCD_t
vec_bcdaddcsq (vBCD_t a, vBCD_t b)
{
  vBCD_t c, sum_ab;
  c = _BCD_CONST_ZERO;
  // compute the sum of (a + b)
  sum_ab = (vBCD_t) __builtin_bcdadd ((vi128_t) a, (vi128_t) b, 0);;
  // Detect the overflow, which should be rare
  if (__builtin_expect (__builtin_bcdadd_ov ((vi128_t) a, (vi128_t) b, 0), 0))
    {
      // use copysign to generate a carry based on the sign of the sum_ab
      c = vec_bcdcpsgn (_BCD_CONST_PLUS_ONE, sum_ab);
    }
  return (c);
}
  0000000000000000000000000000002c   1000000000000000000000000000008c
  0000000000000000000000000000001c + 9000000000000000000000000000008c
                                   =
                                1c   0000000000000000000000000000016c
+ 0000000000000000000000000000002c
+ 0000000000000000000000000000001c
= 0000000000000000000000000000004c
```

The higher operands requires a 3-way (a+b+c) sum to propagate the carry.

```
static inline vBCD_t
vec_bcdaddesqm (vBCD_t a, vBCD_t b, vBCD_t c)
{
  vBCD_t t;
  t = vec_bcdadd (vec_bcdadd (a, b), c);
  return (t);
}
```

where vec_bcdadd is a pveclib wrapper around __builtin_bcdadd to simplify the code. The simplified multiple precision BCD use case looks like this:

```
// r_h|r_l = a_h|a_l + b_h|b_l
r_l = vec_bcdadd (a_l, b_l);
c_l = vec_bcdaddcsq (a_l, b_l);
r_h = vec_bcdaddesqm (a_h, b_h, c_l)
```

But we should look at some more examples before we assume we have a complete solution. For example a subtract that requires a borrow:

```
  21000000000000000000000000000008
- 19000000000000000000000000000008
= 02000000000000000000000000000000
```

The multiple precision BCD would look like this:

```
  0000000000000000000000000000002c 1000000000000000000000000000008c
+ 0000000000000000000000000000001d 9000000000000000000000000000008d
```

But with the example code above we expected result:

```
= 0000000000000000000000000000000c 2000000000000000000000000000000c
```

instead we see:

```
= 0000000000000000000000000000001c 8000000000000000000000000000000d
```

The BCD overflow flag only captures carry/borrow when the bcdadd operands have the same sign (or different signs for bcdsub). In this case it looks like (1 - 9 = -8) which does not overflow.

```
  000000000000000000000000000000002c   10000000000000000000000000000008c
  000000000000000000000000000000001d + 90000000000000000000000000000008d
                                     =
                                  0c   80000000000000000000000000000000d
+ 000000000000000000000000000000002c
+ 000000000000000000000000000000001d
= 000000000000000000000000000000001c
```

We need a way to detect the borrow and fix up the sum to look like (11 - 9 = 2) and generate a carry digit (-1) to propagate the borrow to the higher order digits.

The secondary borrow is detected by comparing the sign of the result to the sign of the first operand. Something like this:

```
t = _BCD_CONST_ZERO;
sign_ab = vec_bcdcpsgn (sum_ab, a);
if (!vec_all_eq(sign_ab, sum_ab))
  {
  // Borrow fix-up code
  }
```

For multiple precision operations it would be better to retain the sign from the first operand and generate a borrow digit (value of '1' with the sign of the uncorrected result).

This requires re-computing the sum/difference, while applying the effect of borrow, and replacing the carry (currently 0) with a signed borrow digit. The corrected sum is the 10's complement (9's complement +1) of the initial sum (like (10 - 8 = 2) or (9 - 8 + 1 = 2). As we obviously don't know how to represent signed BCD with more then 31-digits (10∗∗32 is 32-digits), the 9's complement + 1 is a better plan. We know that initial sum has a different sign from the original first operand. So adding 10∗∗31 with the sign of the first operand to the initial sum applies the borrow operation.

```
c = _BCD_CONST_ZERO;
sign_ab = vec_bcdcpsgn (sum_ab, a);
if (!vec_all_eq(sign_ab, t) && !vec_all_eq(_BCD_CONST_ZERO, t))
  {
    // 10**31 with the original sign of the first operand
    vBCD_t nines = vec_bcdcpsgn (_BCD_CONST_PLUS_NINES, a);
    vBCD_t c10s  = vec_bcdcpsgn (_BCD_CONST_PLUS_ONE, a);
    // Generate the Borrow digit from the initial sum
    c = vec_bcdcpsgn (_BCD_CONST_PLUS_ONE, sum_ab);
    // Invert the sum using the 10s complement
    sum_ab = vec_bcdaddesqm (nines, sum_ab, c10s);
  }
  000000000000000000000000000000002c   10000000000000000000000000000008c
  000000000000000000000000000000001d + 90000000000000000000000000000008d
                                     = ?
                                       80000000000000000000000000000000d
                                     + 9999999999999999999999999999999c
                                     + 000000000000000000000000000000001c
                                  1d   20000000000000000000000000000000c
+ 000000000000000000000000000000002c
+ 000000000000000000000000000000001d
= 0000000000000000000000000000000000c
```

This does not fit well into the separate *add modulo* and *add and write-carry* operations commonly used for fixed binary arithmetic. Instead it requires a combined operation returning both the generated borrow and a sum/difference result with a corrected sign code. The combined add with carry looks like this:

```
static inline vBCD_t
vec_cbcdaddcsq (vBCD_t *cout, vBCD_t a, vBCD_t b)
{
  vBCD_t t, c;
  vBCD_t sum_ab, sign_a, sign_ab;
  sum_ab = vec_bcdadd (a, b);
  if (__builtin_expect (__builtin_bcdadd_ov ((vi128_t) a, (vi128_t) b, 0), 0))
    {
      c = vec_bcdcpsgn (_BCD_CONST_PLUS_ONE, sum_ab);
    }
  else // (a + b) did not overflow, but did it borrow?
    {
      c = _BCD_CONST_ZERO;
      sign_ab = vec_bcdcpsgn (sum_ab, a);
      if (!vec_all_eq(sign_ab, sum_ab) && !vec_all_eq(_BCD_CONST_ZERO, t))
```

```
        {
          // 10**31 with the original sign of the first operand
          vBCD_t nines = vec_bcdcpsgn (_BCD_CONST_PLUS_NINES, a);
          vBCD_t c10s  = vec_bcdcpsgn (_BCD_CONST_PLUS_ONE, a);
          // Generate the Borrow digit from the initial sum
          c = vec_bcdcpsgn (_BCD_CONST_PLUS_ONE, sum_ab);
          // Invert the sum using the 10s complement
          sum_ab = vec_bcdaddesqm (nines, sum_ab, c10s);
        }
    }
  *cout = c;
  return (sum_ab);
}
```

and the usage example looks like this:

```
// r_h|r_l = a_h|a_l + b_h|b_l
r_l = vec_cbcdaddcsq (&c_l, a_l, b_l);
r_h = vec_bcdaddesqm (a_h, b_h, c_l)
```

Todo The BCD add/subtract extend/carry story is not complete. The carry extend operations based only on the **OV** condition codes only works as expected for bcdadd operands with the same sign and bcdsub with different signs. See vec_bcdaddcsq() and vec_bcdaddecsq(). Extended BCD difference (or subtract the same sign or add with different signs) is more complicated. See vec_bcdsubcsq() and vec_bcdsubecsq(). Generating a true borrow seems to require looking one (31-digit) column ahead or behind. The first attempt at generating correct borrowing is implemented in vec_cbcdaddcsq() and vec_cbcdaddecsq(). There are still cases where these operation will generate a borrow and invert (10s complement) incorrectly. The net seems to be that for BCD multiple precision difference to work correctly, the larger magnitude must be the first operand.

7.1.3.2.2 Vector BCD Multiply/Divide Quadword example

BCD multiply and divide operations are not directly supported in the current PowerISA. Decimal multiply and divide are supported in the Decimal Floating-point (DFP) Facility, as well as conversion to and from signed (unsigned) BCD.

So BCD multiply and divide operations can be routed through the DFP Facility with a few caveats.

- DFP Extended format supports up to 34 digits precision
- DFP significand represent digits to the *left* of the implied decimal point.
- DFP finite number are not normalized.

This allows DFP to represent decimal integer and fixed point decimal values with a preferred exponent of 0. The DFP Facility will maintain this preferred exponent for DPF arithmetic operations until:

- An arithmetic operation involves a operand with a non-zero exponent.
- A divide operation generates a result with fractional digits
- A multiply operation generates a result that exceeds 34 digits.

The implementation can insure that input operands are derived from 31-digit BCD values. The results of any divide operations can be truncated back to decimal integer with the preferred 0 exponent. This can be achieved with the DFP Quantize Immediate instruction, specifying the ideal exponent of 0 and a rounding mode of *round toward 0* (see vec_quantize0_Decimal128()). This allows the following implementation:

```
static inline vBCD_t
vec_bcddiv (vBCD_t a, vBCD_t b)
{
  vBCD_t t;
  _Decimal128 d_t, d_a, d_b;
  d_a = vec_BCD2DFP (a);
  d_b = vec_BCD2DFP (b);
  d_t = vec_quantize0_Decimal128 (d_a / d_b);
  t = vec_DFP2BCD (d_t);
  return (t);
}
```

The multiply case is bit more complicated as we need to produce up to 62 digit results without losing precision and DFP only supports 34 digits. This requires splitting the input operands into groups of digits where partial products of any combination of these groups is guaranteed not exceed 34 digits.

One way to do this is split each 31-digit operand into two 16-digit chunks (actually 15 and 16-digits). These chunks are converted to DFP extended format and multiplied to produce four 32-digit partial products. These partial products can be aligned and summed to produce the high and low 31-digits of the full 62-digit product. This is the basis for vec_bcd_mul(), vec_bcdmulh(), and vec_cbcdmul().

A simple vec_and() can be used to isolate the low order 16 BCD digits. It is simple at this point to detect if both operands are 16-digits or less by comparing the original operand to the isolate value. In this case the product can not exceed 32 digits and we can short circuit the product to a single multiply. Here we can safely use binary compare all.

```
const vBCD_t dword_mask = (vBCD_t) CONST_VINT128_DW(15, -1);
vBCD_t t, low_a, low_b, high_a, high_b;
_Decimal128 d_p, d_t, d_a, d_b;
low_a = vec_and (a, dword_mask);
low_b = vec_and (b, dword_mask);
d_a = vec_BCD2DFP (low_a);
d_b = vec_BCD2DFP (low_b);
d_p = d_a * d_b;
if (__builtin_expect ((vec_cmpuq_all_eq ((vui128_t) low_a, (vui128_t) a)
    && vec_cmpuq_all_eq ((vui128_t) low_b, (vui128_t) b)), 1))
  {
    d_t = d_p;
  }
else
  {
  ...
  }
t = vec_DFP2BCD (d_t);
```

This is a case where negative 0 can be generated in the DFP multiply and converted unchanged to BCD. This is handled with the following fix up code:

```
  // Minus zero
  const vui32_t mz = CONST_VINT128_W (0, 0, 0, 0x0000000d);
  ...
#ifdef _ARCH_PWR9
  t = vec_bcdadd (t, _BCD_CONST_ZERO);
#else
  if (vec_all_eq((vui32_t) t, mz))
    t = _BCD_CONST_ZERO;
#endif
  return t;
```

From here the code diverges for multiply low and multiply high (and full combined multiply). Multiply low only needs the 3 lower order partial products. The highest order partial product does not impact the lower order 31-digits and is not needed. Multiply high requires the generation and summation of all 4 partial products. Following code completes the implementation of BCD multiply low:

```
...
else
  {
    _Decimal128 d_ah, d_bh, d_hl, d_lh, d_h;
    high_a = vec_bcdsrqi (a, 16);
    high_b = vec_bcdsrqi (b, 16);
    d_ah = vec_BCD2DFP (high_a);
    d_bh = vec_BCD2DFP (high_b);
    d_hl = d_ah * d_b;
    d_lh = d_a * d_bh;
    d_h = d_hl + d_lh;
    d_h = __builtin_dscliq (d_h, 17);
    d_h = __builtin_dscriq (d_h, 1);
    d_t = d_p + d_h;
  }
```

Here we know that there are higher order digits in one or both operands. First use vec_bcdsrqi() to isolate the high 15-digits of operands a and b. Both Vector unit and DFP Facility have decimal shift operations, but the vector shift operation is faster.

Then convert to DFP and multiply (high_a ∗ low_b and high_b ∗ low_a) for the two middle order partial products which are summed. This sum represents the high 32-digits (the 31-digit sum can carry) of a 48-digit product. Only the lower 16-digits of this sum is needed for the final sum and this needs to be aligned with the high 16 digits of the original lower order partial product.

For this case use **DFP Shift Significand Left Immediate** and **DFP Shift Significand Right Immediate**. All the data is in the DFP Facility and the high cost of the DFP Facility shift is offset by avoiding extra format conversions. We use shift left 17 followed by shift right 1 to clear the highest order DFP digit and avoid any overflow. A final DFP add produces the low order 32 digits of the product which will be truncated to 31-digits in the conversion to BCD.

How we can look at the BCD multiply high (generate the full 62-digit product returning the high 31 digits) and point out the differences. Multiply high also starts by isolating the low order 16 BCD digits, performing the low order multiply (low_a ∗ low_b), and testing for the short circuit (all higher order digits are 0). The first difference (from multiply low) is that in this case only the high digit of the potential 32-digit product is returned.

```
const vBCD_t dword_mask = (vBCD_t) CONST_VINT128_DW(15, -1);
vBCD_t t, low_a, low_b, high_a, high_b;
_Decimal128 d_p, d_t, d_a, d_b;
low_a = vec_and (a, dword_mask);
low_b = vec_and (b, dword_mask);
d_a = vec_BCD2DFP (low_a);
d_b = vec_BCD2DFP (low_b);
d_p = d_a * d_b;
if (__builtin_expect ((vec_cmpuq_all_eq ((vui128_t) low_a, (vui128_t) a)
    && vec_cmpuq_all_eq ((vui128_t) low_b, (vui128_t) b)), 1))
  {
    d_t = __builtin_dscriq (d_p, 31);
  }
else
  {
  ...
  }
t = vec_DFP2BCD (d_t);
```

So the short circuit code shifts the low partial product right 31 digits and returns that value.

If we can not short circuit, Multiply high requires the generation and summation of all four partial products. Following code completes the implementation of BCD multiply high:

```
...
else
  {
    _Decimal128 d_ah, d_bh, d_hl, d_lh, d_h, d_ll, d_m;
    high_a = vec_bcdsrqi (a, 16);
    high_b = vec_bcdsrqi (b, 16);
    d_ah = vec_BCD2DFP (high_a);
    d_bh = vec_BCD2DFP (high_b);
    d_hl = d_ah * d_bl;
    d_lh = d_al * d_bh;
    d_ll = __builtin_dscriq (d_p, 16);
    d_m = d_hl + d_lh + d_ll;
    d_m = __builtin_dscriq (d_m, 15);
    d_h = d_ah * d_bh;
    d_h = __builtin_dscliq (d_h, 1);
    d_t = d_m + d_h;
  }
```

Again we know that there are higher order digits in one or both operands and use vec_bcdsrqi() to isolate the high 15-digits of operands a and b. Then convert to DFP and multiply (high_a ∗ low_b and high_b ∗ low_a) for the two middle order partial products (d_hl and d_lh).

The low order partial product (d_p) was generated above but we need only the high order 15 digits for summation. Shift the low partial product right 16 digits then sum (d_hl + d_lh + d_ll) the low and middle order partial products. This produces the high 32 digits of the lower 48 digit partial sum. Shift this right 15 digits to align with the high order 31 digits for the product.

Then multiply (high_a ∗ high_b) to generate the high order partial product. This represents the high 30 digits of a 62 digits. Shift this left 1 digit to correct the alignment. The sum of the adjusted high and middle order partials gives the high order 31 digits of the 62-digit product.

7.1.3.2.3 Vector BCD to/from Binary conversion

Conversions between Decimal (BCD, Zoned, or string) and binary is another topic which is more complicated that it first appears. Everyone that takes computer science should have learned about *atoi* and *itoa* for conversions between strings of decimal character and binary integers.

ASCII to integer is basically;

- initialize a integer accumulator to 0
- loop
  - multiply the accumulator by 10
  - load the next character and convert to a binary decimal digit
  - Add this digit to the accumulator
- repeat until end of string.

Integer to ASCII is basically;

- initialize a temp variable with the integer number
- loop
  - compute the remainder/modulo of temp by 10
  - convert this binary digit to a character and store as the next char
  - divide temp by 10 and use that for the next iteration
- repeat until temp is zero.

You may have noticed that the algorithms above are not exactly vector ready. Both are serialized on expensive multiply and divide operations. This is not so bad for 9 digit (32-bit) integers but will be noticeable when converting between 128-bit binary and 31-digit BCD.

For the vector BCD equivalent of ***atoi*** we could use the PVECLIB implementation of **Vector Multiply by 10 Extended Unsigned Quadword**. For POWER8, vec_mul10euq() uses; multiple even/odd, a couple of shift left octet immediates, and add quadword. This sequence runs 5-7 instructions and has a minimum latency of 13 cycles. To convert from BCD to binary we need to shift and isolate, one BCD digit at time, then feed that into vec_mul10euq(). Ignoring for now the latency associated with shifting the BCD digits, we can quickly estimate $13 * 32 = 416$ cycles to convert 32 digits.

For the vector BCD equivalent of ***itoa*** we could use the POWER8 **Decimal Add Modulo** instruction. For POWER8 vec_bcdadd() has a latency of 13 cycles. But the conversion would be one bit at a time. Use vec_bcdadd() to multiply by 2 then shift / issolate a bit from the binary value, format / convert that bit to BCD 0/1. and vec_bcdadd() again. So a quick estimate for this conversion is $13 * 2 * 128 = 3328$ cycles.

7.1.3.2.3.1 Vector Parallel conversion

Clearly just using bigger registers for bigger numbers is not helping. So we want to think about algorithms that do more in parallel and leverage the vector unit we have.

For POWER9 we have Decimal Convert From/To Signed Quadword and Decimal Convert From/To Zoned (See vec_bcdcfsq(), vec_bcdctsq(), vec_bcdcfz(), vec_bcdctz()). These provide direct conversion between quadword binary and signed BCD and between signed BCD and zoned characters.

The BCD convert from/to Zoned are simple operation that run 3 cycles latency on POWER9 and 14-27 cycles for the POWER8 implementation. For POWER8 there is some additional complexity verify and converting the preferred *sign code* between BCD and Zoned (of course they are different).

But the BCD convert from/to Signed Quadword operations are a bit heavier, running 37 and 23 cycles latency on PO↵WER9. These instructions execute in the DFU and so are single issue. They also keeps the DFU pipeline busy (for 25 and 11 cycles) and block execution of the next DFU operation for a while. Still this is better than the serial conversion examples described above.

But part of the value of PVECLIB is to provide support across POWER7/8/9 and across compiler versions. The convert instructions above are not supported in current compilers with built-ins so PVECLIB provides in-line assembler implementations for these operations. Now we need look into better algorithms for implementing these operations on POWER7/8.

The Vector unit can multiply, add, or subtract integer elements in parallel. The conversion process is basically multiply and add/sub as we can replace divide operations with the multiplicative inverse. So if we are looking for a way to break the conversion down into steps that can be performed in parallel on elements of the larger value and require fewer steps.

For now we can simplify the problem to unsigned radix conversion and deal with signed conversion as a later cleanup step based on the complete unsigned conversion.

7.1.3.2.3.2 Vector Parallel BCD to quadword conversion

Starting with BCD (Radix 10) to Binary (Radix 2) conversion. The data is represented as 32 BCD digits encoded as 4-bit *nibbles* starting with high orders digits on the left, to low order digits on the right.

Said differently, unsigned BCD vectors are represented as 16-bytes each containing a pair of BCD digits, each in the range 00-99. This is helpful because the PowerISA has instructions that multiply and add integer bytes, in parallel. So it seems possible to convert bytes containing even/odd pairs of BCD digits to integer bytes, each in the range 0-99: simply multiply the even digit by 10 and add the odd digit.

The result is a vector of 16 x radix-100 bytes (binary integers in the range 0-99). Said differently a radix 100 vector represented as 8 halfwords each containing a pair of radix 100 digits, each in the range 0-99. Again these pairs of digits (bytes) can be converted by multiply and add to radix 10,000 halfwords.

Repeat the process three more times:

- convert 8 halfwords pairwise into 4 words each containing values in the range 0-99999999 (radix 10∗∗8 digits).
- convert 4 words pairwise into 2 doublewords, each containing values in the range 0-9999999999999999 (radix 10∗16 digits).
- convert 2 doublewords pairwise into a quadword integer in the range 0-99999999999999999999999999999999.

So in 5 steps, each only using vector multiply and add, we convert 32 BCD digits to a quadword integer.

Note

Actually 10∗∗32 can be represented in 107 bits, but who is counting.

Actually, it is a little more complicated than multiply and add. The digits of the digit pair must be isolated and shifted into alignment before the multiply and add. Looking something like this:

```
vui8_t
test_vec_rdxct100b_0 (vui8_t vra)
{
  vui8_t x10, c10, high_digit, low_digit;
  // Isolate the low_digit
  low_digit = vec_slbi (vra, 4);
  low_digit = vec_srbi (low_digit, 4);
  // Shift the high digit into the units position
  high_digit = vec_srbi (vra, 4);
  // multiply the high digit by 10
  c10 = vec_splats ((unsigned char) 10);
  x10 = vec_mulubm (high_digit, c10);
  // add the low_digit to high_digit * 10.
  return vec_add (x10, low_digit);
}
```

The PowerISA does not provide general *nibble* arithmetic, only byte. So the first operations involve isolating each nibble into separate (high_digit and low_digit) bytes. The high_digit shift also aligns the binary for the multiply and add.

The Multiply Unsigned Byte Modulo (vec_mulubm()) generates vmuleub/vmuloub then loads a permute control vector and permutes the low order bytes of the halfword (even/odd) products into a single vector. Finally, add the x10 product and low_digit to get the binary value in the range 0-99.

This sequence runs 6-10 instructions and 13-22 cycles latency. The lower values assume the shift control and permute control vectors are commoned with other operations.

This is a case where the process on paper is much simpler than the reality of programming computers. The operation is actually (bcd_byte / 16 ∗ 10) + (bcd_byte ∗ 16 / 16) where 16 is the *alignment* radix and 10 is the *decimal* radix at this step. The alignment radix operations are (fortunately) strength reduced to vector byte shift left/right.

Let's use a little algebra to eliminate some of these steps. One approach is to generate a correction factor from the high_digit and the difference between the *alignment* and decimal radix. This correction factor is subtracted directly from the original BCD byte and reduces the operation to (bcd_byte - ((bcd_byte / 16) x (16 - 10)) Which looks something like:

```
vui8_t
test_vec_rdxct100b_1 (vui8_t vra)
{
  vui8_t x6, c6, high_digit;
  // Compute the high digit correction factor. For BCD to binary 100s
  // this is the isolated high digit multiplied by the radix difference
  // in binary.  For this stage we use 0x10 - 10 = 6.
  high_digit = vec_srbi (vra, 4);
  c6 = vec_splats ((unsigned char) (16-10));
  x6 = vec_mulubm (high_digit, c6);
  // Subtract the high digit correction bytes from the original
  // BCD bytes in binary.  This reduces byte range to 0-99.
  return vec_sub (vra, x6);
}
```

Another opportunity is to let the compiler strength reduce the multiply to shift and add. Newer versions of GCC will perform this optimization when using the generic vec_mul built-in for vector integer elements.

Note

Previous to GCC 8, vec_mul() was only supported for vector float and double.

```
#if (__GNUC__ > 7)
  x6 = vec_mul (high_digit, c6);
#else
  x6 = vec_mulubm (high_digit, c6);
#endif
```

This eliminates vector multiply even/odd, the permute, and the load associated with the permute. The final sequence runs 5-7 instructions and 10-12 cycles latency and looks something like this:

```
vspltisb v1,4
vspltisb v13,1
vsrb    v1,v2,v1
vslb    v0,v1,v13
vaddubm v0,v0,v1
vslb    v0,v0,v13
vsububm v2,v2,v0
```

The next step converts adjacent byte pairs to halfwords. We use the same basic formula but adjust the radix constants to; (rdx_hword - ((rdx_hword / 256) x (256 - 100)). Here we need a byte multiply producing a halfword correction factor. No shifts are needed as the vmuleub multiply will access the high byte of each halfword directly.

```
static inline vui16_t
vec_rdxct10kh (vui8_t vra)
{
  vui8_t c156;
  vui16_t x156;
  // Compute the high digit correction factor. For 100s to binary 10ks
  // this is the isolated high digit multiplied by the radix difference
  // in binary.  For this stage we use 256 - 100 = 156.
  c156 = vec_splats ((unsigned char) 156);
#if __BYTE_ORDER__ == __ORDER_LITTLE_ENDIAN__
  x156 = vec_mulo ((vui8_t) vra, c156);
#else
  x156 = vec_mule ((vui8_t) vra, c156);
#endif
  // Subtract the high digit correction halfword from the original
  // 100s byte pair in binary.  This reduces the range to 0-9999.
  return vec_sub ((vui16_t) vra, x156);
}
```

This requires: a constant load, a multiply even byte and subtract halfword. The final sequence runs 2-5 instructions and 9-18 cycles latency and looks something like this:

```
addis   r9,r2,.rodata.cst16+0x90@ha
addi    r9,r9,.rodata.cst16+0x90@l
lvx     v0,0,r9
vmuleub v0,v2,v0
vsubuhm v2,v2,v0
```

This pattern continues for converting halfwords to words, words to doublewords, and doublewords to quadwords. For POWER8 the first 4 steps are supported by vector multiply and subtract instructions. The last step requires a vec_vmuleud() operation implemented in vec_int128_ppc.h, based on vec_muleuw(), vec_mulouw() and vec_adduqm(). The vec_adduqm() operation is single instruction for POWER8. For POWER7 we will need to leverage more operations implemented in vec_int64_ppc.h and vec_int128_ppc.h for the last two steps.

The complete set of steps for converting 32 BCD digits to quadword __int128 binary looks like this:

```
vui128_t
example_vec_bcdctuq (vBCD_t vra)
{
  vui8_t d100;
  vui16_t d10k;
  vui32_t d100m;
  vui64_t d10e;
  vui128_t result;
  d100 = vec_rdxct100b ((vui8_t) vra);
  d10k = vec_rdxct10kh (d100);
  d100m = vec_rdxct100mw (d10k);
  d10e = vec_rdxct10E16d (d100m);
  result = vec_rdxct10e32q (d10e);
  return result;
}
```

For POWER8 the whole sequence runs 24-36 instructions and 65-78 cycles latency. For POWER9 the whole sequence runs 17-26 instructions and 52-65 cycles latency.

Note

> POWER9 has a Decimal Convert to Signed Quadword instruction, but no unsigned (32-digit) convert.

However we can leverage the POWER9 **Vector Multiply by 10 Extended Unsigned Quadword** instruction to extend the 31-digit convert to a full 32-digits. Basically use the **bcdctsq** to convert the high 31-digits and then multiply by 10 and add the last digit. See example below:

```
vui128_t
example_vec_bcdctuq_2 (vBCD_t vra)
{
  vui128_t vrt;
#ifdef  _ARCH_PWR9
  const vui32_t bcd_one = (vui32_t) _BCD_CONST_PLUS_ONE;
  const vui32_t sign_mask = (vui32_t) _BCD_CONST_SIGN_MASK;
  vui128_t vrd;
  vBCD_t sbcd;
  // Need to convert BCD unsigned to signed for bcdctsq
  // But can’t use bcdcpsgn as the unit digit is not a sign code
  // So use vec_and/sel to extract unit digit and insert sign
  vrd = (vui128_t) vec_and ((vui32_t) vra, sign_mask);
  sbcd = (vBCD_t) vec_sel ((vui32_t) vra, bcd_one, sign_mask);
  // Convert top 31 digits to binary
  vrt = (vui128_t) vec_bcdctsq (sbcd);
  // Then X 10 plus the unit digit to complete 32-digit convert
  vrt = vec_mul10euq (vrt, vrd);
#else
  // P7/P8 implementation as above
#endif
  return vrt;
}
```

This adds a few more cycles to split the high digits from the low digit and insert a positive sign code. This requires loading some vector constants which may be commoned with loads from other operations. This adds 2-11 cycles. The **mul10euq** only adds 3 cycles latency to complete the BCD to Binary conversion. This is adds only a 21% to 60% latency over the base **bcdctsq** instruction.

Note

> This process can be extended to 256, 512, 1024-bits, etc by widening the BCD to binary conversion appropriately to blocks of 31 or 32 digits. Then use the basic *atoi* algorithm using extended quadword multiply / add operations from vec_int128_ppc.h (Multiple precision BCD to/from Binary conversion).

7.1.3.2.3.3 Vector Parallel quadword to BCD conversion

Note

> Binary to BCD conversions are challenging a in a number of ways. First any conversion requires division by non powers of 2. Second, for the same element size binary representation holds more equivalent decimal digits then BCD. If the binary value is too large for the BCD target's element size, the results are often undefined. For example vec_bcdcfsq(). So it is important to constrain the magnitude of the binary to fit the BCD target before conversion. See Converting Vector __int128 values to BCD for details.

In most senses, binary to BCD is the reverse of BCD to binary. The radix number in the conversion formula exchange places and the conversion starts with the largest element size (quadword) and works it's way down to the smallest (4-bit nibble).

Let's take a look at the conversion formula. For BCD to Binary we used:

- bin_byte <- (bcd_byte - ((bcd_byte / 16) x (16 - 10))

- bin_byte <- (bcd_byte - ((bcd_byte >> 4) x 6)

So after swapping the conversion (to / from) radix constants we see:

- bcd_byte <-(bin_byte - ((bin_byte / 10) x (10 - 16))
- bcd_byte <-(bin_byte - ((bin_byte / 10) x (-6))
- bcd_byte <-(bin_byte + ((bin_byte / 10) x 6)

The effect is to divide vector elements of 4∗2N bits by 10∗∗N and return the quotient in the high half of the element (in 4∗N bits), and the remainder of this divide in the low half of the element (in 4∗N bits), Where N is a power of 2^n and *n* ranges from 0 to 4 (5 steps again).

Note

> So why doesn't PVECLIB provide these steps as operations. For example: divide a vector unsigned __int128 by 10^{16} and return the quotient in the high doubleword and the remainder in the low doubleword of a vector unsigned long? Because if the input quadword is not less than 10^{32} the result is undefined (the quotient will overflow).

This is good news and bad news. It is good that the correction subtract became a simple add. This allows the uses of multiply sum instruction (where PowerISA has such instructions for the element size). The bad news is that the radix divisor is not a power of two. And since the PowerISA does not have vector integer divide instructions, we use the multiplicative inverse. So in effect, each step of the binary to BCD conversion requires, two multiplies and an add.

So let's look at the first and last step of the conversion (the two extremes). The first step (after verifying that the quadword value is less than $10^{32<}$-1) looks like this:

```
static inline vui64_t
vec_rdxcf10e32q (vui128_t vra)
{
  // Compute the high digit correction factor. For binary 10**32 to
  // 10**16, this is  0x10000000000000000 - 10000000000000000
  // = 18436744073709551616.
  const vui64_t c = CONST_VINT128_DW (0, 18436744073709551616UL);
  // Magic numbers for multiplicative inverse to divide by 10**16
  // are 76624777043294442917917351357515459181, no corrective add,
  // and shift right 51 bits.
  const vui128_t mul_invs_ten16 = (vui128_t) CONST_VINT128_DW(
      0x39a5652fb1137856UL, 0xd30baf9a1e626a6dUL);
  const int shift_ten16 = 51;
  vui64_t result;
  vui128_t x, high_digit;
  // high_digit = vra / 10000000000000000;
  high_digit = vec_mulhuq (vra, mul_invs_ten16);
  high_digit = vec_srqi (high_digit, shift_ten16);
  // multiply high_digit by the radix difference c and add vra
  // This separates the high/low 16 digits into doublewords.
#ifdef _ARCH_PWR9
  // 0 in the high dword of const c reduces vmsumudm to vmuloud
  // but with a qword add included.
  result = (vui64_t) vec_msumudm ((vui64_t) high_digit, c, vra);
#else
  x = vec_vmuloud ((vui64_t) high_digit, c);
  result = (vui64_t) vec_adduqm (vra, x);
#endif
  return result;
}
```

The first multiply is an expensive (40 to 60 cycles) operation as it requires a full Multiply High Unsigned Quadword. The next operation requires a Multiply Odd Unsigned Doubleword then Add Unsigned Quadword Modulo. For POWER9 we can replace these two operations with a single Multiply Sum Unsigned Doubleword Modulo. The latency of this single step is in the same order at the complete BCD to Binary conversion (vec_bcdctuq()).

The conversion steps continue with doubleword to word, word to halfword, halfword to byte, byte to BCD (nibbles). The final step is simple by comparison to the first step.

```
static inline vui8_t
vec_rdxcf100b (vui8_t vra)
{
  vui8_t x6, c6, high_digit;
  // Let the compiler generate the multiplicative inverse code
  high_digit = vra / 10;
  // This separates two digit values into BCD Nibbles.
  // multiply high_digit by the radix difference c and
  x6 = high_digit * 6;
  // add bytes the high digit correction to the original
  // (radix 100) bytes in binary.
  return (vra + x6);
}
```

The GCC vector extensions support dividing a vector char / short / int by a constant. So we can let the compiler generate the multiplicative inverse code for the last three steps. This is not supported (yet) for long and __int128 so the first two steps must explicitly code the multiplicative inverse.

Using GCC vector extensions for the following multiply and add works well in this case as it allows the compiler to perform strength reduction. It is not as useful in the other steps as the programmer knows more about the value ranges then the compiler can or should assume. We know the the quotient and corrective constant always fit into the lower half of the element. This allows the use of the half sized vector multiply odd unsigned while compiler will assume it needs to generate a multiply modulo for the full element size.

For example the third step (word to halfword) we can use Multiply Sum Unsigned Halfword Modulo to replace the multiply odd and add. This is similar to the multiply sum usage in the first step and it is a case not recognized by the compiler.

The full binary to BCD conversion requires all 5 steps to complete the operations and this adds up to 200+ cycles. So this is worth another look.

Initially using the DFP Facility for this binary to BCD conversion was rejected because:

- The DFP Facility only supports signed fixed doubleword conversions (no fixed quadword conversion)
- Fixed binary to DFP conversions are expensive operations
  - For POWER8, 32 cycles latency and 1 per 19 cycles throughput
- The DFP Facility does support DFP to BCD conversions for double and quadword
  - For POWER8, 13 cycle latency and 1 per cycle throughput

Perhaps we can use the vec_rdxcf10e32q() operation we defined above as the first step (factoring quadwords into the 16 digit doublewords). Then use the DFP Facility to convert binary doublewords to BCD. In this case we are not concerned with signed conversion as 10**16 fits in 54-bits binary and guarantees positive binary values. We still have to deal with the VR to/from FPR transfers but that mechanism is already defined and at a reasonable cost (2-4 cycles each way).

```
static inline vBCD_t
vec_BIN2BCD (vui64_t val)
{
#ifdef _ARCH_PWR6
  vBCD_t t;
  _Decimal128 x, y, z;
  // unpack the vector into a FPRp
  z = vec_unpack_Decimal128 ((vf64_t) val);
  // Convert 2 long int values into 2 _Decimal64 values
  // Then convert each _Decimal64 value into 16-digit BCD
  __asm__(
      "dcffix %1,%2;\n"
      "dcffix %L1,%L2;\n"
      "ddedpd 0,%0,%1;\n"
      "ddedpd 0,%L0,%L1;\n"
      : "=d" (x),
      "=&d" (y)
      : "d" (z)
      : );
  // Pack the FPRp back into a vector
  t = (vBCD_t) vec_pack_Decimal128 (x);
  return (t);
#else
  // no solution before P6
#endif
}
```

If we assume that the second Decimal Convert From Fixed (dcffix) is independent and issues 19 cycles after the first, we get 32+19 = 51 cycles to complete. Then another 13+1 cycles to convert back to BCD. Add a few cycles for the unpack and pack operations and we estimate 69 cycles for POWER8 and 58 cycles for POWER9. The totals for vec_rdxcf10e32q() plus vec_BIN2BCD() come to 154-164 for POWER8 and 114-124 for POWER9. This is a 30-60% improvement over the previous (all vector) attempt. So the final unsigned binary to BCD conversion looks like this:

```
static inline vBCD_t
vec_bcdcfuq (vui128_t vra)
{
  vui64_t d10e;
  d10e =vec_rdxcf10e32q (vra);
#ifdef _ARCH_PWR7
  return (vBCD_t) vec_BIN2BCD (d10e);
#else
  vui8_t d100;
  vui16_t d10k;
  vui32_t d100m;
  d100m = vec_rdxcf10E16d (d10e);
  d10k = vec_rdxcf100mw (d100m);
  d100 = vec_rdxcf10kh (d10k);
  return (vBCD_t) vec_rdxcf100b (d10e);
#endif
}
```

Note

> This process can be extended to 256, 512, 1024-bits, etc by widening the first 5 steps appropriately and adding steps using extended quadword multiply and add operations from vec_int128_ppc.h (Quadword Long Division).

7.1.3.2.4 Multiple precision BCD to/from Binary conversion

The simplest case is converting a vector unsigned __int128 to BCD. This requires up to 39 digits across two vectors. This can either be split into 8 and 31 digits for signed conversion or 7 and 32 for unsigned. Signed conversion is preferred where extended BCD result will be input to additional BCD arithmetic. Unsigned is preferred for conversion to Zoned characters for decimal display.

From Converting Vector __int128 values to BCD we see the divide / modulo quadword by constant operations which can used to factor binary quadwords into high and low digit groups for conversion. For example:

```
q = vec_divuq_10e32 (a);
r = vec_moduq_10e32 (a, q);
// high 7 digits
dh = vec_bcdcfuq (q);
// lower 32 digits
dl = vec_bcdcfuq (r);
printf ("%07lld%016lld%016lld", (vui64_t) dh[VEC_DW_L],
        (vui64_t) dl[VEC_DW_H], (vui64_t) dl[VEC_DW_L]);
```

7.1.3.2.4.1 Multiple precision BCD from Binary conversion

The general multiple precision binary to BCD conversion requires quadword long division as described in Quadword Long Division. After each long division the remainder is in a range for conversion to BCD. In the example below the remainder is converted to 32 digit BCD as the last step.

```
// Convert extended quadword binary to BCD 32-digits at a time.
vBCD_t
example_longbcdcf_10e32 (vui128_t *q, vui128_t *d, long int _N)
{
  vui128_t dn, qh, ql, rh;
  long int i;
  // init step for the top digits
  dn = d[0];
  qh = vec_divuq_10e32 (dn);
  rh = vec_moduq_10e32 (dn, qh);
  q[0] = qh;
  // now we know the remainder is less than the divisor.
  for (i=1; i<_N; i++)
    {
      dn = d[i];
      ql = vec_divudq_10e32 (&qh, rh, dn);
      rh = vec_modudq_10e32 (rh, dn, &ql);
      q[i] = ql;
    }
  // convert to BCD and return the remainder for this step
  return vec_bcdcfuq (rh);
}
```

Each call to example_longbcdcf_10e32 () produces the next 32-digit group. Repeated calls where the previous iterations quotient is passed as the dividend to the next step, produce additional 32-digit groups. This continues until the quotient is less then the divisor (in this case 10^{32}). This final quadword quotient provides the highest order 32-digit group for the conversion. The digit groups are produced in order from lowest to highest significance.

As the conversion process continues the number of quadwords in the extended dividend/quotient shrinks. The divide / modulo quadword by constant operations test for leading zeros and skip over them.

7.1.3.2.4.2 Multiple precision BCD to Binary conversion

The general multiple precision binary from BCD conversion only requires extended quadword multiply as described in Extended Quadword multiply. Starting with the high order BCD (32 or 31) digit group, multiply by 10^{32} (or 10^{31}) then add the next digit group to the extended product. Continue until the low order digit group is added. For example:

```
// Convert extended quadword BCD to binary 32-digits at a time.
long int
example_longbcdct_10e32 (vui128_t *d, vBCD_t decimal,
                         long int _C , long int _N)
{
  // ten32  = +100000000000000000000000000000000UQ
  const vui128_t ten32 = (vui128_t)
          { (__int128) 10000000000000000UL * (__int128) 10000000000000000UL };
  const vui128_t zero = (vui128_t) { (__int128) 0UL };
  vui128_t dn, ph, pl, cn, c;
  long int i, cnt;
  cnt = _C;
  dn = zero;
  cn  = zero;
  // case _C == 0 is the initialization step and no multiply required
  if ( cnt == 0 )
    {
      // if the decimal is 0, no conversion is required
      if (vec_cmpuq_all_ne ((vui128_t) decimal, zero))
        {
          cnt++;
          dn = vec_bcdctuq (decimal);
        }
      // But it is a good time to initialize d[]
      for ( i = 0; i < (_N - 1); i++ )
        {
          d[i] = zero;
        }
      d[_N - cnt] = dn;
    }
  else // case _C > 0
    {
      // convert decimal group to binary.
      if (vec_cmpuq_all_ne ((vui128_t) decimal, zero))
        {
          dn = vec_bcdctuq (decimal);
        }
      // Compute extended product, plus the decimal group
      for ( i = (_N - 1); i >= (_N - cnt); i--)
        {
          pl = vec_muludq (&ph, d[i], ten32);
          c = vec_addecuq (pl, dn, cn);
          d[i] = vec_addeuqm (pl, dn, cn);
          cn = c;
          dn = ph;
        }
      // If the product exceeds the current quadword count, extend
      if (vec_cmpuq_all_ne (dn, zero) || vec_cmpuq_all_ne (cn, zero))
        {
          cnt++;
          dn = vec_adduqm (dn, cn);
          d[_N - cnt] = dn;
        }
    }
  return cnt;
}
```

This process starts with a single quadword (the converted high order digit group). As additional digit groups are converted, the extended binary value is multiplied by 10^{32} before adding the converted digit group. The number of quadwords in the array *d[]* expand as needed to hold the binary value.

The interface includes:

- A pointer to an array of quadwords which accumulates the converted binary value.
- A BCD decimal value to be converted and added to the accumulated binary.
- A current quadword count. The number of nonzero quadwords accumulated so far. Should be 0 on the initial call.
- A maximum quadword count.
- Return the updated quadword count. Passed back as current quadword count on the next iteration.

7.1.4 Performance data.

High level performance estimates are provided as an aid to function selection when evaluating algorithms. For background on how *Latency* and *Throughput* are derived see: Performance data.

7.1.5 Macro Definition Documentation

7.1.5.1 vBCD_t

```
#define vBCD_t vui32_t
```

vector signed BCD integer of up to 31 decimal digits.

Note

> Currently the GCC implementation and the `OpenPOWER ELF V2 ABI` disagree on the vector type (vector __int128 vs vector unsigned char) used (parameters and return values) by the BCD built-ins. Using vBCD_t insulates **pveclib** and applications while this is worked out.

7.1.6 Function Documentation

7.1.6.1 vec_BCD2BIN()

```
static vui64_t vec_BCD2BIN (
            vBCD_t val )  [inline], [static]
```

Convert vector of 2 x unsigned 16-digit BCD values to vector 2 x doubleword binary values.

Convert a vector of 16-digit unsigned BCD doublewords to a vector of unsigned long int doublewords. The vector unsigned long int doublewords are in the range 0-9999999999999999.

processor	Latency	Throughput
power8	55	1/51 cycle
power9	59	1/53 cycle

Parameters

val	a vector treated a 2 unsigned BCD 16 digit values.

Returns

a 128-bit vector unsigned long int.

7.1.6.2 vec_BCD2DFP()

```
static _Decimal128 vec_BCD2DFP (
             vBCD_t val )  [inline], [static]
```

Convert a Vector Signed BCD value to __Decimal128.

The BCD vector is permuted into a double float pair before conversion to DPD format via the DFP Encode BCD To DPD Quad instruction.

processor	Latency	Throughput
power8	17	1/cycle
power9	15	1/cycle

Parameters

val	a 128-bit vector treated as a signed BCD 31 digit value.

Returns

a __Decimal128 in a double float pair.

7.1.6.3 vec_bcdadd()

```
static vBCD_t vec_bcdadd (
             vBCD_t a,
             vBCD_t b )  [inline], [static]
```

Decimal Add Signed Modulo Quadword.

Two Signed 31 digit values are added and lower 31 digits of the sum are returned. Overflow (carry-out) is ignored.

processor	Latency	Throughput
power8	13	1/cycle
power9	3	2/cycle

Parameters

a	a 128-bit vector treated as a signed BCD 31 digit value.
b	a 128-bit vector treated as a signed BCD 31 digit value.

Returns

a 128-bit vector which is the lower 31 digits of (a + b).

7.1.6.4 vec_bcdaddcsq()

```
static vBCD_t vec_bcdaddcsq (
            vBCD_t a,
            vBCD_t b )  [inline], [static]
```

Decimal Add & write Carry Signed Quadword.

Two Signed 31 digit BCD values are added, and the carry-out (the high order 32nd digit) of the sum is returned.

Note

This operation will only detect overflows where the operand signs match. It will not detect a borrow if the signs differ. So this operation should only be used if matching signs are guaranteed. Otherwise vec_cbcdaddcsq() should be used.

processor	Latency	Throughput
power8	15-21	1/cycle
power9	6-18	2/cycle

Parameters

a	a 128-bit vector treated as a signed BCD 31 digit value.
b	a 128-bit vector treated as a signed BCD 31 digit value.

Returns

a 128-bit vector with the carry digit. Values are -1, 0, and +1.

7.1.6.5 vec_bcdaddecsq()

```
static vBCD_t vec_bcdaddecsq (
            vBCD_t a,
            vBCD_t b,
            vBCD_t c )  [inline], [static]
```

Decimal Add Extended & write Carry Signed Quadword.

Two Signed 31 digit values and a signed carry-in are added together and the carry-out (the high order 32nd digit) of the sum is returned.

Note

This operation will only detect overflows where the operand signs match. It will not detect a borrow if the signs differ. So this operation should only be used if matching signs are guaranteed. Otherwise vec_cbcdaddecsq() should be used.

processor	Latency	Throughput
power8	28-37	1/cycle
power9	9-21	2/cycle

Parameters

a	a 128-bit vector treated as a signed BCD 31 digit value.
b	a 128-bit vector treated as a signed BCD 31 digit value.
c	a 128-bit vector treated as a signed BCD carry with values -1, 0, or +1.

Returns

a 128-bit vector with the carry digit from the sum (a + b +c). Carry values are -1, 0, and +1.

7.1.6.6 vec_bcdaddesqm()

```
static vBCD_t vec_bcdaddesqm (
            vBCD_t a,
            vBCD_t b,
            vBCD_t c )  [inline], [static]
```

Decimal Add Extended Signed Modulo Quadword.

Two Signed 31 digit values and a signed carry-in are added together and lower 31 digits of the sum are returned. Overflow (carry-out) is ignored.

processor	Latency	Throughput
power8	13	1/cycle
power9	6	2/cycle

Parameters

a	a 128-bit vector treated as a signed BCD 31 digit value.
b	a 128-bit vector treated as a signed BCD 31 digit value.
c	a 128-bit vector treated as a signed BCD carry with values -1, 0, or +1.

Returns

a 128-bit vector which is the lower 31 digits of (a + b + c).

7.1.6.7 vec_bcdcfsq()

```
static vBCD_t vec_bcdcfsq (
            vi128_t vrb )  [inline], [static]
```

Vector Decimal Convert From Signed Quadword returning up to 31 BCD digits.

Vector convert a quadword containing a signed __int128 in the range -9999999999999999999999999999999 to +9999999999999999999999999999999 to the equivalent signed BCD value with up to 31 digits.

Note

If the signed value of vrb is less then -(10∗∗31-1) or greater than 10∗∗31-1 the result is too large for the BCD format and the result is undefined. See Converting Vector __int128 values to BCD for details.

processor	Latency	Throughput
power8	166-176	1/19 cycle
power9	37	1/26 cycle

Parameters

vrb	vector signed __int128 number in the range -9999999999999999999999999999999 to +9999999999999999999999999999999.

Returns

vector signed BCD value.

7.1.6.8 vec_bcdcfud()

```
static vBCD_t vec_bcdcfud (
            vui64_t vrb )  [inline], [static]
```

Vector Decimal Convert From Unsigned doubleword returning up to 2x16 BCD digits.

Vector convert doubleword containing a unsigned long int each in the range 0-9999999999999999 to the equivalent unsigned BCD doubleword value each up to 16 digits.

Note

If either doubleword of vrb is greater than 10∗∗16-1 the result is too large for the BCD format and the result is undefined.

processor	Latency	Throughput
power8	69	1/19 cycle
power9	58	1/21 cycle

Parameters

vrb	a 128-bit vector of unsigned long int numbers, each in the range 0-9999999999999999.

Returns

128-bit vector doublewords of unsigned BCD values each in the range 0-9999999999999999.

7.1.6.9 vec_bcdcfuq()

```
static vBCD_t vec_bcdcfuq (
            vui128_t vra )  [inline], [static]
```

Vector Decimal Convert From Unsigned Quadword returning up to 32 BCD digits.

Vector convert a quadword containing a unsigned __int128 in the range 0-99999999999999999999999999999999 to the equivalent unsigned BCD value with up to 32 digits.

Note

If the value of vrb is greater than 10∗∗32-1 the result is too large for the unsigned BCD format and the result is undefined. See Converting Vector __int128 values to BCD for details.

processor	Latency	Throughput
power8	154-164	1/19 cycle
power9	117-128	1/21 cycle

Parameters

vra	a 128-bit vector as an unsigned __int128 number in the range 0-9999999999999999999999999999999.

Returns

128-bit vector unsigned BCD value in the range 0-9999999999999999999999999999999.

7.1.6.10 vec_bcdcfz()

```
static vBCD_t vec_bcdcfz (
             vui8_t vrb )  [inline], [static]
```

Vector Decimal Convert From Zoned.

Given a Signed 16-digit signed Zoned value vrb, return equivalent Signed BCD value. For Zoned (PS=0) the sign code is in bits 0:3 of byte 15.

- Positive sign codes are: 0x0, 0x1, 0x2, 0x3, 0x8, 0x9, 0xa, 0xb.
- Negative sign codes are: 0x4, 0x5, 0x6, 0x7, 0xc, 0xd, 0xe, 0xf.

The resulting BCD value with up to 16 digits magnitude and set to the preferred BCD sign (0xc or 0xd).

Note

The POWER9 bcdcfz instruction gives undefined results if given invalid input. In this implementation for older processors there is no checking for Zone (bits 0:3) or digit (bits 4:7) range.

processor	Latency	Throughput
power8	14-27	1/cycle
power9	3	2/cycle

Parameters

vrb	a 128-bit vector treated as a signed Zoned 16 digit value.

Returns

a 128-bit BCD value with the magnitude and sign from the Zoned value in vrb.

7.1.6.11 vec_bcdcmp_eqsq()

```
static vbBCD_t vec_bcdcmp_eqsq (
            vBCD_t vra,
            vBCD_t vrb ) [inline], [static]
```

Vector Compare Signed BCD Quadword for equal.

Compare vector signed BCD values and return vector bool true if vra and vrb are equal.

processor	Latency	Throughput
power8	15-17	1/cycle
power9	6-9	2/cycle

Parameters

vra	128-bit vector treated as an vector signed BCD (qword) element.
vrb	128-bit vector treated as an vector signed BCD (qword) element.

Returns

128-bit vector boolean reflecting vector signed BCD compare equal.

7.1.6.12 vec_bcdcmp_gesq()

```
static vbBCD_t vec_bcdcmp_gesq (
            vBCD_t vra,
            vBCD_t vrb ) [inline], [static]
```

Vector Compare Signed BCD Quadword for greater than or equal.

Compare vector signed BCD values and return vector bool true if vra and vrb are greater than or equal.

processor	Latency	Throughput
power8	15-17	1/cycle
power9	6-9	2/cycle

Parameters

vra	128-bit vector treated as an vector signed BCD (qword) element.
vrb	128-bit vector treated as an vector signed BCD (qword) element.

Returns

128-bit vector boolean reflecting vector signed BCD compare greater than or equal.

7.1.6.13 vec_bcdcmp_gtsq()

```
static vbBCD_t vec_bcdcmp_gtsq (
            vBCD_t vra,
            vBCD_t vrb )  [inline], [static]
```

Vector Compare Signed BCD Quadword for greater than.

Compare vector signed BCD values and return vector bool true if vra and vrb are greater than.

processor	Latency	Throughput
power8	15-17	1/cycle
power9	6-9	2/cycle

Parameters

vra	128-bit vector treated as an vector signed BCD (qword) element.
vrb	128-bit vector treated as an vector signed BCD (qword) element.

Returns

128-bit vector boolean reflecting vector signed BCD compare greater than.

7.1.6.14 vec_bcdcmp_lesq()

```
static vbBCD_t vec_bcdcmp_lesq (
            vBCD_t vra,
            vBCD_t vrb )  [inline], [static]
```

Vector Compare Signed BCD Quadword for less than or equal.

Compare vector signed BCD values and return vector bool true if vra and vrb are less than or equal.

processor	Latency	Throughput
power8	15-17	1/cycle
power9	6-9	2/cycle

Parameters

vra	128-bit vector treated as an vector signed BCD (qword) element.
vrb	128-bit vector treated as an vector signed BCD (qword) element.

Returns

128-bit vector boolean reflecting vector signed BCD compare less than or equal.

7.1.6.15 vec_bcdcmp_ltsq()

```
static vbBCD_t vec_bcdcmp_ltsq (
            vBCD_t vra,
            vBCD_t vrb )  [inline], [static]
```

Vector Compare Signed BCD Quadword for less than.

Compare vector signed BCD values and return vector bool true if vra and vrb are less than.

processor	Latency	Throughput
power8	15-17	1/cycle
power9	6-9	2/cycle

Parameters

vra	128-bit vector treated as an vector signed BCD (qword) element.
vrb	128-bit vector treated as an vector signed BCD (qword) element.

Returns

128-bit vector boolean reflecting vector signed BCD compare less than.

7.1.6.16 vec_bcdcmp_nesq()

```
static vbBCD_t vec_bcdcmp_nesq (
            vBCD_t vra,
            vBCD_t vrb )  [inline], [static]
```

Vector Compare Signed BCD Quadword for not equal.

Compare vector signed BCD values and return vector bool true if vra and vrb are not equal.

processor	Latency	Throughput
power8	15-17	1/cycle
power9	6-9	2/cycle

Parameters

vra	128-bit vector treated as an vector signed BCD (qword) element.
vrb	128-bit vector treated as an vector signed BCD (qword) element.

Returns

128-bit vector boolean reflecting vector signed BCD compare not equal.

7.1.6.17 vec_bcdcmpeq()

```
static int vec_bcdcmpeq (
            vBCD_t vra,
            vBCD_t vrb )  [inline], [static]
```

Vector Compare Signed BCD Quadword for equal.

Compare vector signed BCD values and return boolean true if vra and vrb are equal.

processor	Latency	Throughput
power8	13	1/cycle
power9	3	2/cycle

Parameters

vra	128-bit vector treated as an vector signed BCD (qword) element.
vrb	128-bit vector treated as an vector signed BCD (qword) element.

Returns

boolean int for BCD compare, true if equal, false otherwise.

7.1.6.18 vec_bcdcmpge()

```
static int vec_bcdcmpge (
            vBCD_t vra,
            vBCD_t vrb )  [inline], [static]
```

Vector Compare Signed BCD Quadword for greater than or equal.

Compare vector signed BCD values and return boolean true if vra and vrb are greater than or equal.

processor	Latency	Throughput
power8	13	1/cycle
power9	3	2/cycle

Parameters

vra	128-bit vector treated as an vector signed BCD (qword) element.
vrb	128-bit vector treated as an vector signed BCD (qword) element.

Returns

boolean int for BCD compare, true if greater than or equal, false otherwise.

7.1.6.19 vec_bcdcmpgt()

```
static int vec_bcdcmpgt (
            vBCD_t vra,
            vBCD_t vrb )  [inline], [static]
```

Vector Compare Signed BCD Quadword for greater than.

Compare vector signed BCD values and return boolean true if vra and vrb are greater than.

processor	Latency	Throughput
power8	13	1/cycle
power9	3	2/cycle

Parameters

vra	128-bit vector treated as an vector signed BCD (qword) element.
vrb	128-bit vector treated as an vector signed BCD (qword) element.

Returns

boolean int for BCD compare, true if greater than, false otherwise.

7.1.6.20 vec_bcdcmple()

```
static int vec_bcdcmple (
            vBCD_t vra,
            vBCD_t vrb )  [inline], [static]
```

Vector Compare Signed BCD Quadword for less than or equal.

Compare vector signed BCD values and return boolean true if vra and vrb are less than or equal.

processor	Latency	Throughput
power8	13	1/cycle
power9	3	2/cycle

Parameters

vra	128-bit vector treated as an vector signed BCD (qword) element.
vrb	128-bit vector treated as an vector signed BCD (qword) element.

Returns

boolean int for BCD compare, true if less than or equal, false otherwise.

7.1.6.21 vec_bcdcmplt()

```
static int vec_bcdcmplt (
            vBCD_t vra,
            vBCD_t vrb )  [inline], [static]
```

Vector Compare Signed BCD Quadword for less than.

Compare vector signed BCD values and return boolean true if vra and vrb are less than.

processor	Latency	Throughput
power8	13	1/cycle
power9	3	2/cycle

Parameters

vra	128-bit vector treated as an vector signed BCD (qword) element.
vrb	128-bit vector treated as an vector signed BCD (qword) element.

Returns

boolean int for BCD compare, true if less than, false otherwise.

7.1.6.22 vec_bcdcmpne()

```
static int vec_bcdcmpne (
             vBCD_t vra,
             vBCD_t vrb )  [inline], [static]
```

Vector Compare Signed BCD Quadword for not equal.

Compare vector signed BCD values and return boolean true if vra and vrb are not equal.

processor	Latency	Throughput
power8	13	1/cycle
power9	3	2/cycle

Parameters

vra	128-bit vector treated as an vector signed BCD (qword) element.
vrb	128-bit vector treated as an vector signed BCD (qword) element.

Returns

boolean int for BCD compare, true if not equal, false otherwise.

7.1.6.23 vec_bcdcpsgn()

```
static vBCD_t vec_bcdcpsgn (
             vBCD_t vra,
             vBCD_t vrb )  [inline], [static]
```

Vector copy sign BCD.

Given Two Signed BCD 31 digit values vra and vrb, return the magnitude from vra (bits 0:123) and the sign (bits 124:127) from vrb.

processor	Latency	Throughput
power8	2-11	1/cycle
power9	3	2/cycle

Parameters

vra	a 128-bit vector treated as a signed BCD 31 digit value.
vrb	a 128-bit vector treated as a signed BCD 31 digit value.

Returns

a 128-bit BCD value with the magnitude from vra and the sign copied from vrb.

7.1.6.24 vec_bcdctsq()

```
static vi128_t vec_bcdctsq (
            vBCD_t vra )  [inline], [static]
```

Vector Decimal Convert to Signed Quadword.

Vector convert a BCD quadword containing signed 31 digits values to signed __int128, in the range +/- 0-9999999999999999999999999999999.

processor	Latency	Throughput
power8	80-95	1/cycle
power9	23	1/12 cycle

Parameters

vra	a 128-bit vector treated as a signed 31-digit BCD number.

Returns

128-bit vector signed __int128 in the range +/- 0-9999999999999999999999999999999.

7.1.6.25 vec_bcdctub()

```
static vui8_t vec_bcdctub (
            vBCD_t vra )  [inline], [static]
```

Vector Decimal Convert Binary Coded Decimal (BCD) digit pairs to binary unsigned bytes .

Vector convert 16 bytes each containing 2 BCD digits to the equivalent unsigned char, in the range 0-99. Input values should be valid 2 x BCD nibbles in the range 0-9.

processor	Latency	Throughput
power8	13-22	1/cycle
power9	14-23	1/cycle

Parameters

vra	a 128-bit vector treated as 16 unsigned 2-digit BCD numbers.

Returns

128-bit vector unsigned char, For each byte in the range 0-99.

7.1.6.26 vec_bcdctud()

```
static vui64_t vec_bcdctud (
            vBCD_t vra )  [inline], [static]
```

Vector Decimal Convert groups of 16 BCD digits to binary unsigned doublewords.

Vector convert 2 doublewords each containing 16 BCD digits to the equivalent unsigned long int, in the range 0-9999999999999999. Input values should be valid 16 x BCD nibbles in the range 0-9.

processor	Latency	Throughput
power8	40-51	1/cycle
power9	41-52	1/cycle

Parameters

vra	a 128-bit vector treated as 2 unsigned 16-digit BCD numbers.

Returns

128-bit vector unsigned long int, For each doubleword in the range 0-9999999999999999.

7.1.6.27 vec_bcdctuh()

```
static vui16_t vec_bcdctuh (
            vBCD_t vra )  [inline], [static]
```

Vector Decimal Convert groups of 4 BCD digits to binary unsigned halfwords.

Vector convert 8 halfwords each containing 4 BCD digits to the equivalent unsigned short, in the range 0-9999. Input values should be valid 4 x BCD nibbles in the range 0-9.

processor	Latency	Throughput
power8	22-31	1/cycle
power9	23-32	1/cycle

Parameters

vra	a 128-bit vector treated as 8 unsigned 4-digit BCD numbers.

Returns

128-bit vector unsigned short, For each halfword in the range 0-9999.

7.1.6.28 vec_bcdctuq()

```
static vui128_t vec_bcdctuq (
             vBCD_t vra )  [inline], [static]
```

Vector Decimal Convert groups of 32 BCD digits to binary unsigned quadword.

Vector convert a quadword containing 32 BCD digits to the equivalent unsigned __int128, in the range 0-99999999999999999999999999999999. Input values should be valid 32 x BCD nibbles in the range 0-9.

processor	Latency	Throughput
power8	65-78	1/cycle
power9	28-37	1/12 cycle

Parameters

vra	a 128-bit vector treated as an unsigned 32-digit BCD number.

Returns

128-bit vector unsigned __int128 in the range 0-99999999999999999999999999999999.

7.1.6.29 vec_bcdctuw()

```
static vui32_t vec_bcdctuw (
             vBCD_t vra )  [inline], [static]
```

Vector Decimal Convert groups of 8 BCD digits to binary unsigned words.

Vector convert 4 words each containing 8 BCD digits to the equivalent unsigned int, in the range 0-99999999. Input values should be valid 8 x BCD nibbles in the range 0-9.

processor	Latency	Throughput
power8	31-42	1/cycle
power9	32-43	1/cycle

Parameters

vra	a 128-bit vector treated as 4 unsigned 8-digit BCD numbers.

Returns

128-bit vector unsigned int, For each word in the range 0-99999999.

7.1.6.30 vec_bcdctz()

```
static vui8_t vec_bcdctz (
            vBCD_t vrb )  [inline], [static]
```

Vector Decimal Convert To Zoned.

Given a Signed 16-digit signed BCD value vrb, return equivalent Signed Zoned value. For Zoned (PS=0) the sign code is in bits 0:3 of byte 15.

- Positive sign Zone is: 0x30.
- Negative sign Zone is: 0x70.

The resulting Zone value will up to 16 digits magnitude and set to the preferred Zoned sign codes (0x30 or 0x70).

Note

The POWER9 bcdctz instruction gives undefined results if given invalid input. In this implementation for older processors there is no checking for BCD digit (bits 4:7) range.

processor	Latency	Throughput
power8	14-27	1/cycle
power9	3	2/cycle

Parameters

vrb	a 128-bit vector treated as a signed BCD 16 digit value.

Returns

a 128-bit Zoned value with the magnitude (low order 16-digits) and sign from the value in vrb.

7.1.6.31 vec_bcddiv()

```
static vBCD_t vec_bcddiv (
              vBCD_t a,
              vBCD_t b )  [inline], [static]
```

Divide a Vector Signed BCD 31 digit value by another BCD value.

One Signed 31 digit value is divided by a second 31 digit value and the quotient is returned.

processor	Latency	Throughput
power8	102-238	1/cycle
power9	96-228	1/cycle

Parameters

a	a 128-bit vector treated as a signed BCD 31 digit value.
b	a 128-bit vector treated as a signed BCD 31 digit value.

Returns

a 128-bit vector which is the lower 31 digits of (a / b).

7.1.6.32 vec_bcddive()

```
static vBCD_t vec_bcddive (
              vBCD_t a,
              vBCD_t b )  [inline], [static]
```

Decimal Divide Extended.

The dividend *a* is a Signed BCD 31 digit value extended to right internally with 31 decimal 0s. The divisor *b* is Signed BCD 31 digit value. The quotient of *a* || 0^{31} / *b* is truncated to a Decimal integer and returned in Signed BCD format.

processor	Latency	Throughput
power8	102-238	1/cycle
power9	96-228	1/cycle

Parameters

a	a 128-bit vector treated as the high 31-digits of a 62-digit value extended with 0's.
b	a 128-bit vector treated as a signed BCD 31 digit value.

Returns

a 128-bit vector quotient of (a / b).

7.1.6.33 vec_bcdmul()

```
static vBCD_t vec_bcdmul (
            vBCD_t a,
            vBCD_t b )  [inline], [static]
```

Multiply two Vector Signed BCD 31 digit values.

Two Signed 31 digit values are multiplied and the lower 31 digits of the product are returned. Overflow is ignored.

The vector unit does not have a BCD multiply, so we convert the operands to _Decimal128 format and use the DFP quadword multiply. This gets tricky as the product can be up to 62 digits, and _Decimal128 format can only hold 34 digits.

To avoid overflow in the DFP Facility, we split each BCD operand into 15 upper and 16 lower digit halves. This requires up to four decimal multiplies and produces up to four 30-32 digit partial products. These are aligned appropriately (via DFP decimal shift) and summed (via DFP Decimal add) to generate the high and low (31-digit) parts of the 62 digit product.

In this case we only need the lower 31-digits of the product. So only 3 (not 4) DFP multiplies are required. Also we can discard any high digits above 31.

Note

There is early exit case if both operands are 16-digits or less. Here the product can not exceed 32-digits and requires only a single DFP multiply. The DFP2BCD conversion will discard any extra (32th) digit.

processor	Latency	Throughput
power8	94-194	1/cycle
power9	88-227	1/cycle

Parameters

a	a 128-bit vector treated as a signed BCD 31 digit value.
b	a 128-bit vector treated as a signed BCD 31 digit value.

Returns

a 128-bit vector which is the lower 31 digits of (a ∗ b).

7.1.6.34 vec_bcdmulh()

```
static vBCD_t vec_bcdmulh (
            vBCD_t a,
            vBCD_t b )  [inline], [static]
```

Vector Signed BCD Multiply High.

Two Signed 31 digit values are multiplied and the higher 31 digits of the product are returned.

The vector unit does not have a BCD multiply, so we convert the operands to _Decimal128 format and use the DFP quadword multiply. This gets tricky as the product can be up to 62 digits, and _Decimal128 format can only hold 34 digits.

To avoid overflow in the DFP Facility, we split each BCD operand into 15 upper and 16 lower digit halves. This requires up four decimal multiplies and produces four 30-32 digit partial products. These are aligned appropriately (via DFP decimal shift) and summed (via DFP Decimal add) to generate the high and low (31-digit) parts of the 62 digit product.

In this case we only need the upper 31-digits of the product. The lower 31-digits are discarded.

Note

There is early exit case if both operands are 16-digits or less. Here the product can not exceed 32-digits and requires only a single DFP multiply. We use the DFP Decimal shift to discard the lower 31-digits and return the single (32nd) digit.

processor	Latency	Throughput
power8	106-361	1/cycle
power9	99-271	1/cycle

Parameters

a	a 128-bit vector treated as a signed BCD 31 digit value.
b	a 128-bit vector treated as a signed BCD 31 digit value.

Returns

a 128-bit vector which is the higher 31 digits of (a ∗ b).

7.1.6.35 vec_bcds()

```
static vBCD_t vec_bcds (
             vBCD_t vra,
             vi8_t vrb )  [inline], [static]
```

Decimal Shift. Shift a vector signed BCD value, left or right a variable amount of digits (nibbles). The sign nibble is preserved.

processor	Latency	Throughput
power8	14-25	1/cycle
power9	3	2/cycle

Parameters

vra	128-bit vector treated as a signed BCD 31 digit value.
vrb	Digit shift count in vector byte 7.

Returns

a 128-bit vector BCD value shifted right digits.

7.1.6.36 vec_bcdsetsgn()

```
static vBCD_t vec_bcdsetsgn (
             vBCD_t vrb )  [inline], [static]
```

Vector Set preferred BCD Sign.

Given a Signed BCD 31 digit value vrb, return the magnitude from vrb (bits 0:123) and the sign (bits 124:127) set to the preferred sign (0xc or 0xd). Valid positive sign codes are; 0xA, 0xC, 0xE, or 0xF. Valid negative sign codes are; 0xB or 0xD.

Note

The POWER9 bcdsetsgn instruction gives undefined results if given invalid input. In this implementation for older processors only the sign code is checked. In this case, if the sign code is invalid the vrb input value is returned unchanged.

processor	Latency	Throughput
power8	6-26	1/cycle
power9	3	2/cycle

Parameters

vrb	a 128-bit vector treated as a signed BCD 31 digit value.

Returns

a 128-bit BCD value with the magnitude from vra and the sign copied from vrb.

7.1.6.37 vec_bcdslqi()

```
static vBCD_t vec_bcdslqi (
            vBCD_t vra,
            const unsigned int _N )  [inline], [static]
```

Vector BCD Shift Right Signed Quadword.

Shift a vector signed BCD value right _N digits.

processor	Latency	Throughput
power8	6-15	2/cycle
power9	3-6	2/cycle

Parameters

vra	128-bit vector signed BCD 31 digit value.
↩ _↩ *N*	int constant for the number of digits to shift right.

Returns

a 128-bit vector BCD value shifted right _N digits.

7.1.6.38 vec_bcdsluqi()

```
static vBCD_t vec_bcdsluqi (
            vBCD_t vra,
            const unsigned int _N )  [inline], [static]
```

Vector BCD Shift Right unsigned Quadword.

Shift a vector unsigned BCD value right _N digits.

processor	Latency	Throughput
power8	6-15	2/cycle
power9	3-6	2/cycle

Parameters

vra	128-bit vector unsigned BCD 32 digit value.
↩ _↩ *N*	int constant for the number of digits to shift right.

Returns

a 128-bit vector BCD value shifted right _N digits.

7.1.6.39 vec_bcdsr()

```
static vBCD_t vec_bcdsr (
              vBCD_t vra,
              vi8_t vrb )  [inline], [static]
```

Decimal Shift and Round. Shift a vector signed BCD value, left or right a variable amount of digits (nibbles). The sign nibble is preserved. If byte element 7 of the shift count is negative (right shift), and the last digit shifted out is greater then or equal to 5, then increment the shifted magnitude by 1.

processor	Latency	Throughput
power8	14-25	1/cycle
power9	3	2/cycle

Parameters

vra	128-bit vector treated as a signed BCD 31 digit value.
vrb	Digit shift count in vector byte 7.

Returns

a 128-bit vector BCD value shifted right digits.

7.1.6.40 vec_bcdsrqi()

```
static vBCD_t vec_bcdsrqi (
            vBCD_t vra,
            const unsigned int _N )  [inline], [static]
```

Vector BCD Shift Right Signed Quadword Immediate.

Shift a vector signed BCD value right _N digits.

processor	Latency	Throughput
power8	6-15	2/cycle
power9	3-6	2/cycle

Parameters

vra	128-bit vector signed BCD 31 digit value.
↩ _↩ *N*	int constant for the number of digits to shift right.

Returns

a 128-bit vector BCD value shifted right _N digits.

7.1.6.41 vec_bcdsrrqi()

```
static vBCD_t vec_bcdsrrqi (
            vBCD_t vra,
            const unsigned int _N )  [inline], [static]
```

Vector BCD Shift Right and Round Signed Quadword Immediate.

Shift and round a vector signed BCD value right _N digits.

processor	Latency	Throughput
power8	25-34	2/cycle
power9	3-6	2/cycle

Parameters

vra	128-bit vector signed BCD 31 digit value.

Parameters

↩ _↩ *N*	int constant for the number of digits to shift right.

Returns

a 128-bit vector BCD value shifted right _N digits.

7.1.6.42 vec_bcdsruqi()

```
static vBCD_t vec_bcdsruqi (
            vBCD_t vra,
            const unsigned int _N )  [inline], [static]
```

Vector BCD Shift Right Unsigned Quadword immediate.

Shift a vector unsigned BCD value right _N digits.

processor	Latency	Throughput
power8	6-15	2/cycle
power9	3-6	2/cycle

Parameters

vra	128-bit vector unsigned BCD 32 digit value.
↩ _↩ *N*	int constant for the number of digits to shift right.

Returns

a 128-bit vector BCD value shifted right _N digits.

7.1.6.43 vec_bcdsub()

```
static vBCD_t vec_bcdsub (
            vBCD_t a,
            vBCD_t b )  [inline], [static]
```

Subtract two Vector Signed BCD 31 digit values.

Subtract Signed 31 digit values and return the lower 31 digits of of the result. Overflow (carry-out/barrow) is ignored.

processor	Latency	Throughput
power8	13	1/cycle
power9	3	2/cycle

Parameters

a	a 128-bit vector treated a signed BCD 31 digit value.
b	a 128-bit vector treated a signed BCD 31 digit value.

Returns

a 128-bit vector which is the lower 31 digits of (a - b).

7.1.6.44 vec_bcdsubcsq()

```
static vBCD_t vec_bcdsubcsq (
             vBCD_t a,
             vBCD_t b )  [inline], [static]
```

Decimal Sudtract & write Carry Signed Quadword.

Two Signed 31 digit BCD values are subtracted, and the carry-out (the high order 32nd digit) of the difference is returned.

Note

This operation will only detect overflows where the operand signs differ. It will not detect a borrow if the signs match. So this operation should only be used if differing signs are guaranteed.

processor	Latency	Throughput
power8	15-21	1/cycle
power9	6-18	2/cycle

Parameters

a	a 128-bit vector treated as a signed BCD 31 digit value.
b	a 128-bit vector treated as a signed BCD 31 digit value.

Returns

a 128-bit vector with the carry digit. Values are -1, 0, and +1.

7.1.6.45 vec_bcdsubecsq()

```
static vBCD_t vec_bcdsubecsq (
             vBCD_t a,
             vBCD_t b,
             vBCD_t c )  [inline], [static]
```

Decimal Add Extended & write Carry Signed Quadword.

Two Signed 31 digit values and a signed carry-in are added together and the carry-out (the high order 32nd digit) of the sum is returned.

Note

This operation will only detect overflows where the operand signs differ. It will not detect a borrow if the signs match. So this operation should only be used if differing signs are guaranteed.

processor	Latency	Throughput
power8	28-37	1/cycle
power9	9-21	2/cycle

Parameters

a	a 128-bit vector treated as a signed BCD 31 digit value.
b	a 128-bit vector treated as a signed BCD 31 digit value.
c	a 128-bit vector treated as a signed BCD carry with values -1, 0, or +1.

Returns

a 128-bit vector with the carry digit from the sum (a + b +c). Carry values are -1, 0, and +1.

7.1.6.46 vec_bcdsubesqm()

```
static vBCD_t vec_bcdsubesqm (
             vBCD_t a,
             vBCD_t b,
             vBCD_t c )  [inline], [static]
```

Decimal Subtract Extended Signed Modulo Quadword.

Two Signed 31 digit values and a signed carry-in are subtracted (a - b- c) and lower 31 digits of the subtraction is returned. Overflow (carry-out) is ignored.

processor	Latency	Throughput
power8	26	1/cycle
power9	6	2/cycle

Parameters

a	a 128-bit vector treated as a signed BCD 31 digit value.
b	a 128-bit vector treated as a signed BCD 31 digit value.
c	a 128-bit vector treated as a signed BCD carry with values -1, 0, or +1.

Returns

a 128-bit vector which is the lower 31 digits of (a + b + c).

7.1.6.47 vec_bcdtrunc()

```
static vBCD_t vec_bcdtrunc (
             vBCD_t vra,
             vui16_t vrb )  [inline], [static]
```

Decimal Truncate. Truncate a vector signed BCD value vra to N-digits, where N is the unsigned integer value in bits 48-63 of vrb. The first 31-N digits are set to 0 and the result returned.

processor	Latency	Throughput
power8	18-27	1/cycle
power9	3	2/cycle

Parameters

vra	128-bit vector treated as a signed BCD 31 digit value.
vrb	Digit truncate count in vector halfword 3 (bits 48:63).

Returns

a 128-bit vector BCD value with the first 31-count digits set to 0.

7.1.6.48 vec_bcdtruncqi()

```
static vBCD_t vec_bcdtruncqi (
             vBCD_t vra,
             const unsigned short _N )  [inline], [static]
```

Decimal Truncate Quadword Immediate. Truncate a vector signed BCD value vra to N-digits, where N is a unsigned short integer constant. The first 31-N digits are set to 0 and the result returned.

processor	Latency	Throughput
power8	6-17	1/cycle
power9	6	2/cycle

Parameters

vra	128-bit vector treated as a signed BCD 31 digit value.
↩ _↩ *N*	a unsigned short integer constant truncate count.

Returns

a 128-bit vector BCD value with the first 31-count digits set to 0.

7.1.6.49 vec_bcdus()

```
static vBCD_t vec_bcdus (
            vBCD_t vra,
            vi8_t vrb )  [inline], [static]
```

Decimal Unsigned Shift. Shift a vector unsigned BCD value, left or right a variable amount of digits (nibbles).

processor	Latency	Throughput
power8	12-14	1/cycle
power9	3	2/cycle

Parameters

vra	128-bit vector treated as a signed BCD 32 digit value.
vrb	Digit shift count in vector byte 7.

Returns

a 128-bit vector BCD value shifted right digits.

7.1.6.50 vec_bcdutrunc()

```
static vBCD_t vec_bcdutrunc (
             vBCD_t vra,
             vui16_t vrb )  [inline], [static]
```

Decimal Unsigned Truncate. Truncate a vector unsigned BCD value vra to N-digits, where N is the unsigned integer value in bits 48-63 of vrb. The first 32-N digits are set to 0 and the result returned.

processor	Latency	Throughput
power8	16-25	1/cycle
power9	3	2/cycle

Parameters

vra	128-bit vector treated as an unsigned BCD 32 digit value.
vrb	Digit truncate count in vector halfword 3 (bits 48:63).

Returns

a 128-bit vector BCD value with the first 32-count digits set to 0.

7.1.6.51 vec_bcdutruncqi()

```
static vBCD_t vec_bcdutruncqi (
             vBCD_t vra,
             const unsigned short _N )  [inline], [static]
```

Decimal Unsigned Truncate Quadword Immediate. Truncate a vector unsigned BCD value vra to N-digits, where N is a unsigned short integer constant. The first 32-N digits are set to 0 and the result returned.

processor	Latency	Throughput
power8	6-17	1/cycle
power9	6	2/cycle

Parameters

vra	128-bit vector treated as a signed BCD 31 digit value.
↩ _↩ *N*	a unsigned short integer constant truncate count.

Returns

a 128-bit vector BCD value with the first 32-count digits set to 0.

7.1.6.52 vec_BIN2BCD()

```
static vBCD_t vec_BIN2BCD (
            vui64_t val )  [inline], [static]
```

Convert vector unsigned doubleword binary values to Vector unsigned 16-digit BCD values.

Convert a vector of 2 unsigned long int doubleword to 2 16-digit unsigned BCD doublewords. Input doublewords should each be in the range 0-9999999999999999.

processor	Latency	Throughput
power8	69	1/19 cycle
power9	58	1/21 cycle

Parameters

val	a vector unsigned long int.

Returns

a 128-bit vector treated a 2 unsigned BCD 16 digit values.

7.1.6.53 vec_cbcdaddcsq()

```
static vBCD_t vec_cbcdaddcsq (
            vBCD_t * cout,
            vBCD_t a,
            vBCD_t b )  [inline], [static]
```

Combined Decimal Add & Write Carry Signed Quadword.

Two Signed 31 digit BCD values are added, and the carry-out (the high order 32nd digit) of the sum is generated. Alternatively if the intermediate sum changes sign we need to, borrow '1' from the magnitude of the higher BCD value and correct (invert by subtracting from 10∗∗31) the intermediate sum. Both the sum and the carry/borrow are returned.

processor	Latency	Throughput
power8	15-24	1/cycle
power9	6-15	2/cycle

Parameters

cout	a pointer to a 128-bit vector to recieve the BCD carry-out.
a	a 128-bit vector treated as a signed BCD 31 digit value.
b	a 128-bit vector treated as a signed BCD 31 digit value.

Returns

a 128-bit vector with the low order 31-digits of the sum (a+b). Values are -1, 0, and +1.

7.1.6.54 vec_cbcdaddecsq()

```
static vBCD_t vec_cbcdaddecsq (
            vBCD_t * cout,
            vBCD_t a,
            vBCD_t b,
            vBCD_t cin ) [inline], [static]
```

Combined Decimal Add Extended & write Carry Signed Quadword.

Two Signed 31 digit values and a signed carry-in are added together and the carry-out (the high order 32nd digit) of the sum is generated. Alternatively if the intermediate sum changes sign we need to, borrow '1' from the magnitude of the next higher BCD value and correct (invert by subtracting from 10∗∗31) the intermediate sum. Both the sum and the carry/borrow are returned.

processor	Latency	Throughput
power8	54-63	1/cycle
power9	15-24	2/cycle

Parameters

cout	a pointer to a 128-bit vector to recieve the BCD carry-out.
a	a 128-bit vector treated as a signed BCD 31 digit value.
b	a 128-bit vector treated as a signed BCD 31 digit value.
cin	a 128-bit vector treated as a signed BCD carry with values -1, 0, or +1.

Returns

a 128-bit vector with the low order 31-digits of the sum (a+b).

7.1.6.55 vec_cbcdmul()

```
static vBCD_t vec_cbcdmul (
            vBCD_t * p_high,
            vBCD_t a,
            vBCD_t b )  [inline], [static]
```

Combined Vector Signed BCD Multiply High/Low.

Two Signed 31 digit values are multiplied and generates the 62 digit product.

The vector unit does not have a BCD multiply, so we convert the operands to _Decimal128 format and use the DFP quadword multiply. This gets tricky as the product can be up to 62 digits, and _Decimal128 format can only hold 34 digits.

To avoid overflow in the DFP Facility, we split each BCD operand into 15 upper and 16 lower digit halves. This requires up four decimal multiplies and produces four 30-32 digit partial products. These are aligned appropriately (via DFP decimal shift) and summed (via DFP Decimal add) to generate the high and low (31-digit) parts of the 62 digit product.

In this case we compute and return the whole 62-digit product split into two 31-digit BCD vectors.

Note

There is early exit case if both operands are 16-digits or less. Here the product can not exceed 32-digits and requires only a single DFP multiply. The DFP2BCD conversion will extract the lower 31-digits. Then DFP Decimal shift will isolate the high (32nd) digit.

processor	Latency	Throughput
power8	107-413	1/cycle
power9	115-294	1/cycle

Parameters

a	a 128-bit vector treated as a signed BCD 31 digit value.
b	a 128-bit vector treated as a signed BCD 31 digit value.
p_high	a pointer to a 128-bit vector to receive the high 31-digits of the product (a ∗ b).

Returns

a 128-bit vector which is the lower 31 digits of (a ∗ b).

7.1.6.56 vec_cbcdsubcsq()

```
static vBCD_t vec_cbcdsubcsq (
            vBCD_t * cout,
            vBCD_t a,
            vBCD_t b )  [inline], [static]
```

Combined Decimal Subtract & Write Carry Signed Quadword.

Subtract (a -b) Signed 31 digit BCD values and detect the carry/barrow (the high order 32nd digit). If the intermediate sum changes sign we need to, borrow '1' from the magnitude of the higher BCD value and correct (invert by subtracting from 10∗∗31) the intermediate sum. Both the sum and the carry/borrow are returned.

processor	Latency	Throughput
power8	15-24	1/cycle
power9	6-15	2/cycle

Parameters

cout	a pointer to a 128-bit vector to recieve the BCD carry-out (alues are -1, 0, and +1).
a	a 128-bit vector treated as a signed BCD 31 digit value.
b	a 128-bit vector treated as a signed BCD 31 digit value.

Returns

a 128-bit vector with the low order 31-digits of the difference (a+b).

7.1.6.57 vec_DFP2BCD()

```
static vBCD_t vec_DFP2BCD (
            _Decimal128 val )  [inline], [static]
```

Convert a __Decimal128 value to Vector BCD.

The _Decimal128 value is converted to a signed BCD 31 digit value via "DFP Decode DPD To BCD Quad". The conversion result is still in a double float register pair and so is permuted into single vector register for use.

processor	Latency	Throughput
power8	17	1/cycle
power9	15	1/cycle

Parameters

val	a __Decimal128 in a double float pair.

Returns

a 128-bit vector treated a signed BCD 31 digit value.

7.1.6.58 vec_pack_Decimal128()

```
static vf64_t vec_pack_Decimal128 (
            _Decimal128 lval )  [inline], [static]
```

Pack a FPR pair (_Decimal128) to a doubleword vector (vector double).

processor	Latency	Throughput
power8	2	2/cycle
power9	3	2/cycle

Parameters

lval	FPR pair containing a _Decimal128.

Returns

vector double containing the doublewords of the FPR pair.

7.1.6.59 vec_quantize0_Decimal128()

```
static _Decimal128 vec_quantize0_Decimal128 (
            _Decimal128 val )  [inline], [static]
```

Quantize (truncate) a _Decimal128 value before convert to BCD.

Truncate (round toward 0) and justify right the input _Decimal128 value so that the unit digit is in the right most position. This supports BCD multiply and divide using DFP instructions by truncating fractional digits before conversion back to BCD.

processor	Latency	Throughput
power8	15	1/cycle
power9	12	1/cycle

Parameters

val	a _Decimal128 value.

Returns

The quantized __Decimal128 value in a double float pair.

7.1.6.60 vec_rdxcf100b()

```
static vui8_t vec_rdxcf100b (
             vui8_t vra ) [inline], [static]
```

Vector Decimal Convert Binary Coded Decimal (BCD) digit pairs from radix 100 binary integer bytes.

Convert 16 radix 100 digits to 32 BCD Format decimal digits. Input is radix 100 digits as binary bytes in the range 0-99. Each byte converted to the equivalent BCD digit pair in adjacent nibbles.

This can be used as the last stage operation in wider binary to decimal conversions.

Note

the nibble high to low digit word is effectively big endian. This matches the digit order precedence of Decimal Add/Subtract.

processor	Latency	Throughput
power8	24-34	1/cycle
power9	27-37	1/cycle

Parameters

vra	a 128-bit vector treated as a vector unsigned char of radix 100 digits.

Returns

128-bit vector unsigned char of BCD nibble pairs in the range 0-9.

7.1.6.61 vec_rdxcf100mw()

```
static vui16_t vec_rdxcf100mw (
             vui32_t vra ) [inline], [static]
```

Vector Decimal Convert radix 10**8 Binary words to pairs of radix 10,000 binary halfwords.

Convert 4 radix 10**8 digits to 8 adjacent radix 10,000 digits. Input is radix 10**8 digits as binary words in the range 0-99999999. Each word converted to the equivalent radix 10,000 pair in adjacent halfword.

This can be used as a intermediate stage operation in wider binary to decimal conversions.

Note

The high to low digit order is effectively big endian. This matches the digit order precedence of Decimal Add/↩ Subtract.

processor	Latency	Throughput
power8	18-25	1/cycle
power9	19-26	1/cycle

Parameters

vra	a 128-bit vector treated as a vector unsigned int of radix 10∗∗8 digits.

Returns

128-bit vector unsigned short radix 10,000 pairs in the range 0-9999.

7.1.6.62 vec_rdxcf10E16d()

```
static vui32_t vec_rdxcf10E16d (
            vui64_t vra )  [inline], [static]
```

Vector Decimal Convert radix 10∗∗16 Binary doublewords to pairs of radix 10∗∗8 binary words.

Convert 2 radix 10∗∗16 digits to 4 adjacent radix 10∗∗8 digits. Input is radix 10∗∗16 digits as binary doublewords in the range 0-9999999999999999. Each doubleword converted to the equivalent radix 10∗∗8 pair in adjacent words.

This can be used as a intermediate stage operation in wider binary to decimal conversions.

Note

The high to low digit order is effectively big endian. This matches the digit order precedence of Decimal Add/↩ Subtract.

processor	Latency	Throughput
power8	51-61	1/cycle
power9	30-40	1/cycle

Parameters

vra	a 128-bit vector treated as a vector unsigned long of radix 10∗∗16 digits.

Returns

128-bit vector unsigned short radix 10∗∗8 pairs in the range 0-99999999.

7.1.6.63 vec_rdxcf10e32q()

```
static vui64_t vec_rdxcf10e32q (
            vui128_t vra )  [inline], [static]
```

Vector Decimal Convert radix 10∗∗32 Binary quadword to pairs of radix 10∗∗16 binary doublewords.

Convert a binary quadword to 2 adjacent radix 10∗∗16 digits. Input is a binary quadwords in the range 0-99999999999999999999999999999999. The quadword converted to the equivalent radix 10∗∗18 pair in adjacent doublewords.

This can be used as a first stage operation in binary to decimal conversions.

Note

Results are undefined if the input value is greater than 10∗∗32 - 1. See Converting Vector __int128 values to BCD for details.

The high to low digit order is effectively big endian. This matches the digit order precedence of Decimal Add/↩ Subtract.

processor	Latency	Throughput
power8	85-95	1/cycle
power9	56-66	1/cycle

Parameters

vra	a 128-bit vector treated as a vector unsigned __int128 in the range 0-99999999999999999999999999999999.

Returns

128-bit vector unsigned long radix 10∗∗16 pairs in the range 0-9999999999999999.

7.1.6.64 vec_rdxcf10kh()

```
static vui8_t vec_rdxcf10kh (
            vui16_t vra )  [inline], [static]
```

Vector Decimal Convert radix 10,000 Binary halfwords to pairs of radix 100 binary bytes.

Convert 8 radix 10,000 digits to 16 adjacent radix 100 digits. Input is radix 10,000 digits as binary halfwords in the range 0-9999. Each halfword converted to the equivalent radix 100 pair in adjacent bytes.

This can be used as a intermediate stage operation in wider binary to decimal conversions.

Note

The high to low digit order is effectively big endian. This matches the digit order precedence of Decimal Add/↩ Subtract.

processor	Latency	Throughput
power8	24-34	1/cycle
power9	27-37	1/cycle

Parameters

vra	a 128-bit vector treated as a vector unsigned short of radix 10,000 digits.

Returns

128-bit vector unsigned char radix 100 pairs in the range 0-99.

7.1.6.65 vec_rdxcfzt100b()

```
static vui8_t vec_rdxcfzt100b (
            vui8_t zone00,
            vui8_t zone16 )  [inline], [static]
```

Vector Decimal Convert Zoned Decimal digit pairs to to radix 100 binary integer bytes..

Convert 32 decimal digits from Zoned Format (one character per digit, in 2 vectors) to Binary coded century format. Century format is adjacent digit pairs converted to a binary integer in the range 0-99. Each century digit is stored in a byte. Input values should be valid decimal characters in the range 0-9.

Note

Zoned numbers are character strings with the high order digit on the left.

The high to low digit order is effectively big endian. This matches the digit order precedence of Decimal Add/↩ Subtract.

This can be used as the first stage operation in wider decimal to binary conversions. Basically the result of this stage are binary coded 100s "digits" that can be passed to vec_bcdctb10ks().

processor	Latency	Throughput
power8	15-17	1/cycle
power9	17-20	1/cycle

Parameters

zone00	a 128-bit vector char containing the high order 16 digits of a 32-digit number.
zone16	a 128-bit vector char containing the low order 16 digits of a 32-digit number.

Returns

> 128-bit vector unsigned char. For each byte, 2 adjacent zoned digits are converted to the equivalent binary representation in the range 0-99.

7.1.6.66 vec_rdxct100b()

```
static vui8_t vec_rdxct100b (
            vui8_t vra )  [inline], [static]
```

Vector Decimal Convert Binary Coded Decimal (BCD) digit pairs to radix 100 binary integer bytes.

Convert 32 decimal digits from BCD Format (one 4-bit nibble per digit) to Binary coded century format. Century format is adjacent digit pairs converted to a binary integer in the range 0-99. Each century digit is stored in a byte. Input values should be valid BCD nibbles in the range 0-9.

This can be used as the first stage operation in wider decimal to binary conversions. Basically the result of this stage are binary coded Century "digits" that can be passed to vec_bcdctb10ks().

Note

> the nibble high to low digit word is effectively big endian. This matches the digit order precedence of Decimal Add/Subtract.

processor	Latency	Throughput
power8	13-22	1/cycle
power9	14-23	1/cycle

Parameters

vra	a 128-bit vector treated as a vector unsigned char of BCD nibble pairs.

Returns

128-bit vector unsigned char, For each byte, BCD digit pairs are converted to the equivalent binary representation in the range 0-99.

7.1.6.67 vec_rdxct100mw()

```
static vui32_t vec_rdxct100mw (
            vui16_t vra )  [inline], [static]
```

Vector Decimal Convert radix 10,000 digit halfword pairs to radix 100,000,000 binary integer words.

Convert from 10k digit Format (one 10k per halfword) to Binary coded 100m (one per word) format. 100m format is adjacent 10k digit pairs converted to a binary integer in the range 0-99999999. Input halfword values should be valid 10Ks in the range 0-9999. The result will be binary int values in the range 0-99999999.

This can be used as the intermediate stage operation in a wider BCD to binary conversions. Basically the result of this stage are binary coded 100,000,000s "digit" words which can be passed to vec_bcdctb10es().

Note

the 10k digit high to low order is effectively big endian. This matches the digit order precedence of Decimal Add/Subtract.

processor	Latency	Throughput
power8	9-18	1/cycle
power9	9-18	1/cycle

Parameters

vra	a 128-bit vector treated as a vector unsigned short of radix 10k digit pairs.

Returns

128-bit vector unsigned int. For each halfword, adjacent 10k digit pairs are converted to the equivalent binary word integer representation in the range 0-99999999.

7.1.6.68 vec_rdxct10E16d()

```
static vui64_t vec_rdxct10E16d (
            vui32_t vra )  [inline], [static]
```

Vector Decimal Convert radix 100,000,000 digit word pairs to radix 10E16 binary integer doublewords.

Convert from 100m digit format (one 100m digit per word) to Binary coded 10p (one per doubleword) format. 10p format is adjacent 100m digit pairs converted to a binary long integer in the range 0-9999999999999999 (10 quadrillion). Input word values should be valid 100m in the range 0-99999999.

This can be used as the intermediate stage operation in a wider BCD to binary conversions. Basically the result of this stage are binary coded 10,000,000,000,000,000s "digits" doublewords which can be passed to vec_bcdctb10e32().

Note

> the 100m digit high to low order is effectively big endian. This matches the digit order precedence of Decimal Add/Subtract.

processor	Latency	Throughput
power8	9-18	1/cycle
power9	9-18	1/cycle

Parameters

vra	a 128-bit vector treated as a vector unsigned int of radix 100m digit pairs.

Returns

> 128-bit vector unsigned long. For each word pair, containing 8 digit equivalent value each, adjacent 100m digits are converted to the equivalent binary doubleword representation in the range 0-9999999999999999.

7.1.6.69 vec_rdxct10e32q()

```
static vui128_t vec_rdxct10e32q (
            vui64_t vra )  [inline], [static]
```

Vector Decimal Convert radix 10E16 digit pairs to radix 10E32 __int128 quadwords.

Convert from 10p digit format (one 10p digit per doubleword) to binary __int128 (one per quadword) format. Input doubleword values should be valid long integers in the range 0-9999999999999999. The result will be a binary _int128 value in the range 0-99999999999999999999999999999999.

This can be used as the final stage operation in a 32-digit BCD to binary __int128 conversion.

Note

> the 10e16-1 digit high to low order is effectively big endian. This matches the digit order precedence of Decimal Add/Subtract.

processor	Latency	Throughput
power8	25-32	1/cycle
power9	10-19	2/cycle

Parameters

vra	a 128-bit vector treated as a vector unsigned long of radix 10e16 digit pairs.

Returns

> 128-bit vector unsigned __int128. The doubleword pair, of 16 equivalent digits each, are converted to the equivalent binary quadword representation in the range 0-99999999999999999999999999999999.

7.1.6.70 vec_rdxct10kh()

```
static vui16_t vec_rdxct10kh (
            vui8_t vra )  [inline], [static]
```

Vector Decimal Convert radix 100 digit pairs to radix 10,000 binary integer halfwords.

Convert from 16 century digit Format (one century per byte) to 8 Binary coded 10k (one per halfword) format. 10K format is adjacent century digit pairs converted to a binary integer in the range 0-9999 . Input byte values should be valid 100s in the range 0-99. The result vector will be 8 short int values in the range 0-9999.

This can be used as the intermediate stage operation in wider BCD to binary conversions. Basically the result of this stage are binary coded 10,000s "digits" which can be passed to vec_bcdctb100ms().

Note

> the 100s digit high to low order is effectively big endian. This matches the digit order precedence of Decimal Add/Subtract.

processor	Latency	Throughput
power8	9-18	1/cycle
power9	9-18	1/cycle

Parameters

vra	a 128-bit vector treated as a vector unsigned char of radix 100 digit pairs.

Returns

128-bit vector unsigned short. For each halfword, adjacent pairs of century digits pairs are converted to the equivalent binary halfword representation in the range 0-9999.

7.1.6.71 vec_setbool_bcdinv()

```
static vb128_t vec_setbool_bcdinv (
            vBCD_t vra ) [inline], [static]
```

Vector Set Bool from Signed BCD Quadword if invalid.

If the quadword's sign nibble is 0xB, 0xD, 0xA, 0xC, 0xE, or 0xF and all 31 digit nibbles 0-9 then return a vector bool __int128 that is all '0's. Otherwise return all '1's.

processor	Latency	Throughput
power8	15 - 39	1/cycle
power9	3 - 15	1/cycle

Parameters

vra	a 128-bit vector treated as signed BCD quadword.

Returns

a 128-bit vector bool of all '0's if the BCD digits and sign are valid. Otherwise all '1's.

7.1.6.72 vec_setbool_bcdsq()

```
static vb128_t vec_setbool_bcdsq (
            vBCD_t vra ) [inline], [static]
```

Vector Set Bool from Signed BCD Quadword.

If the quadword's sign nibble is 0xB or 0xD then return a vector bool __int128 that is all '1's. Otherwise if the sign nibble is 0xA, 0xC, 0xE, or 0xF then return all '0's.

/note For _ARCH_PWR7 and earlier (No vector BCD instructions),

this implementation only tests for a valid plus sign nibble. Otherwise the BCD value is assumed to be negative.

processor	Latency	Throughput
power8	17 - 26	2/cycle
power9	5 - 14	2/cycle

Parameters

vra	a 128-bit vector treated as signed BCD quadword.

Returns

a 128-bit vector bool of all '1's if the sign is negative. Otherwise all '0's.

7.1.6.73 vec_signbit_bcdsq()

```
static int vec_signbit_bcdsq (
            vBCD_t vra )  [inline], [static]
```

Vector Sign bit from Signed BCD Quadword.

If the quadword's sign nibble is 0xB or 0xD then return a non-zero value. Otherwise if the sign nibble is 0xA, 0xC, 0xE, or 0xF then return all '0's.

/note For _ARCH_PWR7 and earlier (No vector BCD instructions), this implementation only tests for a valid minus sign nibble. Otherwise the BCD value is assumed to be positive.

processor	Latency	Throughput
power8	15 - 26	2/cycle
power9	5 - 14	2/cycle

Parameters

vra	a 128-bit vector treated as signed BCD quadword.

Returns

a none-zero value if the sign is negative. Otherwise return '0's.

7.1.6.74 vec_unpack_Decimal128()

```
static _Decimal128 vec_unpack_Decimal128 (
            vf64_t lval )  [inline], [static]
```

Unpack a doubleword vector (vector double) into a FPR pair. (_Decimal128).

processor	Latency	Throughput
power8	2	1/cycle
power9	3	1/cycle

Parameters

lval	FPR pair containing a _Decimal128.

Returns

FPR pair containing a _Decimal128.

7.1.6.75 vec_zndctuq()

```
static vui128_t vec_zndctuq (
            vui8_t zone00,
            vui8_t zone16 )  [inline], [static]
```

Vector Zoned Decimal Convert 32 digits to binary unsigned quadword.

Vector convert 2x quadwords each containing 16 digits to the equivalent unsigned __int128, in the range 0-99999999999999999999999999999999. Input values should be valid 32 zoned digits in the range '0'-'9'.

processor	Latency	Throughput
power8	67-73	1/cycle
power9	55-62	1/cycle

Parameters

zone00	a 128-bit vector char containing the high order 16 digits of a 32-digit number.
zone16	a 128-bit vector char containing the low order 16 digits of a 32-digit number.

Returns

128-bit vector unsigned __int128 in the range 0-99999999999999999999999999999999.

7.2 src/pveclib/vec_char_ppc.h File Reference

Header package containing a collection of 128-bit SIMD operations over 8-bit integer (char) elements.

```
#include <pveclib/vec_common_ppc.h>
```

Functions

- static vui8_t vec_absdub (vui8_t vra, vui8_t vrb)

  Vector Absolute Difference Unsigned byte.
- static vui8_t vec_clzb (vui8_t vra)

  Vector Count Leading Zeros Byte for a unsigned char (byte) elements.
- static vui8_t vec_ctzb (vui8_t vra)

  Vector Count Trailing Zeros Byte for a unsigned char (byte) elements.
- static vui8_t vec_isalnum (vui8_t vec_str)

  Vector isalpha.
- static vui8_t vec_isalpha (vui8_t vec_str)

  Vector isalnum.
- static vui8_t vec_isdigit (vui8_t vec_str)

  Vector isdigit.
- static vui8_t vec_mrgahb (vui16_t vra, vui16_t vrb)

  Vector Merge Algebraic High Byte operation.
- static vui8_t vec_mrgalb (vui16_t vra, vui16_t vrb)

  Vector Merge Algebraic Low Byte operation.
- static vui8_t vec_mrgeb (vui8_t vra, vui8_t vrb)

  Vector Merge Even Bytes operation.
- static vui8_t vec_mrgob (vui8_t vra, vui8_t vrb)

  Vector Merge Odd Halfwords operation.
- static vi8_t vec_mulhsb (vi8_t vra, vi8_t vrb)

  Vector Multiply High Signed Bytes.
- static vui8_t vec_mulhub (vui8_t vra, vui8_t vrb)

  Vector Multiply High Unsigned Bytes.
- static vui8_t vec_mulubm (vui8_t vra, vui8_t vrb)

  Vector Multiply Unsigned Byte Modulo.
- static vui8_t vec_popcntb (vui8_t vra)

  Vector Population Count byte.
- static vb8_t vec_setb_sb (vi8_t vra)

  Vector Set Bool from Signed Byte.
- static vui8_t vec_slbi (vui8_t vra, const unsigned int shb)

  Vector Shift left Byte Immediate.
- static vi8_t vec_srabi (vi8_t vra, const unsigned int shb)

  Vector Shift Right Algebraic Byte Immediate.
- static vui8_t vec_srbi (vui8_t vra, const unsigned int shb)

  Vector Shift Right Byte Immediate.
- static vui8_t vec_shift_leftdo (vui8_t vrw, vui8_t vrx, vui8_t vrb)

  Shift left double quadword by octet. Return a vector unsigned char that is the left most 16 chars after shifting left 0-15 octets (chars) of the 32 char double vector (vrw||vrx). The octet shift amount is from bits 121:124 of vrb.
- static vui8_t vec_toupper (vui8_t vec_str)

  Vector toupper.
- static vui8_t vec_tolower (vui8_t vec_str)

  Vector tolower.
- static vui8_t vec_vmrgeb (vui8_t vra, vui8_t vrb)

  Vector Merge Even Bytes.
- static vui8_t vec_vmrgob (vui8_t vra, vui8_t vrb)

  Vector Merge Odd Byte.

7.2.1 Detailed Description

Header package containing a collection of 128-bit SIMD operations over 8-bit integer (char) elements.

Most of these operations are implemented in a single VMX or VSX instruction on newer (POWER6/POWER7/POWER8/POWER9) processors. This header serves to fill in functional gaps for older (POWER7, POWER8) processors and provides in-line assembler implementations for older compilers that do not provide the build-ins.

Most vector char (8-bit integer) operations are are already covered by the original VMX (AKA Altivec) instructions. VMX intrinsic (compiler built-ins) operations are defined in <altivec.h> and described in the compiler documentation. PowerISA 2.07B (POWER8) added several useful byte operations (count leading zeros, population count) not included in the original VMX. PowerISA 3.0B (POWER9) adds several more (absolute difference, compare not equal, count trailing zeros, extend sign, extract/insert, and reverse bytes). Most of these intrinsic (compiler built-ins) operations are defined in <altivec.h> and described in the compiler documentation.

Note

> The compiler disables associated <altivec.h> built-ins if the **mcpu** target does not enable the specific instruction. For example if you compile with **-mcpu=power7**, vec_vclz and vec_vclzb will not be defined. But vec_clzb is always defined in this header, will generate the minimum code, appropriate for the target, and produce correct results.

This header covers operations that are either:

- Implemented in later processors and useful to programmers if the same operations are available on slightly older processors. This is required even if the operation is defined in the OpenPOWER ABI or <altivec.h>, as the compiler disables the associated built-ins if the **mcpu** target does not enable the instruction.
- Defined in the OpenPOWER ABI but *not* yet defined in <altivec.n> provided by available compilers in common use. Examples include Count Leading Zeros and Population Count.
- Commonly used operations, not covered by the ABI or <altivec.h>, and require multiple instructions or are not obvious. Examples include the multiply high, ASCII character tests, and shift immediate operations.

7.2.2 Endian problems with byte operations

It would be useful to provide a vector multiply high byte (return the high order 8-bits of the 16-bit product) operation. This can be used for multiplicative inverse (effectively integer divide) operations. Neither integer multiply high nor divide are available as vector instructions. However the multiply high byte operation can be composed from the existing multiply even/odd byte operations followed by the vector merge even byte operation. Similarly a multiply low (modulo) byte operation can be composed from the existing multiply even/odd byte operations followed by the vector merge odd byte operation.

As a prerequisite we need to provide the merge even/odd byte operations. While PowerISA has added these operations for word and doubleword, instructions are not defined for byte and halfword. Fortunately vector merge operations are just a special case of vector permute. So the vec_vmrgob() and vec_vmrgeb() implementation can use vec_perm and appropriate selection vectors to provide these merge operations.

As described for other element sizes this is complicated by *little-endian* (LE) support as specified in the OpenPOWER ABI and as implemented in the compilers. Little-endian changes the effective vector element numbering and the location of even and odd elements. This means that the vector built-ins provided by altivec.h may not generate the instructions you would expect.

See also

Endian problems with halfword operations

General Endian Issues

So this header defines endian independent byte operations vec_vmrgeb() and vec_vmrgob(). These operations are used in the implementation of the endian sensitive vec_mrgeb() and vec_mrgob(). These support the OpenPOWER ABI mandated merge even/odd semantic.

We also provide the merge algebraic high/low operations vec_mrgahb() and vec_mrgalb() to simplify extended precision arithmetic. These implementations use vec_vmrgeb() and vec_vmrgob() as extended precision byte order does not change with endian. These operations are used in turn to implement multiply byte high/low/modulo (vec_mulhsb(), vec_mulhub(), vec_mulubm()).

These operations provide a basis for using the multiplicative inverse as a alternative to integer divide.

See also

Examples, Divide by integer constant

7.2.3 Performance data.

The performance characteristics of the merge and multiply byte operations are very similar to the halfword implementations. (see Performance data.).

7.2.3.1 More information.

High level performance estimates are provided as an aid to function selection when evaluating algorithms. For background on how *Latency* and *Throughput* are derived see: Performance data.

7.2.4 Function Documentation

7.2.4.1 vec_absdub()

```
static vui8_t vec_absdub (
              vui8_t vra,
              vui8_t vrb )  [inline], [static]
```

Vector Absolute Difference Unsigned byte.

Compute the absolute difference for each byte. For each unsigned byte, subtract B[i] from A[i] and return the absolute value of the difference.

processor	Latency	Throughput
power8	4	1/cycle
power9	3	2/cycle

Parameters

vra	vector of 16 unsigned bytes
vrb	vector of 16 unsigned bytes

Returns

vector of the absolute difference.

7.2.4.2 vec_clzb()

```
static vui8_t vec_clzb (
             vui8_t vra )  [inline], [static]
```

Vector Count Leading Zeros Byte for a unsigned char (byte) elements.

Count the number of leading '0' bits (0-7) within each byte element of a 128-bit vector.

For POWER8 (PowerISA 2.07B) or later use the Vector Count Leading Zeros Byte instruction **vclzb**. Otherwise use sequence of pre 2.07 VMX instructions. SIMDized count leading zeros inspired by:

Warren, Henry S. Jr and *Hacker's Delight*, 2nd Edition, Addison Wesley, 2013. Chapter 5 Counting Bits, Figure 5-12.

processor	Latency	Throughput
power8	2	2/cycle
power9	3	2/cycle

Parameters

vra	128-bit vector treated as 16 x 8-bit unsigned integer (byte) elements.

Returns

128-bit vector with the leading zeros count for each byte element.

7.2.4.3 vec_ctzb()

```
static vui8_t vec_ctzb (
            vui8_t vra )  [inline], [static]
```

Vector Count Trailing Zeros Byte for a unsigned char (byte) elements.

Count the number of trailing '0' bits (0-8) within each byte element of a 128-bit vector.

For POWER9 (PowerISA 3.0B) or later use the Vector Count Trailing Zeros Byte instruction **vctzb**. Otherwise use a sequence of pre ISA 3.0 VMX instructions. SIMDized count trailing zeros inspired by:

Warren, Henry S. Jr and *Hacker's Delight*, 2nd Edition, Addison Wesley, 2013. Chapter 5 Counting Bits, Section 5-4.

processor	Latency	Throughput
power8	6-8	2/cycle
power9	3	2/cycle

Parameters

vra	128-bit vector treated as 16 x 8-bit unsigned char (byte) elements.

Returns

128-bit vector with the trailing zeros count for each byte element.

7.2.4.4 vec_isalnum()

```
static vui8_t vec_isalnum (
            vui8_t vec_str )  [inline], [static]
```

Vector isalpha.

Return a vector boolean char with a true indicator for any character that is either Lower Case Alpha ASCII or Upper Case ASCII. False otherwise.

processor	Latency	Throughput
power8	10-20	1/cycle
power9	11-21	1/cycle

Parameters

vec_str	vector of 16 ASCII characters

Returns

vector bool char of the isalpha operation applied to each character of vec_str. For each byte 0xff indicates true (isalpha), 0x00 indicates false.

7.2.4.5 vec_isalpha()

```
static vui8_t vec_isalpha (
            vui8_t vec_str ) [inline], [static]
```

Vector isalnum.

Return a vector boolean char with a true indicator for any character that is either Lower Case Alpha ASCII, Upper Case ASCII, or numeric ASCII. False otherwise.

processor	Latency	Throughput
power8	9-18	1/cycle
power9	10-19	1/cycle

Parameters

vec_str	vector of 16 ASCII characters

Returns

vector bool char of the isalnum operation applied to each character of vec_str. For each byte 0xff indicates true (isalpha), 0x00 indicates false.

7.2.4.6 vec_isdigit()

```
static vui8_t vec_isdigit (
            vui8_t vec_str ) [inline], [static]
```

Vector isdigit.

Return a vector boolean char with a true indicator for any character that is ASCII decimal digit. False otherwise.

processor	Latency	Throughput
power8	4-13	1/cycle
power9	5-14	1/cycle

Parameters

vec_str	vector of 16 ASCII characters

Returns

vector bool char of the isdigit operation applied to each character of vec_str. For each byte 0xff indicates true (isdigit), 0x00 indicates false.

7.2.4.7 vec_mrgahb()

```
static vui8_t vec_mrgahb (
            vui16_t vra,
            vui16_t vrb )  [inline], [static]
```

Vector Merge Algebraic High Byte operation.

Merge only the high byte from 16 x Algebraic halfwords across vectors vra and vrb. This is effectively the Vector Merge Even Byte operation that is not modified for Endian.

For example merge the high 8-bits from each of 16 x 16-bit products as generated by vec_muleub/vec_muloub. This result is effectively a vector multiply high unsigned byte.

processor	Latency	Throughput
power8	2-13	2/cycle
power9	3-14	2/cycle

Parameters

vra	128-bit vector unsigned short.
vrb	128-bit vector unsigned short.

Returns

A vector merge from only the high bytes of the 16 x Algebraic halfwords across vra and vrb.

7.2.4.8 vec_mrgalb()

```
static vui8_t vec_mrgalb (
            vui16_t vra,
            vui16_t vrb )  [inline], [static]
```

Vector Merge Algebraic Low Byte operation.

Merge only the low bytes from 16 x Algebraic halfwords across vectors vra and vrb. This is effectively the Vector Merge Odd Bytes operation that is not modified for Endian.

For example merge the low 8-bits from each of 16 x 16-bit products as generated by vec_muleub/vec_muloub. This result is effectively a vector multiply low unsigned byte.

processor	Latency	Throughput
power8	2-13	2/cycle
power9	3-14	2/cycle

Parameters

vra	128-bit vector unsigned int.
vrb	128-bit vector unsigned int.

Returns

A vector merge from only the high halfwords of the 8 x Algebraic words across vra and vrb.

7.2.4.9 vec_mrgeb()

```
static vui8_t vec_mrgeb (
            vui8_t vra,
            vui8_t vrb ) [inline], [static]
```

Vector Merge Even Bytes operation.

Merge the even byte elements from the concatenation of 2 x vectors (vra and vrb).

Note

The element numbering changes between Big and Little Endian. So the compiler and this implementation adjusts the generated code to reflect this.

processor	Latency	Throughput
power8	2-13	2/cycle
power9	3-14	2/cycle

Parameters

vra	128-bit vector unsigned char.
vrb	128-bit vector unsigned char.

Returns

A vector merge from only the even bytes of vra and vrb.

7.2.4.10 vec_mrgob()

```
static vui8_t vec_mrgob (
            vui8_t vra,
            vui8_t vrb )  [inline], [static]
```

Vector Merge Odd Halfwords operation.

Merge the odd halfword elements from the concatenation of 2 x vectors (vra and vrb).

Note

The element numbering changes between Big and Little Endian. So the compiler and this implementation adjusts the generated code to reflect this.

processor	Latency	Throughput
power8	2-13	2/cycle
power9	3-14	2/cycle

Parameters

vra	128-bit vector unsigned char.
vrb	128-bit vector unsigned char.

Returns

A vector merge from only the odd bytes of vra and vrb.

7.2.4.11 vec_mulhsb()

```
static vi8_t vec_mulhsb (
            vi8_t vra,
            vi8_t vrb )  [inline], [static]
```

Vector Multiply High Signed Bytes.

Multiple the corresponding byte elements of two vector signed char values and return the high order 8-bits, for each 16-bit product element.

processor	Latency	Throughput
power8	9-13	1/cycle
power9	10-14	1/cycle

Parameters

vra	128-bit vector signed char.
vrb	128-bit vector signed char.

Returns

vector of the high order 8-bits of the product of the byte elements from vra and vrb.

7.2.4.12 vec_mulhub()

```
static vui8_t vec_mulhub (
              vui8_t vra,
              vui8_t vrb )  [inline], [static]
```

Vector Multiply High Unsigned Bytes.

Multiple the corresponding byte elements of two vector unsigned char values and return the high order 8-bits, for each 16-bit product element.

processor	Latency	Throughput
power8	9-13	1/cycle
power9	10-14	1/cycle

Parameters

vra	128-bit vector unsigned char.
vrb	128-bit vector unsigned char.

Returns

vector of the high order 8-bits of the product of the byte elements from vra and vrb.

7.2.4.13 vec_mulubm()

```
static vui8_t vec_mulubm (
              vui8_t vra,
              vui8_t vrb )  [inline], [static]
```

Vector Multiply Unsigned Byte Modulo.

Multiple the corresponding byte elements of two vector unsigned char values and return the low order 8-bits of the 16-bit product for each element.

Note

vec_mulubm can be used for unsigned or signed char integers. It is the vector equivalent of Multiply Low Byte.

processor	Latency	Throughput
power8	9-13	1/cycle
power9	10-14	1/cycle

Parameters

vra	128-bit vector unsigned char.
vrb	128-bit vector unsigned char.

Returns

vector of the low order 8-bits of the unsigned product of the byte elements from vra and vrb.

7.2.4.14 vec_popcntb()

```
static vui8_t vec_popcntb (
             vui8_t vra )  [inline], [static]
```

Vector Population Count byte.

Count the number of '1' bits (0-8) within each byte element of a 128-bit vector.

For POWER8 (PowerISA 2.07B) or later use the Vector Population Count Byte instruction. Otherwise use simple Vector (VMX) instructions to count bits in bytes in parallel. SIMDized population count inspired by:

Warren, Henry S. Jr and *Hacker's Delight*, 2nd Edition, Addison Wesley, 2013. Chapter 5 Counting Bits, Figure 5-2.

processor	Latency	Throughput
power8	2	2/cycle
power9	3	2/cycle

Parameters

vra	128-bit vector treated as 16 x 8-bit integers (byte) elements.

Returns

128-bit vector with the population count for each byte element.

7.2.4.15 vec_setb_sb()

```
static vb8_t vec_setb_sb (
            vi8_t vra )  [inline], [static]
```

Vector Set Bool from Signed Byte.

For each byte, propagate the sign bit to all 8-bits of that byte. The result is vector bool char reflecting the sign bit of each 8-bit byte.

processor	Latency	Throughput
power8	2-4	2/cycle
power9	2-5	2/cycle

Parameters

vra	Vector signed char.

Returns

vector bool char reflecting the sign bit of each byte.

7.2.4.16 vec_shift_leftdo()

```
static vui8_t vec_shift_leftdo (
            vui8_t vrw,
            vui8_t vrx,
            vui8_t vrb )  [inline], [static]
```

Shift left double quadword by octet. Return a vector unsigned char that is the left most 16 chars after shifting left 0-15 octets (chars) of the 32 char double vector (vrw||vrx). The octet shift amount is from bits 121:124 of vrb.

This sequence can be used to align a unaligned 16 char substring based on the result of a vector count leading zero of of the compare boolean.

processor	Latency	Throughput
power8	6-8	1/cycle
power9	8-9	1/cycle

Parameters

vrw	upper 16-bytes of the 32-byte double vector.
vrx	lower 16-bytes of the 32-byte double vector.
vrb	Shift amount in bits 121:124.

Returns

upper 16-bytes of left shifted double vector.

7.2.4.17 vec_slbi()

```
static vui8_t vec_slbi (
            vui8_t vra,
            const unsigned int shb )  [inline], [static]
```

Vector Shift left Byte Immediate.

Shift left each byte element [0-15], 0-7 bits, as specified by an immediate value. The shift amount is a const unsigned int in the range 0-7. A shift count of 0 returns the original value of vra. Shift counts greater then 7 bits return zero.

processor	Latency	Throughput
power8	4-11	2/cycle
power9	5-11	2/cycle

Parameters

vra	a 128-bit vector treated as a vector unsigned char.
shb	Shift amount in the range 0-7.

Returns

128-bit vector unsigned char, shifted left shb bits.

7.2.4.18 vec_srabi()

```
static vi8_t vec_srabi (
            vi8_t vra,
            const unsigned int shb )  [inline], [static]
```

Vector Shift Right Algebraic Byte Immediate.

Shift right each byte element [0-15], 0-7 bits, as specified by an immediate value. The shift amount is a const unsigned int in the range 0-7. A shift count of 0 returns the original value of vra. Shift counts greater then 7 bits return the sign bit propagated to each bit of each element.

processor	Latency	Throughput
power8	4-11	2/cycle
power9	5-11	2/cycle

Parameters

vra	a 128-bit vector treated as a vector signed char.
shb	Shift amount in the range 0-7.

Returns

128-bit vector signed char, shifted right shb bits.

7.2.4.19 vec_srbi()

```
static vui8_t vec_srbi (
            vui8_t vra,
            const unsigned int shb )  [inline], [static]
```

Vector Shift Right Byte Immediate.

Shift right each byte element [0-15], 0-7 bits, as specified by an immediate value. The shift amount is a const unsigned int in the range 0-7. A shift count of 0 returns the original value of vra. Shift counts greater then 7 bits return zero.

processor	Latency	Throughput
power8	4-11	2/cycle
power9	5-11	2/cycle

Parameters

vra	a 128-bit vector treated as a vector unsigned char.
shb	Shift amount in the range 0-7.

Returns

128-bit vector unsigned char, shifted right shb bits.

7.2.4.20 vec_tolower()

```
static vui8_t vec_tolower (
             vui8_t vec_str )  [inline], [static]
```

Vector tolower.

Convert any Upper Case Alpha ASCII characters within a vector unsigned char into the equivalent Lower Case character. Return the result as a vector unsigned char.

processor	Latency	Throughput
power8	8-17	1/cycle
power9	9-18	1/cycle

Parameters

vec_str	vector of 16 ASCII characters

Returns

vector char converted to lower case.

7.2.4.21 vec_toupper()

```
static vui8_t vec_toupper (
             vui8_t vec_str )  [inline], [static]
```

Vector toupper.

Convert any Lower Case Alpha ASCII characters within a vector unsigned char into the equivalent Upper Case character. Return the result as a vector unsigned char.

processor	Latency	Throughput
power8	8-17	1/cycle
power9	9-18	1/cycle

Parameters

vec_str	vector of 16 ASCII characters

Returns

vector char converted to upper case.

7.2.4.22 vec_vmrgeb()

```
static vui8_t vec_vmrgeb (
            vui8_t vra,
            vui8_t vrb )  [inline], [static]
```

Vector Merge Even Bytes.

Merge the even byte elements from the concatenation of 2 x vectors (vra and vrb).

Note

This function implements the operation of a Vector Merge Even Bytes instruction, if the PowerISA included such an instruction. This implementation is NOT Endian sensitive and the function is stable across BE/LE implementations. Using Big Endian element numbering:

- res[0] = vra[0];
- res[1] = vrb[0];
- res[2] = vra[2];
- res[3] = vrb[2];
- res[4] = vra[4];
- res[5] = vrb[4];
- res[6] = vra[6];
- res[7] = vrb[6];
- res[8] = vra[8];
- res[9] = vrb[8];
- res[10] = vra[10];
- res[11] = vrb[10];
- res[12] = vra[12];
- res[13] = vrb[12];
- res[14] = vra[14];
- res[15] = vrb[14];

processor	Latency	Throughput
power8	2-13	2/cycle
power9	3-14	2/cycle

Parameters

vra	128-bit vector unsigned char.
vrb	128-bit vector unsigned char.

Returns

A vector merge from only the even bytes of vra and vrb.

7.2.4.23 vec_vmrgob()

```
static vui8_t vec_vmrgob (
            vui8_t vra,
            vui8_t vrb )  [inline], [static]
```

Vector Merge Odd Byte.

Merge the odd byte elements from the concatenation of 2 x vectors (vra and vrb).

Note

This function implements the operation of a Vector Merge Odd Bytes instruction, if the PowerISA included such an instruction. This implementation is NOT Endian sensitive and the function is stable across BE/LE implementations. Using Big Endian element numbering:

- res[0] = vra[1];
- res[1] = vrb[1];
- res[2] = vra[3];
- res[3] = vrb[3];
- res[4] = vra[5];
- res[5] = vrb[5];
- res[6] = vra[7];
- res[7] = vrb[7];
- res[8] = vra[9];
- res[9] = vrb[9];
- res[10] = vra[11];
- res[11] = vrb[11];
- res[12] = vra[13];
- res[13] = vrb[13];
- res[14] = vra[15];
- res[15] = vrb[15];

processor	Latency	Throughput
power8	2-13	2/cycle
power9	3-14	2/cycle

Parameters

vra	128-bit vector unsigned char.
vrb	128-bit vector unsigned char.

Returns

A vector merge from only the odd bytes of vra and vrb.

7.3 src/pveclib/vec_common_ppc.h File Reference

Common definitions and typedef used by the collection of Power Vector Library (pveclib) headers.

```
#include <stdint.h>
#include <altivec.h>
```

Classes

- union __VEC_U_128

  Union used to transfer 128-bit data between vector and non-vector types.

Macros

- #define CONST_VINT64_DW(__dw0, __dw1) {__dw1, __dw0}

  Arrange elements of dword initializer in high->low order.

- #define CONST_VINT128_DW(__dw0, __dw1) (vui64_t){__dw1, __dw0}

  Initializer for 128-bits vector, as two unsigned long long elements in high->low order. May require an explicit cast.
- #define CONST_VINT128_DW128(__dw0, __dw1) (vui128_t)((vui64_t){__dw1, __dw0})

  A vector unsigned __int128 initializer, as two unsigned long long elements in high->low order.

- #define CONST_VINT128_W(__w0, __w1, __w2, __w3) (vui32_t){__w3, __w2, __w1, __w0}

  Arrange word elements of a unsigned int initializer in high->low order. May require an explicit cast.

- #define CONST_VINT32_W(__w0, __w1, __w2, __w3) {__w3, __w2, __w1, __w0}

  Arrange elements of word initializer in high->low order.

- #define CONST_VINT128_H(__hw0, __hw1, __hw2, __hw3, __hw4, __hw5, __hw6, __hw7) (vui16_t){__hw7, __hw6, __hw5, __hw4, __hw3, __hw2, __hw1, __hw0}

  Arrange halfword elements of a unsigned int initializer in high->low order. May require an explicit cast.

- #define CONST_VINT16_H(__hw0, __hw1, __hw2, __hw3, __hw4, __hw5, __hw6, __hw7) {__hw7, __hw6, __hw5, __hw4, __hw3, __hw2, __hw1, __hw0}

  Arrange elements of halfword initializer in high->low order.

- #define CONST_VINT128_B(_b0, _b1, _b2, _b3, _b4, _b5, _b6, _b7, _b8, _b9, _b10, _b11, _b12, _b13, _b14, _b15) (vui8_t){_b15, _b14, _b13, _b12, _b11, _b10, _b9, _b8, _b7, _b6, _b5, _b4, _b3, _b2, _b1, _b0}

  Arrange byte elements of a unsigned int initializer in high->low order. May require an explicit cast.

- #define CONST_VINT8_B(_b0, _b1, _b2, _b3, _b4, _b5, _b6, _b7, _b8, _b9, _b10, _b11, _b12, _b13, _b14, _b15) {_b15, _b14, _b13, _b12, _b11, _b10, _b9, _b8, _b7, _b6, _b5, _b4, _b3, _b2, _b1, _b0}

  Arrange elements of byte initializer in high->low order.

- #define VEC_DW_H 1

  Element index for high order dword.

- #define VEC_DW_L 0

  Element index for low order dword.

- #define VEC_W_H 3

  Element index for highest order word.

- #define VEC_W_L 0

  Element index for lowest order word.

- #define VEC_WE_0 3

  Element index for vector splat word 0.

- #define VEC_WE_1 2

  Element index for vector splat word 1.

- #define VEC_WE_2 1

  Element index for vector splat word 2.

- #define VEC_WE_3 0

  Element index for vector splat word 3.

- #define VEC_HW_H 7

  Element index for highest order hword.

- #define VEC_HW_L_DWH 4

  Element index for lowest order hword of the high dword.

- #define VEC_HW_L 0

  Element index for lowest order hword.

- #define VEC_BYTE_L 0

  Element index for lowest order byte.

- #define VEC_BYTE_L_DWH 8

  Element index for lowest order byte of the high dword.

- #define VEC_BYTE_L_DWL 0

  Element index for lowest order byte of the low dword.

- #define VEC_BYTE_H 15

Element index for highest order byte.

- #define VEC_BYTE_HHW 14

  Element index for second lowest order byte.

Typedefs

- typedef __vector unsigned char vui8_t

  vector of 8-bit unsigned char elements.
- typedef __vector unsigned short vui16_t

  vector of 16-bit unsigned short elements.
- typedef __vector unsigned int vui32_t

  vector of 32-bit unsigned int elements.
- typedef __vector unsigned long long vui64_t

  vector of 64-bit unsigned long long elements.
- typedef __vector signed char vi8_t

  vector of 8-bit signed char elements.
- typedef __vector short vi16_t

  vector of 16-bit signed short elements.
- typedef __vector int vi32_t

  vector of 32-bit signed int elements.
- typedef __vector long long vi64_t

  vector of 64-bit signed long long elements.
- typedef __vector float vf32_t

  vector of 32-bit float elements.
- typedef __vector double vf64_t

  vector of 64-bit double elements.
- typedef __vector __bool char vb8_t

  vector of 8-bit bool char elements.
- typedef __vector __bool short vb16_t

  vector of 16-bit bool short elements.
- typedef __vector __bool int vb32_t

  vector of 32-bit bool int elements.
- typedef __vector __bool long long vb64_t

  vector of 64-bit bool long long elements.
- typedef __vector __int128 vi128_t

  vector of one 128-bit signed __int128 element.
- typedef __vector unsigned __int128 vui128_t

  vector of one 128-bit unsigned __int128 element.
- typedef __vector __bool __int128 vb128_t

  vector of one 128-bit bool __int128 element.

Functions

- static unsigned __int128 vec_transfer_vui128t_to_uint128 (vui128_t vra)

  Transfer a vector unsigned __int128 to __int128 scalar.
- static vui128_t vec_transfer_uint128_to_vui128t (unsigned __int128 gprp)

  Transfer a __int128 scalar to vector unsigned __int128.
- static unsigned long long scalar_extract_uint64_from_low_uint128 (unsigned __int128 gprp)

  Extract the low doubleword from a __int128 scalar.
- static unsigned long long scalar_extract_uint64_from_high_uint128 (unsigned __int128 gprp)

  Extract the high doubleword from a __int128 scalar.
- static unsigned __int128 scalar_insert_uint64_to_uint128 (unsigned long long high, unsigned long long low)

  Insert High/low doublewords into a __int128 scalar.

Variables

- const vui128_t vtipowof10 [ ]

  table powers of 10 [0-38] in vector __int128 format.
- const vui128_t vtifrexpof10 [ ]

  table used to verify 128-bit frexp operations for powers of 10.
- const _Decimal128 decpowof2 [ ]

  table powers of 2 [0-1077] in _Decimal128 format.

7.3.1 Detailed Description

Common definitions and typedef used by the collection of Power Vector Library (pveclib) headers.

This includes:

- Typedefs as short names of common vector types.
- Union used to transfer 128-bit data between vector and non-vector types.
- Helper macros that make declaring constants and accessing elements a little easier.

7.3.2 Consistent vector type naming

Type names should be short, concise, and consistent. The ABI defines the vector types as extensions of the existing C Language types. So while *vector unsigned long long* is consistent it is neither short or concise. Pveclib uses the following naming convention for typedefs used in its operations, function prototypes, and internal variables.

- Starting with the **v** prefix for vector.
- followed by one of the element classes:

  - **i** for signed integer.
  - **ui** for unsigned integer.
  - **f** for floating-point.
  - **b** for bool.
- followed by the element size in bits:
  - 8, 16, 32, 64, 128
- Ending with the **_t** suffix signifying a typedef.

For example: vi32_t is a vector int, vui32_t is a vector unsigned int, vb32_t is a vector bool int, and vf32_t is vector float.

7.3.3 Transferring 128-bit types

The OpenPOWER ABI and the GCC compiler define a number of 128-bit scalar types that are not vector types:

- __int128 (a general purpose register pair)
- _Decimal128 (a floating-point even/odd register pair)
- __ibm128 (a floating-point register pair)
- __float128 (a vector register)

These are not cast nor assignment compatible with any vector type. However it may be useful to transfer to/from vector types for conversion or manipulation within an operation. For example:

- Conversions between __float128 and __int128, __ibm128, and _Decimal128 types.
- Conversions between vector BCD integers and __int128 and _Decimal128 types.
- Conversions between vector __int128 and __float128, __ibm128, and _Decimal128 types.

Here we use the __VEC_U_128 union to affect the transfer between the various types. We assume (fervently hope) that the compiler will recognize and optimize these as registers to registers transfers using the hardware instructions provided.

The vector to/from __float128 transfer should be the simplest as __float128 operations are defined over the vector register set. However __float128 types are defined in the PowerISA and OpenPOWER ABI, as scalars that just happens to use vector registers for parameter passing and operations. This distinction between scalars and vector prevents a direct cast between types. The __VEC_U_128 union is the simplest work around but in most cases no code should generated for this transfer. For example: vec_xfer_bin128_2_vui128t() and vec_xfer_vui128t_2_bin128().

Any vector to/from __int128 transfer requires a transfer between vector and general purpose registers. POWER8 (PowerISA 2.07B) added Move to/from Vector Scalar Register (mfvsr, mtvsr) instructions. Again the __VEC_U_128 union is used to effect the transfer and the compiler should leverage the move instructions in the generated code.

Any vector to/from __ibm128 or _Decimal128 requires a transfer between a pair of FPRs and a Vector Scalar Register (VSR). Technically this is transfer between the upper doubleword of two VSRs in the lower bank (VSR0-31) and another VSR. POWER7 (PowerISA 2.06B) provides the Permute Doubleword Immediate (xxpermdi) instruction. Again the __VEC_U_128 union is used to effect the transfer and the compiler should leverage the Permute Doubleword Immediate instructions in the generate code. For example: vec_BCD2DFP() and vec_DFP2BCD().

7.3.4 Endian and vector constants

Vector constants are often needed for: masking operations, range checks, permute selection, and radix conversion. Also compiler support for large integer and floating-point constants may be limited by the compiler. For example the GCC compilers support the (vector) __int128 type but do not directly support __int128 (39 digit) decimal constants. Another example is __float128 where the type and Q suffix constants are recent additions. In both cases we need to construct: large numeric constants, special values (infinity and NaN), masks for manipulating the sign bit and exponent bits. Often these values will be constructed from vectors of word or doubleword constants.

Note

> GCC does not support expressing an integer constant of type __int128 for targets where long long integer is less than 128 bits wide. This applies to the PowerPC target as the long long type is reserved for 64-bit integers. This was verified in GCC 8.2,
>
> GCC __float128 support for the PowerPC target began with GCC 6. In GCC 6 __float128 support is off by default and has to be explicitly enabled via the '-mfloat128' option. Starting with GCC 7, __float128 is enabled by default with VSX support.

Defining large constants for vectors is complicated by *little-endian* (LE) support as specified in the OpenPOWER ABI and as implemented in the compilers. Little-endian changes the effective vector element numbering and the order of constant elements in initializers. But the __int128 numerical order of magnitude or floating-point format does not change in registers. The high order bits are on the left and the low order bits are on the right.

So for example:

```
const vui32_t signmask = { 0x80000000, 0, 0, 0 };
const vui32_t expmask = { 0x7fff0000, 0, 0, 0 };
```

are correct sign and exponent masks for __float128 in big endian (BE) but would be incorrect for little endian (LE). To get correct results for both endians, one could code something like this:

```
#if __BYTE_ORDER__ == __ORDER_LITTLE_ENDIAN__
  const vui32_t signmask = { 0, 0, 0, 0x80000000 };
  const vui32_t expmask = { 0, 0, 0, 0x7fff0000 };
#else
  const vui32_t signmask = { 0x80000000, 0, 0, 0 };
  const vui32_t expmask = { 0x7fff0000, 0, 0, 0 };
#endif
```

But this gets tedious after the first dozen times. Also this can be confusing because it does not appear to the match the floating-point format diagrams in the PowerISA. The sign-bit and the exponent are always on the left.

So this header provides endian sensitive macros that maintain consistent "magnitude" order. For example:

```
const vui32_t signmask = CONST_VINT128_W (0x80000000, 0, 0, 0);
const vui32_t expmask = CONST_VINT128_W (0x7fff0000, 0, 0, 0);
```

This is always correct in either endian.

Another example; the multiplicative inverse for __int128 10**32 is 211857340822306639531405861550393824741. The GCC compiler will not accept this constant in a vector __int128 initializer. The next best thing would be

```
// The multiplicative inverse for 1 / 10**32 is
// 211857340822306639531405861550393824741
// or 0x9f623d5a8a732974cfbc31db4b0295e5
const vui128_t mulinv_10to32 =
         (vui128_t) CONST_VINT128_DW128 ( 0x9f623d5a8a732974UL,
                                          0xcfbc31db4b0295e5UL );
```

Here we use the CONST_VINT128_DW128 macro to maintain magnitude order across endian. Again the high order bits are on the left and the low order bits are on the right.

See also

> Endian problems with word operations
>
> General Endian Issues

7.3.5 Function Documentation

7.3.5.1 scalar_extract_uint64_from_high_uint128()

```
static unsigned long long scalar_extract_uint64_from_high_uint128 (
             unsigned __int128 gprp )  [inline], [static]
```

Extract the high doubleword from a __int128 scalar.

Parameters

gprp	a unsigned __int128 value.

Returns

The high doubleword of __int128.

7.3.5.2 scalar_extract_uint64_from_low_uint128()

```
static unsigned long long scalar_extract_uint64_from_low_uint128 (
             unsigned __int128 gprp )  [inline], [static]
```

Extract the low doubleword from a __int128 scalar.

Parameters

gprp	a unsigned __int128 value.

Returns

The low doubleword of __int128.

7.3.5.3 scalar_insert_uint64_to_uint128()

```
static unsigned __int128 scalar_insert_uint64_to_uint128 (
             unsigned long long high,
             unsigned long long low )  [inline], [static]
```

Insert High/low doublewords into a __int128 scalar.

Parameters

high	doubleword of a __int128.
low	doubleword of a __int128.

Returns

The combined quadword as a __int128 scalar.

7.3.5.4 vec_transfer_uint128_to_vui128t()

```
static vui128_t vec_transfer_uint128_to_vui128t (
            unsigned __int128 gprp )  [inline], [static]
```

Transfer a __int128 scalar to vector unsigned __int128.

The compiler does not allow direct transfer (assignment or type cast) between __int128 scalars and vector types. Vectors are held in 128-bit VRs (VSRs) and __int128 scalars are held in pair of 64-bit GPRs. So this operation requires a transfer between registers of different types/sizes.

processor	Latency	Throughput
power8	7	1/cycle
power9	5	1/cycle

Parameters

gprp	a unsigned __int128 value.

Returns

The original value returned as a vector unsigned__int128.

7.3.5.5 vec_transfer_vui128t_to_uint128()

```
static unsigned __int128 vec_transfer_vui128t_to_uint128 (
            vui128_t vra )  [inline], [static]
```

Transfer a vector unsigned __int128 to __int128 scalar.

The compiler does not allow direct transfer (assignment or type cast) between __int128 scalars and vector types. Vectors are held in 128-bit VRs (VSRs) and __int128 scalars are held in pair of 64-bit GPRs. So this operation requires a transfer between registers of different types/sizes.

processor	Latency	Throughput
power8	6-7	1/cycle
power9	5-6	2/cycle

Parameters

vra	a vector unsigned __int128 value.

Returns

The original value returned as a __int128 scalar.

7.4 src/pveclib/vec_f128_ppc.h File Reference

Header package containing a collection of 128-bit SIMD operations over Quad-Precision floating point elements.

```
#include <pveclib/vec_common_ppc.h>
#include <pveclib/vec_int128_ppc.h>
#include <pveclib/vec_f64_ppc.h>
```

Classes

- union __VF_128

  Union used to transfer 128-bit data between vector and __float128 types.

Typedefs

- typedef vui128_t vf128_t

  vector of 128-bit binary128 element. Same as __float128 for PPC.

- typedef vf128_t __Float128

  Define __Float128 if not defined by the compiler. Same as __float128 for PPC.

- typedef vf128_t __binary128

  Define __binary128 if not defined by the compiler. Same as __float128 for PPC.

- typedef vf128_t __float128

  Define __float128 if not defined by the compiler. Same as __float128 for PPC.

- typedef long double __IBM128

  Define __IBM128 if not defined by the compiler. Same as old long double for PPC.

Functions

- static vui64_t vec_const64_f128_128 (void)

  Generate doubleword splat constant 128.
- static vui32_t vec_const128_f128_128 (void)

  Generate Quadword constant 128.
- static vui64_t vec_mask64_f128exp (void)

  Generate Doubleword Quad-Precision exponent mask.
- static vui32_t vec_mask128_f128exp (void)

  Generate Quadword Quad-Precision exponent mask.
- static vui32_t vec_mask128_f128mag (void)

  Generate Quadword Quad-Precision magnitude mask.
- static vui32_t vec_mask128_f128sig (void)

  Generate Quadword Quad-Precision significand mask.
- static vui32_t vec_mask128_f128sign (void)

  Generate Quadword Quad-Precision Sign-bit mask.
- static vui32_t vec_mask128_f128Cbit (void)

  Generate Quadword C-bit mask Immediate.
- static vui32_t vec_mask128_f128Lbit (void)

  Generate Quadword L-bit mask Immediate.
- static vui32_t vec_mask128_f128Qbit (void)

  Generate Quadword QNaN-bit mask Immediate.
- static __binary128 vec_sel_bin128_2_bin128 (__binary128 vfa, __binary128 vfb, vb128_t mask)

  Select and Transfer from one of two __binary128 scalars under a 128-bit mask. The result is a __binary128 of the selected value.
- static vui32_t vec_and_bin128_2_vui32t (__binary128 f128, vui32_t mask)

  Transfer a quadword from a __binary128 scalar to a vector int and logical AND with a mask.
- static vui32_t vec_andc_bin128_2_vui32t (__binary128 f128, vui32_t mask)

  Transfer a quadword from a __binary128 scalar to a vector int and logical AND Compliment with mask.
- static vui32_t vec_or_bin128_2_vui32t (__binary128 f128, vui32_t mask)

  Transfer a quadword from a __binary128 scalar to a vector int and logical OR with mask.
- static vui32_t vec_xor_bin128_2_vui32t (__binary128 f128, vui32_t mask)

  Transfer a quadword from a __binary128 scalar to a vector int and logical Exclusive OR with mask.
- static vui128_t vec_andc_bin128_2_vui128t (__binary128 f128, vui128_t mask)

  Transfer a quadword from a __binary128 scalar to a vector __int128 and logical AND Compliment with mask.
- static vui8_t vec_xfer_bin128_2_vui8t (__binary128 f128)

  Transfer function from a __binary128 scalar to a vector char.
- static vui16_t vec_xfer_bin128_2_vui16t (__binary128 f128)

  Transfer function from a __binary128 scalar to a vector short int.
- static vui32_t vec_xfer_bin128_2_vui32t (__binary128 f128)

  Transfer function from a __binary128 scalar to a vector int.
- static vui64_t vec_mrgh_bin128_2_vui64t (__binary128 vfa, __binary128 vfb)

  Merge High and Transfer function from a pair of __binary128 scalars to a vector long long int.
- static vui64_t vec_mrgl_bin128_2_vui64t (__binary128 vfa, __binary128 vfb)

  Merge Low and Transfer function from a pair of __binary128 scalars to a vector long long int.
- static vui64_t vec_xfer_bin128_2_vui64t (__binary128 f128)

  Transfer function from a __binary128 scalar to a vector long long int.

- static vui128_t vec_xfer_bin128_2_vui128t (__binary128 f128)

  Transfer function from a __binary128 scalar to a vector __int128.
- static __binary128 vec_xfer_vui8t_2_bin128 (vui8_t f128)

  Transfer a vector unsigned char to __binary128 scalar.
- static __binary128 vec_xfer_vui16t_2_bin128 (vui16_t f128)

  Transfer a vector unsigned short to __binary128 scalar.
- static __binary128 vec_xfer_vui32t_2_bin128 (vui32_t f128)

  Transfer a vector unsigned int to __binary128 scalar.
- static __binary128 vec_xfer_vui64t_2_bin128 (vui64_t f128)

  Transfer a vector unsigned long long to __binary128 scalar.
- static __binary128 vec_xfer_vui128t_2_bin128 (vui128_t f128)

  Transfer a vector unsigned __int128 to __binary128 scalar.
- static __binary128 vec_absf128 (__binary128 f128)

  Absolute Quad-Precision.
- static int vec_all_isfinitef128 (__binary128 f128)

  Return true if the __float128 value is Finite (Not NaN nor Inf).
- static int vec_all_isinff128 (__binary128 f128)

  Return true if the __float128 value is infinity.
- static int vec_all_isnanf128 (__binary128 f128)

  Return true if the __float128 value is Not a Number (NaN).
- static int vec_all_isnormalf128 (__binary128 f128)

  Return true if the __float128 value is normal (Not NaN, Inf, denormal, or zero).
- static int vec_all_issubnormalf128 (__binary128 f128)

  Return true if the __float128 value is subnormal (denormal).
- static int vec_all_isunorderedf128 (__binary128 vfa, __binary128 vfb)

  Return true if either __float128 value (vra, vrb) is NaN.
- static int vec_all_iszerof128 (__binary128 f128)

  Return true if the __float128 value is +-0.0.
- static __binary128 vec_copysignf128 (__binary128 f128x, __binary128 f128y)

  Copy the sign bit from f128x and merge with the magnitude from f128y. The merged result is returned as a __float128 value.
- static __binary128 vec_const_huge_valf128 ()

  return a positive infinity.
- static __binary128 vec_const_inff128 ()

  return a positive infinity.
- static __binary128 vec_const_nanf128 ()

  return a quiet NaN.
- static __binary128 vec_const_nansf128 ()

  return a signaling NaN.
- static vb128_t vec_cmpeqtoqp (__binary128 vfa, __binary128 vfb)

  Vector Compare Equal (Total-order) Quad-Precision.
- static vb128_t vec_cmpequzqp (__binary128 vfa, __binary128 vfb)

  Vector Compare Equal (Zero-unordered) Quad-Precision.
- static vb128_t vec_cmpequqp (__binary128 vfa, __binary128 vfb)

  Vector Compare Equal (Unordered) Quad-Precision.
- static vb128_t vec_cmpgetoqp (__binary128 vfa, __binary128 vfb)

  Vector Compare Greater Than or Equal (Total-order) Quad-Precision.

- static vb128_t vec_cmpgeuzqp (__binary128 vfa, __binary128 vfb)

  Vector Compare Greater Than Or Equal (Zero-unordered) Quad-Precision.
- static vb128_t vec_cmpgeuqp (__binary128 vfa, __binary128 vfb)

  Vector Compare Greater Than or Equal (Unordered) Quad-Precision.
- static vb128_t vec_cmpgttoqp (__binary128 vfa, __binary128 vfb)

  Vector Compare Greater Than (Total-order) Quad-Precision.
- static vb128_t vec_cmpgtuzqp (__binary128 vfa, __binary128 vfb)

  Vector Compare Greater Than (Zero-unordered) Quad-Precision.
- static vb128_t vec_cmpgtuqp (__binary128 vfa, __binary128 vfb)

  Vector Compare Greater Than (Unordered) Quad-Precision.
- static vb128_t vec_cmpletoqp (__binary128 vfa, __binary128 vfb)

  Vector Compare Less Than or Equal (Total-order) Quad-Precision.
- static vb128_t vec_cmpleuzqp (__binary128 vfa, __binary128 vfb)

  Vector Compare Less Than or Equal (Zero-unordered) Quad-Precision.
- static vb128_t vec_cmpleuqp (__binary128 vfa, __binary128 vfb)

  Vector Compare Less Than or Equal (Unordered) Quad-Precision.
- static vb128_t vec_cmplttoqp (__binary128 vfa, __binary128 vfb)

  Vector Compare Less Than (Total-order) Quad-Precision.
- static vb128_t vec_cmpltuzqp (__binary128 vfa, __binary128 vfb)

  Vector Compare Less Than (Zero-unordered) Quad-Precision.
- static vb128_t vec_cmpltuqp (__binary128 vfa, __binary128 vfb)

  Vector Compare Less Than (Unordered) Quad-Precision.
- static vb128_t vec_cmpnetoqp (__binary128 vfa, __binary128 vfb)

  Vector Compare Not Equal (Total-order) Quad-Precision.
- static vb128_t vec_cmpneuzqp (__binary128 vfa, __binary128 vfb)

  Vector Compare Not Equal (Zero-unordered) Quad-Precision.
- static vb128_t vec_cmpneuqp (__binary128 vfa, __binary128 vfb)

  Vector Compare Not Equal (Unordered) Quad-Precision.
- static int vec_cmpqp_all_toeq (__binary128 vfa, __binary128 vfb)

  Vector Compare all Equal (Total-order) Quad-Precision.
- static int vec_cmpqp_all_uzeq (__binary128 vfa, __binary128 vfb)

  Vector Compare all Equal (Zero-unordered) Quad-Precision.
- static int vec_cmpqp_all_eq (__binary128 vfa, __binary128 vfb)

  Vector Compare all Equal (Unordered) Quad-Precision.
- static int vec_cmpqp_all_toge (__binary128 vfa, __binary128 vfb)

  Vector Compare all Greater Than Or Equal (Total-order) Quad-Precision.
- static int vec_cmpqp_all_uzge (__binary128 vfa, __binary128 vfb)

  Vector Compare all Greater Than Or Equal (Zero-unordered) Quad-Precision.
- static int vec_cmpqp_all_ge (__binary128 vfa, __binary128 vfb)

  Vector Compare all Greater Than Or Equal (Unordered) Quad-Precision.
- static int vec_cmpqp_all_togt (__binary128 vfa, __binary128 vfb)

  Vector Compare all Greater Than (Total-order) Quad-Precision.
- static int vec_cmpqp_all_uzgt (__binary128 vfa, __binary128 vfb)

  Vector Compare all Greater Than (Zero-unordered) Quad-Precision.
- static int vec_cmpqp_all_gt (__binary128 vfa, __binary128 vfb)

  Vector Compare all Greater Than (Unordered) Quad-Precision.
- static int vec_cmpqp_all_tole (__binary128 vfa, __binary128 vfb)

Vector Compare All Less Than Or Equal (Total-order) Quad-Precision.

- static int vec_cmpqp_all_uzle (__binary128 vfa, __binary128 vfb)

  Vector Compare all Less Than Or Equal (Zero-unordered) Quad-Precision.
- static int vec_cmpqp_all_le (__binary128 vfa, __binary128 vfb)

  Vector Compare all Less Than Or Equal (Unordered) Quad-Precision.
- static int vec_cmpqp_all_tolt (__binary128 vfa, __binary128 vfb)

  Vector Compare All Less Than (Total-order) Quad-Precision.
- static int vec_cmpqp_all_uzlt (__binary128 vfa, __binary128 vfb)

  Vector Compare all Less Than (Zero-unordered) Quad-Precision.
- static int vec_cmpqp_all_lt (__binary128 vfa, __binary128 vfb)

  Vector Compare all Less Than (Unordered) Quad-Precision.
- static int vec_cmpqp_all_tone (__binary128 vfa, __binary128 vfb)

  Vector Compare all Not-Equal (Total-order) Quad-Precision.
- static int vec_cmpqp_all_uzne (__binary128 vfa, __binary128 vfb)

  Vector Compare all Not-Equal (Zero-unordered) Quad-Precision.
- static int vec_cmpqp_all_ne (__binary128 vfa, __binary128 vfb)

  Vector Compare all Not-Equal (Unordered) Quad-Precision.
- static int vec_cmpqp_exp_eq (__binary128 vfa, __binary128 vfb)

  Vector Compare Quad-Precision Exponents for Equal.
- static int vec_cmpqp_exp_gt (__binary128 vfa, __binary128 vfb)

  Vector Compare Exponents Quad-Precision for Greater Than.
- static int vec_cmpqp_exp_lt (__binary128 vfa, __binary128 vfb)

  Vector Compare Exponents Quad-Precision for Less Than.
- static int vec_cmpqp_exp_unordered (__binary128 vfa, __binary128 vfb)

  Vector Compare Exponents Quad-Precision for Unordered.
- static vb128_t vec_isfinitef128 (__binary128 f128)

  Return 128-bit vector boolean true if the __float128 value is Finite (Not NaN nor Inf).
- static int vec_isinf_signf128 (__binary128 f128)

  Return true (nonzero) value if the __float128 value is infinity. If infinity, indicate the sign as +1 for positive infinity and -1 for negative infinity.
- static vb128_t vec_isinff128 (__binary128 f128)

  Return a 128-bit vector boolean true if the __float128 value is infinity.
- static vb128_t vec_isnanf128 (__binary128 f128)

  Return 128-bit vector boolean true if the __float128 value is Not a Number (NaN).
- static vb128_t vec_isnormalf128 (__binary128 f128)

  Return 128-bit vector boolean true if the __float128 value is normal (Not NaN, Inf, denormal, or zero).
- static vb128_t vec_issubnormalf128 (__binary128 f128)

  Return 128-bit vector boolean true value, if the __float128 value is subnormal (denormal).
- static vb128_t vec_isunorderedf128 (__binary128 vfa, __binary128 vfb)

  Return 128-bit vector boolean true value, if either __float128 value (vra, vrb) is NaN.
- static vb128_t vec_iszerof128 (__binary128 f128)

  Return 128-bit vector boolean true value, if the value that is +-0.0.
- static __binary128 vec_nabsf128 (__binary128 f128)

  Negative Absolute value Quad-Precision.
- static __binary128 vec_negf128 (__binary128 f128)

  Negate the sign bit of a __float128 input and return the resulting __float128 value.
- static __binary128 vec_self128 (__binary128 vfa, __binary128 vfb, vb128_t mask)

Select and Transfer from one of two __binary128 scalars under a 128-bit mask. The result is a __binary128 of the selected value.

- static vb128_t vec_setb_qp (__binary128 f128)

  Vector Set Bool from Quadword Floating-point.
- static int vec_signbitf128 (__binary128 f128)

  Return int boolean true if the __float128 value is negative (sign bit is '1').
- static __binary128 vec_xsaddqpo (__binary128 vfa, __binary128 vfb)

  VSX Scalar Add Quad-Precision using round to Odd.
- static __binary128 vec_xssubqpo (__binary128 vfa, __binary128 vfb)

  VSX Scalar Subtract Quad-Precision using round to Odd.
- static __binary128 vec_xscvdpqp (vf64_t f64)

  VSX Scalar Convert Double-Precision to Quad-Precision format.
- static vf64_t vec_xscvqpdpo (__binary128 f128)

  VSX Scalar Convert with round Quad-Precision to Double-Precision (using round to odd).
- static vui64_t vec_xscvqpudz (__binary128 f128)

  VSX Scalar Convert with round to zero Quad-Precision to Unsigned doubleword.
- static vui128_t vec_xscvqpuqz (__binary128 f128)

  VSX Scalar Convert with round to zero Quad-Precision to Unsigned Quadword.
- static __binary128 vec_xscvsdqp (vi64_t int64)

  VSX Scalar Convert Signed-Doubleword to Quad-Precision format.
- static __binary128 vec_xscvudqp (vui64_t int64)

  VSX Scalar Convert Unsigned-Doubleword to Quad-Precision format.
- static __binary128 vec_xscvsqqp (vi128_t int128)

  VSX Scalar Convert Signed-Quadword to Quad-Precision format.
- static __binary128 vec_xscvuqqp (vui128_t int128)

  VSX Scalar Convert Unsigned-Quadword to Quad-Precision format.
- static __binary128 vec_xsmulqpo (__binary128 vfa, __binary128 vfb)

  VSX Scalar Multiply Quad-Precision using round to Odd.
- static __binary128 vec_xsiexpqp (vui128_t sig, vui64_t exp)

  Scalar Insert Exponent Quad-Precision.
- static vui64_t vec_xsxexpqp (__binary128 f128)

  Scalar Extract Exponent Quad-Precision.
- static vui128_t vec_xsxsigqp (__binary128 f128)

  Scalar Extract Significand Quad-Precision.
- static vui64_t vec_xxxexpqpp (__binary128 vfa, __binary128 vfb)

  Vector Extract Exponent Quad-Precision Pair.

7.4.1 Detailed Description

Header package containing a collection of 128-bit SIMD operations over Quad-Precision floating point elements.

PowerISA 3.0B added Quad-Precision floating point type and operations to the Vector-Scalar Extension (VSX) facility. The first hardware implementation is available in POWER9.

PowerISA 3.1 added new min/max/compare Quad-Precision operations. Also added new quadword (128-bit) integer operations including converts between quadword integer and Quad-Precision floating point. The first hardware implementation is available in POWER10.

While all Quad-Precision operations are on 128-bit vector registers, they are defined as scalars in the PowerISA. The OpenPOWER ABI also treats the __float128 type as scalar that just happens to use vector registers for parameter passing and operations. As such no operations using __float128 (_Float128, or __ieee128) as parameter or return value are defined as vector built-ins in the ABI or <altivec.h>.

Note

> GCC 8.2 does document some built-ins, using the *scalar* prefix (scalar_extract_exp, scalar_extract_sig, scalar_↩ test_data_class), that do accept the __ieee128 type. This work seems to be incomplete as scalar_exp_cmp_* for the __ieee128 type are not present. GCC 7.3 defines vector and scalar forms of the extract/insert_exp for float and double but not for __ieee128. These built-ins are not defined in GCC 6.4. See `compiler documentation`. These are useful operations and can be implemented in a few vector logical instructions for earlier machines. So it seems reasonable to add these to pveclib for both vector and scalar forms.

Quad-Precision is not supported in hardware until POWER9. However the compiler and runtime supports the __float128 type and arithmetic operations via soft-float emulation for earlier processors. The soft-float implementation follows the ABI and passes __float128 parameters and return values in vector registers.

Note

> The performance of the libgcc soft-float (KF Mode) runtime for __float128 varies between releases of GCC. Some GCC versions introduced a bug that miss-compiled the transfer of __float128 parameters to GPR pairs as required for generic soft-float implementation. For POWER8 this results in a significant performance hit.

The PowerISA 3.0 also defines a number of useful quad-precision operations using the "round-to-odd" override. This is useful when the results of quad-precision arithmetic must be rounded to a shorter precision while avoiding double rounding. Recent GCC compilers support these operations as built-ins for the POWER9 target, but they not supported by the C language or GCC runtime library. This means that round-to-odd is not easily available to libraries that need to support IEEE-128 on POWER8. Again it may be reasonable to add these to pveclib.

Note

> See `GCC Avoids Double Rounding Errors With Round-To-Odd`

Another issue is the performance of GCC soft-float runtime for IEEE-128 (KF mode). There seem to be a number of issues with code generation for transfers from __float128 to 64-bit integer GPRs. This is required to match the ABI (vector) parameters to the soft-float runtime using integer scalars. For POWER8 targets the GCC compiler generates store vector followed by two load doubleword instructions. This generates high frequencies of load-hit-store rejects at runtime. It also looks like there is significant instruction latency associated with the XER carry bit required for extended (128-bit) integer arithmetic.

Note

> The first issue (parameter store reload) is a compiler bug. The second issue is an attribute of the hardware implementation and associate trade-offs. Both of these issues can be avoided by providing a soft-float implementation for __float128 using VSX vector 128-bit arithmetic and logical operations.

For the QP operations that have libgcc implementations and we have corresponding PVECLIB implementations we can do direct performance comparisons. So far micro-benchmarks show a significant performance gain for the PVECLIB vector implementations vs the GCC KF mode runtime.

Power8 QP	cmpqp	cvtdpqp	cvtqpdp	cvtuqqp	cvtqpuq	mulqp	addqp
%improvement	22.4	60.7	46.2*	28.9	72.4	1.8*	10.1*

Note

> These micro-benchmarks and others are included in src/testsuite. Specifically files vec_perf_f128.c and vec_f128_dummy.c. Make check will compile the appropriate files and link them into the pveclib_perf executable. Items marked with "*" compare the pveclib round-to-odd implementation to the soft-float default rounding mode. The libgcc soft-float runtime does not implement round-to-odd.

There are number of __float128 operations that should generate a single instruction for POWER9 and few (less than 10) instructions for POWER8. This includes all of the __float128 classification functions (isnormal/subnormal/finite/inf/nan/zero). Unfortunately for POWER8 the compilers will generate calls to the GCC runtime (__unordkf2, __gekf2, ...) for these functions. In many cases the code size generated for the runtime calls far exceeds any in-line VSX code PVECLIB will generate.

So it is not unreasonable for this header to provide vector forms of the __float128 classification functions (isnormal/subnormal/finite/inf/nan/zero). It is little additional effort to include the sign bit manipulation operations (copysign, abs, nabs, and neg).

These functions can be implemented directly using (one or more) POWER9 instructions, or a few vector logical and integer compare instructions for POWER7/8. Each is comfortably small enough to be in-lined and inherently faster than the equivalent POSIX or compiler built-in runtime functions. Performing these operations in-line and directly in vector registers (VRs) avoids call/return and VR <-> GPR transfer overhead. It also seems reasonable to provide Quad-Precision extract/insert exponent/significand and compare exponent operations for POWER7/8.

The PVECLIB implementations for quad-precision arithmetic and conversion operations are large enough that most applications will want to call a library. PVECLIB will build and release the appropriate CPU tuned libraries. This will follow the general design used for multiple quadword integer multiply functions (vec_int512_ppc.h).

Note

> At this time, PVECLIB does not intend to replace existing GCC/libm IEEE-128 runtime APIs and will maintain it own unique name-space. However if the maintainers of these projects want to leverage PVECLIB they are allowed under the terms of the `Apache License, Version 2.0.`

These PVECLIB operations should be useful for applications using Quad-Precision while needing to still support POWER8 but also build for POWER9/10. An important goal is to allow applications and libraries to safely substitute PVECLIB operations for C language and math.h __float128 operators and functions as point optimizations. The largest gains will be seen for builds targeting POWER8 without degrading performance when targeting POWER9/10. They should also be useful and improve performance of soft-float implementations of Quad-Precision math library functions.

Note

> The compiler disables associated <altivec.h> built-ins if the **mcpu** target does not enable the specific instruction. For example if you compile with **-mcpu=power8**, Quad-Precision floating-point built-ins operations useful for floating point classification are not defined. This header provides the appropriate substitutions, will generate the minimum code, appropriate for the target, and produce correct results.
>
> Most ppc64le compilers will default to **-mcpu=***power8* if **-mcpu** is not specified.

This header covers operations that are any of the following:

- Implemented in hardware instructions in newer processors, but useful to programmers on slightly older processors (even if the equivalent function requires more instructions).

- Defined in the OpenPOWER ABI but *not* yet defined in <altivec.h> provided by available compilers in common use. Examples include scalar_test_neg, scalar_test_data_class, etc.
- Defined for POWER9 (as built-ins) but not supported in the soft-float runtime implementation provided for POWER8. Examples include the arithmetic/conversion operations supporting the *round-to-odd* override.
- Providing vector float tests for special conditions without generating extraneous floating-point exceptions. This is important for implementing __float128 forms of ISO C99 Math functions. Examples include vector isnan, isinf, etc.
- Commonly used operations, not covered by the ABI or <altivec.h>, and require multiple instructions or are not obvious.

7.4.2 Vector implementation of Quad-Precision Soft-float

The discussion above raises a interesting question. If we can provide useful implementations of Quad-Precision; classification, extract/insert, and compare exponent operations, why not continue with Quad-Precision compare, convert to/from integer, and arithmetic operations?

This raises the stakes in complexity and size of implementation. Providing a vector soft-float implementation equivalent to the GCC run-time libgcc **__addkf3**/**__divkf3**/**__mulkf3**/**__subkf3** would be a substantial effort. The IEEE standard is exacting about rounding and exception handling. Comparisons require special handling of; signed zero, infinities, and NaNs. Even float/integer conversions require correct rounding and return special values for overflow. Also it is not clear how such an effort would be accepted.

The good news is PVECLIB already provides a strong quadword integer operations set. Integer Add, subtract, and multiply are covered with the usual compare/shift/rotate operations (See vec_int128_ppc.h and vec_int64_ppc.h). The weak spot is general quadword integer divide. Until recently, integer divide has not been part of the vector ISA. But the introduction of Vector Divide Signed/Unsigned Quadword in POWER10 raises the priority of vector integer divide for PVECLIB.

For now we propose a phased approach, starting with enablers and infrastructure, building up layers, starting simple and adding complexity.

- Basic enablers; classification, extract/insert exponent, compare exponent.
- Quad-Precision comparison operators.
  - Initially ignore special cases and exceptions
  - Add Signed Zero, Infinity. and NaN special cases
  - Exceptions (FPSCR) when someone asks
- Quad-Precision from/to integer word/doubleword/quadword.
  - Cases that don't require rounding (i.e truncate and DW to QP).
  - Cases that require rounding
    - Round to odd.
    - Round to Nearest/Even
    - Others if asked
- Quad-Precision arithmetic
  - Add/Sub/Mul
    - Round-to-Odd first

  - Fused Multiply-Add
    - Round-to-Odd first
  - Divide
    - Round-to-Odd first
    - Will need vec_int128_ppc.h implementation of vec_divuq()
  - Signed Zero, Infinity. and NaN special cases
  - Other rounding modes
  - Exceptions (FPSCR) when someone asks

The intent is that such PVECLIB operations can be mixed in with or substituted for C Language _FLoat128 expressions or functions. The in-lined operations should have performance advantages over equivalent library functions on both POWER8/9.

This is a big list. It is TBD how far I will get given my current limited resources.

Note

> We are focusing on POWER8 here because the implementation gets a lot harder for POWER7 and earlier. POWER7 is missing:
>
> - Vector Quadword integer add/sub with carry extend.
> - Vector Doubleword integer arithmetic, compares. and count-leading zeros.
> - Vector Word integer multiply.
> - Direct transfer between VRs and GPRs.

7.4.2.1 Quad-Precision data class and exponent access for POWER8

Most math library functions need to test the data class (normal, infinity, NaN, etc) and or range of input values. This usually involves separating the sign, exponent, and significand out from __float128 values, and comparing one or more of these parts, to special integer values.

PowerISA 3.0B (POWER9) provides instructions for these in addition to a comprehensive set of arithmetic and compare instructions. These operations are also useful for the soft-float implementation of __float128 for POWER8 and earlier. The OpenPOWER ABI specifies __float128 parameters are in VRs and are immediately accessible to VMX/VSR instructions. This is important as the cost of transferring values between VRs and GPRs is quite high on POWER8 and even higher for POWER7 and earlier (which requires store to temporaries and reload).

Fortunately these operations only require logical (and/or/xor), shift and integer compare operations to implement. These are available as vector intrinsics or provides by PVECLIB (see vec_int128_ppc.h).

The operations in this group include:

- Altivec like predicates; vec_all_isfinitef128(), vec_all_isinff128(), vec_all_isnanf128(), vec_all_isnormalf128(), vec_all_issubnormalf128(), vec_all_iszerof128(), vec_signbitf128().
- Vector boolean predicates; vec_isfinitef128(), vec_isinff128(), vec_isnanf128(), vec_isnormalf128(), vec_issubnormalf128(), vec_iszerof128(), vec_setb_qp().
- Sign bit manipulation; vec_absf128(), vec_nabsf128(), vec_negf128(), vec_copysignf128().

- Data manipulation; vec_xsiexpqp(), vec_xsxexpqp(), vec_xxxexpqpp(), vec_xsxsigqp().
- Exponent Compare; vec_cmpqp_exp_eq(), vec_cmpqp_exp_gt(), vec_cmpqp_exp_lt(), vec_cmpqp_exp_unordered().

For example the data class test isnan:

```
static inline vb128_t
vec_isnanf128 (__binary128 f128)
{
#if defined (_ARCH_PWR9) && defined (scalar_test_data_class) && \
  defined (__FLOAT128__) && (__GNUC__ > 7)
  vui32_t result = CONST_VINT128_W (0, 0, 0, 0);
  if (scalar_test_data_class (f128, 0x40))
    result = CONST_VINT128_W (-1, -1, -1, -1);
  return (vb128_t)result;
#else
  vui32_t tmp;
  const vui32_t signmask = CONST_VINT128_W (0x80000000, 0, 0, 0);
  const vui32_t expmask = CONST_VINT128_W (0x7fff0000, 0, 0, 0);
  tmp = vec_andc_bin128_2_vui32t (f128, signmask);
  return vec_cmpgtuq ((vui128_t)tmp , (vui128_t)expmask);
#endif
}
```

Which has implementations for POWER9 (and later) and POWER8 (and earlier).

For POWER9 it generates:

```
xststdcqp cr0,v2,64
bne      .+12
xxspltib vs34,255
b        .+8
xxspltib vs34,0
```

Which uses the intrinsic scalar_test_data_class() to generate the VSX Scalar Test Data Class Quad-Precision instruction with "data class mask" of *class.NaN* to set the condition code. If the condition is *match*, load the 128-bit bool value of all 1's (true). Otherwise load all 0's (false).

For POWER8 it generates

```
addis    r9,r2,@ha.rodata.cst16+0x30
vspltisw v1,-1
vspltisw v12,0
addi     r9,r9,@l.rodata.cst16+0x30
vslw     v1,v1,v1
lvx      v0,0,r9
vsldoi   v1,v1,v12,12
xxlandc vs33,vs34,vs33
vsubcuq v0,v0,v1
xxspltw vs32,vs32,3
vcmpequw v2,v0,v12
```

The first 7 instructions above, load the constant vectors needed by the logic. These constants only need to be generated once per function and can be shared across operations.

In the C code we use a special transfer function combined with logical AND complement (vec_andc_bin128_2_vui32t()). This is required because while __float128 values are held in VRs, the compiler considers them to be scalars and will not allow simple casts to (any) vector type. So the PVECLIB implementation provides *xfer* function using a union to transfer the __float128 value to a vector type. In most case this logical transfer simply serves to make the compiler happy and does not need to generate any code. In this case the *xfer* function combines the transfer with a vector and complement to mask off the sign bit.

Then compare the masked result as a 128-bit integer value greater than infinity (expmask). Here we use the vec_cmpgtuq() operation from vec_int128_ppc.h. For POWER8, vec_cmpgtuq() generates the Vector Subtract and Write Carry Unsigned Quadword instruction for 128-bit unsigned compares. A '0' carry indicates greater than. The next two instructions (from vec_setb_ncq()) convert the carry bit to the required 128-bit bool value.

While the POWER8 sequence requires more instructions (including the const vector set up) than POWER9, it is not significantly larger. And as mentioned above, the set-up code can be optimized across operations sharing the same constants. The code (less the setup) is only 10 cycles for POWER8 vs 6 for POWER9. Also the code is not any larger than the function call overhead for the libgcc runtime equivalent **__unordkf2**. And is much faster then the generic soft-float implementation.

Another example, Scalar Extract Exponent Quad-Precision:

```
static inline vui64_t
vec_xsxexpqp (__binary128 f128)
{
  vui64_t result;
#if defined (_ARCH_PWR9) && defined (__FLOAT128__) && (__GNUC__ > 7)
  __asm__(
      "xsxexpqp %0,%1"
      : "=v" (result)
      : "v" (f128)
      : );
#else
  vui32_t tmp;
  const vui32_t expmask = CONST_VINT128_W (0x7fff0000, 0, 0, 0)
  tmp = vec_and_bin128_2_vui32t (f128, expmask);
  result = (vui64_t) vec_sld (tmp, tmp, 10);
#endif
  return result;
}
```

Which has implementations for POWER9 (and later) and POWER8 (and earlier).

For POWER9 it generates the VSX Scalar Extract Exponent Quad-Precision instruction.

```
xsxexpqp v2,v2
```

Note

> Would like to use the intrinsic scalar_extract_exp() here but this is not available until GCC 11 (or later). Also GCC defines these scalar built-ins to return integer scalar values in GPRs. This would defeat the purpose of an all vector implementation.

For POWER8 we generate

```
addis   r9,r2,.rodata.cst16+0xc0@ha
addi    r9,r9,.rodata.cst16+0xc0@l
lvx     v13,0,r9
xxland  vs34,vs34,vs45
vsldoi  v2,v2,v2,10
```

The first 3 instructions above load the constant vector needed by the logic. This constant only needs to be generated once per function and can be shared across operations.

Again we use a special transfer function combined with logical AND (vec_and_bin128_2_vui32t()) to transfer the__float128 to a vector type and mask off all bits except for the 15-bit exponent. Then we rotate the exponent logically right 48-bit to right justify the exponent in vector doubleword 0. This matches the results of the xsxexpqp instruction.

7.4.2.2 Intermediate results and Rounding for Quad-Precision

The IEEE-128 floating-point storage (external) format fits neatly in 128-bits. But this compact format needs to be expanded internally during QP operations. The sign and exponent are normally manipulated separately from the significand. And for finite values the Leading-bit (implied but not included in the storage format) must be restored to take part in arithmetic/rounding/normalization operations.

Note

> The *Leading*, *Hidden*, and *Implicit* bits are different names for the most significant bit of the significand. This bit is Hidden or Implicit only for the external or storage format of floating-point numbers. But it needs to be explicitly represented in the internal *Intermediate Results (**IR**)*. See Also: PowerISA 3.0 7.3.3 VSX Floating-Point Execution Models.

For a soft-float implementation of IEEE-128 on POWER8 we want to extract these components into 128-bit vector registers and operate on them using vector instructions. This allows direct use of 128-bit arithmetic/shift/rotate operations (see vec_int128_ppc.h), while avoiding expensive transfers between VRs and GPRs.

To extract the sign-bit we can either AND with a 128-bit mask or use a set-bool operation (vec_setb_qp() or vec_setb_sq()). The masked sign-bit can be ORed with the final IEEE-128 vector result to set the appropriate sign. The 128-bit vector bool can be used with vec_sel() (vec_self128(), vec_selsq(), vec_seluq()) to select results based on the sign-bit while avoiding branch logic.

We use vec_xsxexpqp() to extract the 15-bit exponent into a vector doubleword integer element. The biased exponent is returned in the high doubleword (matching the POWER9 instruction). Depending on the operation, the exponent (or derived values) may need to be transfered/replicated to the low doubleword element. This is easily accomplished using vec_splatd(). Operations requiring two Quad-precision operands can combine the extracted exponents into a single vector doubleword using vec_mrgahd().

Note

> Operations requiring two Quad-precision operands have an opportunity to vectorize the exponent extraction and comparisons (all/any) for finite vs NaN/Infinity. See vec_xxxexpqpp().

We use vec_xsxsigqp() to extract the 113-bit significand into a vector quadword integer. This operation restores the leading-bit for normal (not NaN, Infinity, denormal or zero) values. The significand is returned right-justified in the quadword.

At the end of the operation we can use vec_or() and vec_xsiexpqp() to combine these (sign, exponent, and significand) components into a IEEE-128 result.

7.4.2.2.1 Representing Intermediate results for Quad-Precision

Internal IEEE floating-point operations will need/generate additional bits to support normalization and rounding. The PowerISA describes a **VSX Execution Model for IEEE Operations**

See also

> PowerISA 3.0B, Book I, Section 7.3.3 VSX Floating-Point Execution Models.

IEEE quad-precision execution model

		0	**1 ---------------------------— 112**			
S	C	L	FRACTION	G	R	X
- Sign bit						
- Carry bit						
- Leading bit, also called the implicit or hidden bit						
- Fraction (112-bits)						
- Guard bit						
- Round bit						
- (X) AKA Sticky bit, logical OR of remaining bits						

This model is a guide for processor design and soft-float implementors. This is also described as the *Intermediate result Representation (**IR**)*. As such the implementation may arrange these bits into different registers as dictated by design and performance.

The GRX bits extend the low order bits of the fraction and are required for rounding. Basically these bits encode how *near* the intermediate result is to a representable result. The GR bits are required for post-normalization of the result and participate in shifts during normalization. For right shifts, bits shift out of the R-bit are logically ORed into the X-bit. For left shifts, 0 bits shifted into the R-bit (the X-bit is ignored).

As mentioned before, it is convenient to keep the sign-bit in a separate vector quadword. This not an extension of the significand but is needed to select results for arithmetic and some rounding modes. The remaining (C through X) bits can be represented in a vector quadword register or a vector register pair.

7.4.2.2.1.1 IR for conversion

For example integer to QP conversions can be represented in a vector quadword by left justifying the magnitude before normalization and rounding. For example from vec_xscvuqqp():

```
{ // We need to produce a normal QP, so we treat the integer like a
  // denormal, then normalize it.
  // Start with the quad exponent bias + 127 then subtract the count of
  // leading '0's. The 128-bit significand can have 0-127 leading '0's.
  vui64_t q_expm = (vui64_t) CONST_VINT64_DW (0, (0x3fff + 127));
  vui64_t i64_clz = (vui64_t) vec_clzq (q_sig);
  q_sig = vec_slq (q_sig, (vui128_t) i64_clz);
  q_exp = vec_subudm (q_expm, i64_clz);
  // This is the part that might require rounding.
  // The Significand (including the L-bit) is right justified in
  // in the high-order 113-bits of q_sig.
  // The guard, round, and sticky (GRX) bits are in the low-order
  // 15 bits.
  ...
}
```

See Examples for Round to Nearest Even and Examples for Round toward Zero.

The *Round to Nearest Even* case may increment the significand and that may generate a carry from the **L-bit**. One option is to use vec_addcuq() to capture the carry. For example:

```
...
// We add 0x3fff to GRX-bits which may carry into low order sig-bit
// This may result in a carry out from the L-bit into C-bit.
q_carry = vec_addcuq (q_sig, (vui128_t) RXmask);
q_sig = vec_adduqm (q_sig, (vui128_t) RXmask);
// Generate a bool mask from the carry to use in the vsel
qcmask = vec_setb_cyq (q_carry);
// Two cases; 1) We did carry so shift (double) left 112 bits
q_sigc = vec_sldqi (q_carry, q_sig, 112);
// 2) no carry so shift right 15 bits
q_sig = vec_srqi ((vui128_t) q_sig, 15);
// Select which based on carry
q_sig = (vui128_t) vec_sel ((vui32_t) q_sig, (vui32_t) q_sigc, (vui32_t) qcmask);
// Increment the exponent based on the carry
q_exp = vec_addudm (q_exp, (vui64_t) q_carry);
```

In this case having the carry as a separate vector simplifies adjusting the exponent.

7.4.2.2.1.2 IR for addition

Quad-precision addition and subtraction is a case where right justifying the **IR** is helpful. For addition/subtraction the **IR** only needs 117-bits which can be accommodated in a single 128-vector. Significands (which includes the leading/implicit bit) can be converted to **IR** form by shifting left 3-bits. This still leaves room on the left for the carry-bit. For example:

```
// Precondition the significands before add so the GRX bits
// are in the least significant 3 bit.
a_sig = vec_slqi (a_sig, 3);
b_sig = vec_slqi (b_sig, 3);
```

In this case we need to insure that any right shifts of the **IR** collect any bits shifted-away into the X-bit. For example:

```
// At this point we can assume that:
```

```
// 1) The magnitude (vfa) >= magnitude (vfb)
// 1a) Exponents (a_exp, b_exp) in the range E_min -> E_max
// 1b) The GRX-bits of a_sig/b_sig are still 0b000.
// 2) a_exp >= b_exp
// 2a) If a_exp == b_exp then a_sig >= b_sig
// 2b) If a_exp > b_exp then
//     shift (b_sig) right by (a_exp - b_exp) bits
//     any bits shifted out of b_sig are ORed into the X-bit
if (vec_cmpud_all_lt (b_exp, a_exp))
  {
    vui64_t d_exp, l_exp;
    vui128_t t_sig;
    const vui64_t exp_128 = (vui64_t) CONST_VINT64_DW( 128, 128 );
    d_exp = vec_subudm (a_exp, b_exp);
    if (vec_cmpud_all_lt (d_exp, exp_128))
      {
        l_exp = vec_subudm (exp_128, d_exp);
        t_sig = vec_srq (b_sig, (vui128_t) d_exp);
        x_bits = vec_slq (b_sig, (vui128_t) l_exp);
      }
    else
      {
        x_bits = b_sig;
        t_sig = (vui128_t) q_zero;
      }
    // Collect any bits shifted-away, into a single carry bit
    p_odd = vec_addcuq (x_bits, (vui128_t) q_ones);
    // OR this into the X-bit
    b_sig = (vui128_t) vec_or ((vui32_t) t_sig, (vui32_t) p_odd);
  }
```

In this case, condition-1 means that, only b_sig needs right shifting before significand addition. The a_sig can be used directly as it had previously been left shifted 3-bits where the GRX-bits were set to 0b000.

A simpler case occurs when addition generates a carry. Here we need to shift right 1-bit while preserving any nonzero X-bit. For example:

```
// Check for carry and adjust
if (vec_cmpuq_all_gt (s_sig, (vui128_t) sigov))
  {
    p_odd = (vui128_t) vec_and ((vui32_t) s_sig, xbitmask);
    s_sig = vec_srqi (s_sig, 1);
    s_sig = (vui128_t) vec_or ((vui32_t) s_sig, (vui32_t) p_odd);
    q_exp = vec_addudm (q_exp, q_one);
  }
```

These two sequences preserve the X-bit going into the rounding stage. See Examples for Round to Nearest Even and Examples for Round to Odd.

7.4.2.2.1.3 IR for multiply

Quad-precision Multiply and Multiply-Add require quadword register pairs to hold the **IR** product. The product of two 113-bit significands requires 226-bits. This includes the product C-/L-bits but we will need at least 3 additional bits for GRX.

We can use operations from vec_int128_ppc.h to produce the double quadword product. By pre-adjusting the inputs before the multiply we can align the split between the high 114-bits (right justified) and low 112-bits (left justified) of the product to align with the high and low quadword registers. For example:

```
// Pre-align the multiply inputs so that the product is split
// with the CL-Fraction-bits in high_sig, and GRX-bits in low_sig.
a_sig = vec_slqi (a_sig, 8);
b_sig = vec_slqi (b_sig, 8);
low_sig = vec_muludq (&high_sig, a_sig, b_sig);
```

The high_sig includes the C-/L-bits and high-order 112-bits of product fraction. The low_sig includes the low order 112-bits of the fraction followed by 16-bits of zero.

Most of the low-order fraction bits contribute to the X-bit. So we can normally delay collecting x-bits until the rounding stage. Only the most extreme denormals will result in a right shift large enough to require intervention to preserve bits that would otherwise be shifted away.

After normalization the high-order bits of low_sig become the GRX-bits. Actually we can treat the low order 126-bits as uncollected x-bits. We use this to simplify the rounding process. For this case we can manipulate low_sig to generate a carry for rounding up/down/odd and then add/subtract/or this carry to high_sig to produce the rounded product significand. See Examples for Round to Nearest Even and Examples for Round to Odd.

7.4.2.2.2 Rounding for Quad-Precision

The PowerISA support 6 rounding modes for Quad-Precision

- Round to Nearest Even
- Round towards Zero
- Round towards +Infinity
- Round towards -Infinity
- Round to Nearest Away
- Round to Odd

Note

> See `GCC Avoids Double Rounding Errors With Round-To-Odd`

The first four modes are encoded in the **FPSCR**$_{\mathbf{RN}}$ rounding mode bits. The last two are encoded in instructions as instruction local overrides. The VSX Scalar Round to Quad-Precision Integer instruction can override the RN and encode any of the six rounding modes.

The rounding mode results are defined in terms of the intermediate result (IR), and how close it is to the *representable result*, based on the GRX-bits. The IR is either; exact, closer to the next lower (NL) representable result, Midway between, or closer to the next Higher (NH) representable result,

G	R	X	interpretation
0	0	0	IR is exact
0	0	1	IR is closer to NL
0	1	0	IR is closer to NL
0	1	1	IR is closer to NL
1	0	0	IR is midway between NL and NH
1	0	1	IR is closer to NH
1	1	0	IR is closer to NH
1	1	1	IR is closer to NH

Next lower is effectively truncating the IR (setting GRX = 0b000), while next higher will increment the significand by one.

- Round to Nearest Even
  - If exact chose IR
  - Otherwise if IR is closer to NL, choose NL
  - Otherwise if IR is closer to NH, choose NH
  - Otherwise if IR in midway, choose whichever makes the result even.
- Round towards Zero
  - If exact chose IR
  - Otherwise, choose NL
- Round towards +Infinity

  - If exact chose IR
  - Otherwise if positive, choose NH
  - Otherwise if negative, choose NL

- Round towards -Infinity
  - If exact chose IR
  - Otherwise if positive, choose NL
  - Otherwise if negative, choose NH
- Round to Nearest Away
  - If exact chose IR
  - Otherwise if G = 0, choose NL
  - Otherwise if G = 1, choose NH
- Round to Odd
  - If exact chose IR
  - Otherwise, choose NL, and if G=1, or R=1, or X=1, set the least significant bit to 1.

Note

> The coding examples that follow are incomplete. More examples will be provided are more operation are completed and tested.

The exact coding for rounding modes depends on how the **IR** is represented in vector registers and this may differ by operation. Conversions, addition, and subtraction tend to use a single vector register where the GRX resides in the low-order bits. While Multiplies generate double quadword results and so use vector register pairs. Here the CL and Fraction bits are right justified in a high_sig vector. While the low-order fraction/GRX bits are left justified in a low_sig vector.

7.4.2.2.2.1 Examples for Round to Nearest Even

Example for Convert to Quad-precision:

```
...
// We add 0x3fff to GRX-bits which may carry into low order sig-bit
// This may result in a carry out from the L-bit into C-bit.
q_carry = vec_addcuq (q_sig, (vui128_t) RXmask);
q_sig = vec_adduqm (q_sig, (vui128_t) RXmask);
// Generate a bool mask from the carry to use in the vsel
qcmask = vec_setb_cyq (q_carry);
// Two cases; 1) We did carry so shift (double) left 112 bits
q_sigc = vec_sldqi (q_carry, q_sig, 112);
// 2) no carry so shift right 15 bits
q_sig = vec_srqi ((vui128_t) q_sig, 15);
// Select which based on carry
q_sig = (vui128_t) vec_sel ((vui32_t) q_sig, (vui32_t) q_sigc, (vui32_t) qcmask);
// Increment the exponent based on the carry
q_exp = vec_addudm (q_exp, (vui64_t) q_carry);
```

This code runs about 16 instructions.

Example for Add/Subtract Quad-precision:

```
// Round to nearest even from low_sig bits
// For "round to Nearest, ties to even".
// GRX = 0b001 - 0b011; truncate
// GRX = 0b100 and bit-127 is odd; round up, otherwise truncate
// GRX = 0b100 - 0b111; round up
const vui32_t rmask = CONST_VINT128_W(0, 0, 0, 3);
vui128_t p_rnd;
```

```
// We can simplify by copying the low order fraction bit (p_odd)
// and ADD it to the bit-X. This forces a tie into rounding up
// if the current fraction is odd, making it even.
// This simplifies to (s_sig + rmask + p_odd)
// Rounding will generate a carry into the low order fraction bit
// if and only if GRX > 0b100 or (GRX == 0b100) && (bit-127 == 1)
p_tmp = vec_srqi (s_sig, 3);
p_odd = (vui128_t) vec_and ((vui32_t) p_tmp, onemask);
s_sig = vec_addecuq (s_sig, (vui128_t) rmask, p_odd);
q_sig = vec_srqi (s_sig, 3);
```

Adding 3 plus the fraction odd bit to thex GRX-bits generates a carry into the low-order fraction bit for rounding. In this case we use the extend form of add quadword to effect a 3 way add. After rounding convert the **IR** into a significand by shifting right 3 bits.

Example for Multiply Quad-precision:

```
const vui32_t rmask = CONST_VINT128_W(0x7fffffff, -1, -1, -1);
vui128_t p_rnd;
// We can simplify by copying the low order bit (p_odd) of high_sig
// and OR it with the bit-X. This forces a tie into rounding up
// if the current fraction is odd, making it even.
// This simplifies to the carry from (low_sig + rmask + p_odd)
// Rounding will generate a carry into p_rnd.
// if and only if GRX > 0b100 or (GRX == 0b100) && (bit-127 == 1)
p_odd = (vui128_t) vec_and ((vui32_t) high_sig, onemask);
p_rnd = vec_addecuq (low_sig, (vui128_t) rmask, p_odd);
q_sig =  vec_adduqm (high_sig, p_rnd);
```

Here we take advantage of *uncollected x-bits* in low_sig. Until we add the rmask and generate the carry, we can O↩R/ADD p_odd to any bit in low_sig except the Guard or Round bits. In this case we use the extend/carry form of add quadword to effect a 3 way add and generate the carry/round bit.

Note

> In the specific examples above the **and** with the *onemask* is not required before use as the VRC operand of vec_addecuq(). The instruction internally uses only bit[127] for the carry-in on the extended quadword add.

7.4.2.2.2.2 Examples for Round toward Zero

Example for Convert Quadword to Quad-precision:

```
// Simplest case, shift right 15 bits
q_sig = vec_srqi ((vui128_t) q_sig, 15);
```

This code runs about 3 instructions.

7.4.2.2.2.3 Examples for Round to Odd

Example for Convert to Quad-precision.

```
const vui32_t RXmask = CONST_VINT128_W ( 0, 0, 0, 0x7fff);
vui32_t q_odd;
// For "Round to Odd".
// If if G=1, or R=1, or X=1, Set least significant bit to 1.
// Isolate GRX bit then add the mask.
q_odd = vec_and ((vui32_t) q_sig, RXmask);
// The add will generate a carry into bit 112, for non-zero GRX
q_odd = vec_add (q_odd, RXmask);
// Or this into bit 112 of the q_sig.
q_sig = (vui128_t) vec_or ((vui32_t) q_sig, q_odd);
q_sig = vec_srqi ((vui128_t) q_sig, 15);
```

This code runs about 6 instructions to load the mask and round=odd.

Example for Add/Subtract Quad-precision:

```
// Round to odd from low order GRX-bits
p_tmp = vec_slqi (s_sig, 125);
p_odd = vec_addcuq (p_tmp, (vui128_t) q_ones);
q_sig = vec_srqi (s_sig, 3);
q_sig = (vui128_t) vec_or ((vui32_t) q_sig, (vui32_t) p_odd);
```

Examples for Multiply Quad-precision. For this case we can manipulate low_sig to generate a carry for rounding up/down/odd and then add/subtract/or this carry to high_sig to produce the rounded product significand. For example round-to-odd:

```
// Round to odd from low_sig bits
p_odd = vec_addcuq (low_sig, (vui128_t) q_ones);
q_sig = (vui128_t)  vec_or ((vui32_t) high_sig, (vui32_t) p_odd);
```

7.4.2.3 Quad-Precision compares for POWER8

IEEE floating-point compare is a bit more complicated than binary integer compare operations. The two main complications are; Not-a-Number (NaN) which IEEE insists are *unordered*, and signed 0.0 where IEEE insists that -0.0 is equal to +0.0. If you ignore the NaN and signed 0.0 cases you can treat floating-point values as signed magnitude binary integers, and use integer compares and boolean logic. Which looks like this:

- $a =^f b == (a =^s b)$
- $a <^f b == (a >=^s 0 \& a <^s b) | (a <^s 0 \& a >^u b)$
- $a <=^f b == (a >=^s 0 \& a <=^s b) | (a <^s 0 \& a >=^u b)$

Where; $=^f$, $<^f$, and $<=^f$ are the desired floating-point compares, $=^s$, $<^s$, $<=^s$, $>^s$ and $>=^s$, are signed integer compares, and $=^u$, $>^u$, and $>=^u$ are unsigned integer compares.

See also

> "Hacker's Delight, 2nd Edition," Henry S. Warren, Jr, Addison Wesley, 2013. Chapter 17, Floating-point, Section 17-3 Comparing Floating-Point Numbers Using Integer Operations.

One key implication of this is that we will need signed and unsigned 128-bit compare operations. Instructions for 128-bit integer compares was added for PowerISA 3.1 (POWER10) but we also need to support POWER8/9. The good news is that PowerISA 2.07B (POWER8) includes Vector Add/Subtract Modulo/Carry/Extend Quadword instructions. Specifically Vector Subtract & write Carry Unsigned Quadword can implement all the unsigned ordered (<. <=, >, >=) compares by manipulating the comparand order and evaluating the carry for 0 or 1.

POWER8 also includes vector doubleword integer compare instructions. And the Vector Compare Equal To Unsigned Doubleword instruction can be used with a few additional logical operations to implement 128-bit equal and not equal operations. These operations are already provided by vec_int128_ppc.h.

Some additional observations:

- The unsigned compare equal can be used for unsigned or signed integers.
- $(a >=^s 0) == \sim(a <^s 0)$.
  - So we only need one compare boolean and the binary NOT.
  - $((a >=^s 0) \& a <^s b) | (\sim(a >=^s 0) \& a >^u b)$.
  - Now this starts to look like a vector select operation.
  - $(src1 \& \sim mask) | (src2 \& mask)$
- $(a >=^s 0)$ is a special case that only depends on the sign-bit.
  - A unsigned compare can be used with a slight change,
  - Propagating the sign-bit across the (quad)word generates the same boolean. This is the vec_setb_sq() operation. The __float128 variant is vec_setb_qp()

Note

The examples that follow, use vector __int128 parameters instead of __binary128 to avoid the hassles of cast incompatibility between scalar __binary128's and vector types. The actual implementations use the xfer functions.

```
vb128_t
test_cmpltf128_v1c (vi128_t vfa128, vi128_t vfb128)
{
  vb128_t altb, agtb;
  vb128_t signbool;
  vb128_t result;
  // Replace (vfa >= 0) with (vfa < 0) == vec_setb_qp (vfa)
  const vui8_t shift = vec_splat_u8 (7);
  vui8_t splat = vec_splat ((vui8_t) vfa128, VEC_BYTE_H);
  signbool = (vb128_t) vec_sra (splat, shift);
  altb = vec_cmpltsq (vfa128, vfb128);
  agtb = vec_cmpgtuq ((vui128_t) vfa128, (vui128_t) vfb128);
  result = (vb128_t) vec_sel ((vui32_t)altb, (vui32_t)agtb, (vui32_t)signbool);
  return result;
}
```

Now we can tackle the pesky signed 0.0 case. The simplest method is to add another term that test for either a or b is -0.0. This simplifies to just logical a OR b and unsigned compare to -0.0. Which looks like this:

- $a =^f b == (a =^s b) \mid ((a \mid b) == 0x80000000...0)$
- $a <^f b == (a >=^s 0 \,\&\, a <^s b) \mid ((a <^s 0 \,\&\, a >^u b) \,\&\, ((a \mid b) != 0x80000000...0))$
- $a <=^f b == (a >=^s 0 \,\&\, a <=^s b) \mid ((a <^s 0 \,\&\, a >=^u b) \mid ((a \mid b) == 0x80000000...0))$

Again we can replace signed compares (a >= 0) and (a < 0) with a single vec_setb_qp() and simplify the boolean logic by using vec_sel(). For the ((a | b) != 0x80000000...0) term we can save an instruction by replacing vec_cmpneuq() with vec_cmpequq() and replacing the AND operation with AND compliment.

```
vb128_t
test_cmpltf128_v2c (vi128_t vfa128, vi128_t vfb128)
{
  const vui32_t signmask = CONST_VINT128_W (0x80000000, 0, 0, 0);
  vb128_t altb, agtb, nesm;
  vui32_t or_ab;
  vb128_t signbool;
  vb128_t result;
  // Replace (vfa >= 0) with (vfa < 0) == vec_setb_qp (vfa)
  const vui8_t shift = vec_splat_u8 (7);
  vui8_t splat = vec_splat ((vui8_t) vfa128, VEC_BYTE_H);
  signbool = (vb128_t) vec_sra (splat, shift);
  altb = vec_cmpltsq (vfa128, vfb128);
  agtb = vec_cmpgtuq ((vui128_t) vfa128, (vui128_t) vfb128);
  or_ab = vec_or ((vui32_t) vfa128, (vui32_t) vfb128 );
  // For ne compare eq and and compliment
  nesm = vec_cmpequq ((vui128_t) or_ab, (vui128_t) signmask);
  agtb = (vb128_t) vec_andc ((vui32_t) agtb, (vui32_t) nesm);
  // select altb for 0's and agtb for 1's
  return (vb128_t) vec_sel ((vui32_t)altb, (vui32_t)agtb, (vui32_t)signbool);
}
```

This sequence runs 27 instructions when you include the constant loads.

An alternative compare method converts both floating-point values in a way that a single (unsigned) integer compare can be used.

```
// for each comparand
 if (n >= 0)
   n = n + 0x80000000;
 else
   n = -n;
// Use unsigned integer comparison
```

An interesting feature of this method is that +0.0 becomes (0x00000000 + 0x80000000 = 0x80000000) and -0.0 becomes (0x80000000 - 0x80000000 = 0x80000000) which effectively converts any -0.0 into +0.0 for comparison. Signed 0.0 solved.

Another optimization converts (n = n + 0x80000000) to (n = n XOR 0x80000000). Gives the same result and for POWER8 a vec_xor() is 2 cycles latency vs 4 cycles for _vec_adduqm().

```
vb128_t
test_cmpltf128_v3d (vui128_t vfa128, vui128_t vfb128)
{
  const vui32_t zero = CONST_VINT128_W (0, 0, 0, 0);
  const vui32_t signmask = CONST_VINT128_W (0x80000000, 0, 0, 0);
  const vui8_t shift = vec_splat_u8 (7);
  vb128_t result;
  vb128_t age0, bge0;
  vui128_t vra, vrap, vran;
  vui128_t vrb, vrbp, vrbn;
  vui8_t splta, spltb;
  // signbool = vec_setb_qp;
  splta = vec_splat ((vui8_t) vfa128, VEC_BYTE_H);
  age0 = (vb128_t) vec_sra (splta, shift);
  vrap = (vui128_t) vec_xor ((vui32_t) vfa128, signmask);
  vran = (vui128_t) vec_subuqm ((vui128_t) zero, (vui128_t) vfa128);
  vra  = (vui128_t) vec_sel ((vui32_t)vrap, (vui32_t)vran, (vui32_t)age0);
  spltb = vec_splat ((vui8_t) vfb128, VEC_BYTE_H);
  bge0 = (vb128_t) vec_sra (spltb, shift);
  vrbp = (vui128_t) vec_xor ((vui32_t) vfb128, signmask);
  vrbn = (vui128_t) vec_subuqm ((vui128_t) zero, (vui128_t) vfb128);
  vrb  = (vui128_t) vec_sel ((vui32_t)vrbp, (vui32_t)vrbn, (vui32_t)bge0);
  return vec_cmpltuq (vra, vrb);
}
```

This sequence runs (approximately) 20 instructions when you include loading the required constants. It also manages to use only splat-immediate forms to load constants and so does not need to establish the TOC pointer nor any address calculations to access constants via load instructions.

The next IEEE issue is detecting NaNs and returning *unordered* status. Adding the following code to a compare operation insures that if either comparand is NaN; false (unordered) is returned for compares (eq, lt, gt).

```
if (vec_all_isnanf128 (vfa) || vec_all_isnanf128 (vfb))
  return (vb128_t) vec_splat_u32 (0);
```

The pair of vec_all_isnanf128() operations add significant overhead both in code size (39 instructions) and cycles. This form should only be used if is required for correct results and has not been tested by prior logic in this code path.

See also

The combined operation vec_all_isunorderedf128().

Note

At this point we are not trying to comply with PowerISA by setting any FPSCR bits associated with Quad-Precision compare. If such is required, VXSNAN and/or VXVC can be set using the Move To FPSCR Bit 1 (mtfsb1) instruction.

7.4.2.4 Quad-Precision converts for POWER8

IEEE floating-point conversions are also a bit complicated. Dealing with Not-a-Number (NaN), Infinities and subnormal is part of it. But the conversion may also require normalization and rounding depending on element size and types involved. Some examples:

- Double precision floats and long long integers can be represented exactly in Quad precision float. But:
- Down conversions (to doubleword) from Quad-Precision may require rounding/truncation.
  - Conversions to integer that overflow are given special values.

- Conversions between QP and quadword integer may also require rounding/truncation.
  - 128-bit integer values may not fit into the QPs 113-bit significand.

For PowerISA 3.0 (POWER9) includes full hardware instruction support for Quad-Precision, Including:

- Conversions between Quad-Precison and Double floating-point (xscvdpqp, xscvqpdp[o]).
- Conversions between Quad-Precison and doubleword and word integer (xscvqpsdz, xscvqpswz, xscvqpudz, xscvqpuwz, xscvsdqp, xscvudqp).

PowerISA 3.1 (POWER10) includes:

- Conversions between Quad-Precison and quadword integer (xscvqpsqz, xscvqpuqz, xscvsqqp, xscvuqqp).

For POWER8 (and earlier) we need to do a little more work The general plan for conversion starts by disassembling the input value into its parts and analyze. For signed integer values disassemble usually means sign and unsigned magnitude. Analysis might be a range check or counting leading zeros. For floating point values this is usually sign, exponent, and significand. Analysis usually means determining the data class (NaN, infinity, normal, subnormal, zero) as each requires special handling in the conversion.

Conversion involves adjusting the *parts* as needed to match the type of the result. This is normally only adds and shifts. Finally we need to reassemble the parts based on the result type. For integers this normally just converting the unsigned magnitude to a signed '2's complement value based on the sign of the input. For floating-point this requires merging the sign bit with the (adjusted) significand and merging that with the (adjusted) exponent.

The good news is that all of the required operations are already available in **altivec.h** or PVECLIB.

7.4.2.4.1 Convert Double-Precision to Quad-Precision

This is one of the simpler conversions as the conversion is always exact (no rounding/truncation is required, and no overflow is possible). The process starts with disassembling the double-precision value.

```
const vui32_t signmask = CONST_VINT128_W (0x80000000, 0, 0, 0);
f64[VEC_DW_L] = 0.0; // clear the right most element to zero.
// Extract the exponent, significand, and sign bit.
d_exp = vec_xvxexpdp (f64);
d_sig = vec_xvxsigdp (f64);
q_sign = vec_and ((vui32_t) f64, signmask);
```

We insure that the low-order doubleword of the vector f64 is zeroed. This is necessary for then we normalize the 128-bit significand for the quad-precision result. The operations vec_xvxexpdp() and vec_xvxsigdp() are provided by vec_f64_ppc.h supporting both the POWER9 instruction and equivalent implementation for POWER8. And finally we extract the sign-bit. We can't use the copysign() here due to the difference in type.

Now we analyze the data class of the double-precision input.

```
if (vec_all_isfinitef64 (f64))
  { // Not NaN or Inf
    if (vec_all_isnormalf64 (vec_splat (f64, VEC_DW_H)))
      {
      // ... adjust exponent and expand significand
      }
    else
      { // Must be zero or subnormal
        if (vec_all_iszerof64 (f64))
          {
            // ... copy zero exponent and significand
          }
        else
          { // Must be subnormal
```

```
            // ... normalize signifcand for QP and adjust exponent
          }
      }
  }
else
  { // isinf or isnan.
    // ... set exponent to QP max and expand significand
  }
```

This code is arranged with an eye to the most common cases and specifics of the conversion required by each data class. The operations vec_all_isfinitef64(), vec_all_isnormalf64() and vec_all_iszerof64() are provided by vec_f64_ppc.h supporting both the POWER9 instruction and equivalent implementation for POWER8.

The normal case requires shifting the significand and adjusting the exponent.

```
const vui64_t exp_delta = (vui64_t) CONST_VINT64_DW ( (0x3fff - 0x3ff), 0 );
        ...
        q_sig = vec_srqi ((vui128_t) d_sig, 4);
        q_exp = vec_addudm (d_exp, exp_delta);
```

The double significand has the fraction bits starting a bit-12 and the implied '1' in bit-11. For quad-precision we need to shift this right 4-bits to align the fraction to start in bit-16. We need a quadword shift as the significand will now extend into the high order bits of the second (low order) doubleword. To adjust the exponent we need to convert the double biased exponent (1 to 2046) into unbiased (-1022 to +1023) by subtracting the exponent bias (+1023 or 0x3ff) value. Then we can add the quad-precision exponent bias (+16383 or 0x3fff) to compute the final exponent. We can combine the bias difference into a single constant (0x3fff - 0x3ff) and only need a single add at runtime.

The operations vec_srqi() is provided by vec_int128_ppc.h and vec_addudm() is provided by vec_int64_ppc.h. We use PVECLIB operations here to insure that this code is safe to use with older compilers and pre-POWER8 processors.

The zero case requires setting the quad-precision significand and exponent to zero.

```
q_sig = (vui128_t) d_sig;
q_exp = (vui64_t) d_exp;
```

We know that the double significand and exponent are zero, so just assign them to the quad-precision parts. The sign bit will applied later with the common insert exponent code.

The subnormal case is a bit more complicated. The tricky part is while the double-precision value is subnormal the equivalent quad-precision value is not. So we need to normalize the significand and compute a new exponent.

```
// Need to adjust the quad exponent by the f64 denormal exponent
// (-1023) knowing that the f64 sig will have at least 12 leading '0's
vui64_t q_denorm = (vui64_t) CONST_VINT64_DW ( (16383 - (1023 -12)), 0 );
vui64_t f64_clz;
f64_clz = vec_clzd (d_sig);
d_sig = vec_vsld (d_sig, f64_clz);
q_exp = vec_subudm (q_denorm, f64_clz);
q_sig = vec_srqi ((vui128_t) d_sig, 15);
```

We use a doubleword count leading zeros (ctz) and shift left to normalize the significand so that the first '1'-bit moves to bit-0. Then we compute the quad-precision exponent by subtracting the ctz value from a constant (16383 - (1023 -12)). This represents the quad-precision exponent bias, minus the double-precsion exponent bias, minus the minimum leading zero count for the double-precision subnormal significand.

The NaN/Infinity case requires shifting the significand and setting the exponent to quad-precision max.

```
q_sig = vec_srqi ((vui128_t) d_sig, 4);
q_exp = (vui64_t) CONST_VINT64_DW (0x7fff, 0);
```

We need this shift as NaN has a non-zero significand and it might be nonzero in one of the low order bits. Separating out the infinity case (where the significand is zero) is not worth the extra (isnan) test to avoid the shift.

Now that all the parts are converted to quad-precision, we are ready to reassemble the QP result.

```
// Copy Sign-bit to QP significand before insert.
q_sig = (vui128_t) vec_or ((vui32_t) q_sig, q_sign);
// Insert exponent into significand to complete conversion to QP
result = vec_xsiexpqp (q_sig, q_exp);
```

Putting this all together we get something like:

```
vui64_t d_exp, d_sig, q_exp;
vui128_t q_sig;
vui32_t q_sign;
const vui64_t exp_delta = (vui64_t) CONST_VINT64_DW ( (0x3fff - 0x3ff), 0 );
const vui32_t signmask = CONST_VINT128_W (0x80000000, 0, 0, 0);
f64[VEC_DW_L] = 0.0; // clear the right most element to zero.
// Extract the exponent, significand, and sign bit.
d_exp = vec_xvxexpdp (f64);
d_sig = vec_xvxsigdp (f64);
q_sign = vec_and ((vui32_t) f64, signmask);
if (vec_all_isfinitef64 (f64))
  { // Not NaN or Inf
    if (vec_all_isnormalf64 (vec_splat (f64, VEC_DW_H)))
      {
        q_sig = vec_srqi ((vui128_t) d_sig, 4);
        q_exp = vec_addudm (d_exp, exp_delta);
      }
    else
      { // zero or subnormal
        if (vec_all_iszerof64 (f64))
          {
            q_sig = (vui128_t) d_sig;
            q_exp = (vui64_t) d_exp;
          }
        else
          { // Must be subnormal
            vui64_t q_denorm = (vui64_t) CONST_VINT64_DW ( (0x3fff - 1023 -12), 0 );
            vui64_t f64_clz;
            f64_clz = vec_clzd (d_sig);
            d_sig = vec_vsld (d_sig, f64_clz);
            q_exp = vec_subudm (q_denorm, f64_clz);
            q_sig = vec_srqi ((vui128_t) d_sig, 15);
          }
      }
  }
else
  { // isinf or isnan.
    q_sig = vec_srqi ((vui128_t) d_sig, 4);
    q_exp = (vui64_t) CONST_VINT64_DW (0x7fff, 0);
  }
// Copy Sign-bit to QP significand before insert.
q_sig = (vui128_t) vec_or ((vui32_t) q_sig, q_sign);
// Insert exponent into significand to complete conversion to QP
result = vec_xsiexpqp (q_sig, q_exp);
```

At this stage we have a functionally correct implementation and now we can look for opportunities for optimization. One issue is the generated code is fairly large (∼436 bytes and ∼100 instructions). For POWER8 the data class predicates (vec_all_isfinitef64, etc) each require one or more vector constant loads and bit mask operations before the associated vector compares. Also the extract significand operation requires the equivalent of isnormal (with two vector compares) as preparation for conditionally restoring the implied (hidden) bit.

By testing the extracted (exponent and significand) parts directly we can simplify the compare logic and eliminate some (redundant) vector constant loads. For example:

```
const vui64_t d_naninf = (vui64_t) CONST_VINT64_DW ( 0x7ff, 0 );
const vui64_t d_denorm = (vui64_t) CONST_VINT64_DW ( 0, 0 );
// ...
// The extract sig operation has already tested for finite/subnormal.
// So avoid testing isfinite/issubnormal again by simply testing
// the extracted exponent.
if (__builtin_expect (!vec_cmpud_all_eq (d_exp, d_naninf), 1))
  { // Not Nan or Inf
    if (__builtin_expect (!vec_cmpud_all_eq (d_exp, d_denorm), 1))
      {
      // ... adjust exponent and expand significand
      }
    else
      { // Must be zero or subnormal
        if (vec_cmpud_all_eq (d_sig, d_denorm))
          {
            // ... copy zero exponent and significand
          }
        else
          { // Must be subnormal
            // ... normalize signifcand for QP and adjust exponent
          }
      }
  }
else
  { // isinf or isnan.
    // ... set exponent to QP max and expand significand
  }
```

The implementation based on this logic is smaller (∼300 bytes and ∼75 instructions). Performance results TBD.

7.4.2.4.2 Convert Doubleword integer to Quad-Precision

Converting binary integers to floating point is simpler as there are fewer data classes to deal with. Basically zero and non-zero numbers (no signed 0s, infinities or NaNs). Also the conversion from 64-bit integers to 128-bit floating-point is always exact (there is no rounding).

Unsigned doubleword is the simplest case. We only need to test for binary zero. If zero just return a QP +0.0 constant. Otherwise we can treat the binary magnitude as a denormalized number and normalize it. The binary zero test and processing looks like this:

```
int64[VEC_DW_L] = 0UL; // clear the right most element to zero.
// Quick test for 0UL as this case requires a special exponent.
d_sig = int64;
if (vec_cmpud_all_eq (int64, d_zero))
  { // Zero sign, exponent and significand.
    result = vec_xfer_vui64t_2_bin128 (d_zero);
  }
else ...
```

For the non-zero case we assume the binary point follows the unit bit (bit-63) of the 64-bit magnitude. Then we use count leading zeros to find the first significant bit. This count is used to normalize/shift (left justify) the magnitude and adjust the QP exponent to reflect the binary point following the unit (original doubleword bit 63) bit. So far we are using only doubleword data and instructions.

```
{ // We need to produce a normalized QP, so we treat the integer
  // like a denormal, then normalize it.
  // Start with the quad exponent bias + 63 then subtract the count of
  // leading '0's. The 64-bit sig can have 0-63 leading '0's.
  vui64_t q_expm = (vui64_t) CONST_VINT64_DW ((0x3fff + 63), 0 );
  vui64_t i64_clz = vec_clzd (int64);
  d_sig = vec_vsld (int64, i64_clz);
  q_exp = vec_subudm (q_expm, i64_clz);
  q_sig = vec_srqi ((vui128_t) d_sig, 15);
  result = vec_xsiexpqp (q_sig, q_exp);
}
```

The high order bit (after normalization) will become the *implicit* (hidden) bit in QP format. So we shift the quadword right 15-bits to become the QP significand. This shift includes the low order 64-bits we zeroed out early on and zeros out the sign-bit as a bonus. Finally we use vec_xsiexpqp() to merge the adjusted exponent and significand.

The signed doubleword conversion is bit more complicated. We deal with zero case in the same way. Otherwise we need to separate the signed doubleword into a sign-bit and unsigned 64-bit magnitude. Which looks something like this:

```
const vui64_t d_zero = (vui64_t) CONST_VINT64_DW ( 0, 0 );
const vui32_t signmask = CONST_VINT128_W (0x80000000, 0, 0, 0);
...
    // Convert 2s complement to signed magnitude form.
    q_sign = vec_and ((vui32_t) int64, signmask);
    d_neg   = vec_subudm (d_zero, (vui64_t)int64);
    d_sign = (vui64_t) vec_cmpequd ((vui64_t) q_sign, (vui64_t) signmask);
    // Select the original int64 if positive otherwise the negated value.
    d_sig = (vui64_t) vec_sel ((vui32_t) int64, (vui32_t) d_neg, (vui32_t) d_sign);
```

The normalization process is basically the same as unsigned but we merge the sign-bit into the significant before inserting the exponent.

```
// Count leading zeros and normalize.
i64_clz = vec_clzd (d_sig);
d_sig = vec_vsld (d_sig, i64_clz);
q_exp = vec_subudm (q_expm, i64_clz);
q_sig = vec_srqi ((vui128_t) d_sig, 15);
// Copy Sign-bit to QP significand before insert.
q_sig = (vui128_t) vec_or ((vui32_t) q_sig, q_sign);
// Insert exponent into significand to complete conversion to QP
result = vec_xsiexpqp (q_sig, q_exp);
```

7.4.2.4.3 Convert Quad-Precision to Quadword integer

Convertions between quad-precision and quadword integers is complicated by the fact that the QP significand is only 113-bits while the quadword integer magnitude can be 127/128 bits. It may not be possible to represent the quadword magnitude exactly. Conversions from quad-precision float to integer may have nonzero fractions which require rounding/truncation.

For POWER9 we have the **VSX Scalar Convert with round to zero Quad-Precision to Signed/Unsigned Doubleword *(xscvqpsdz/xscvqpudz)*** instructions. For POWER10 we have the **VSX Scalar Convert with round to zero Quad-Precision to Signed/Unsigned Quadword *(xscvqpsqz/xscvqpuqz)*** instructions. Conversion using other rounding modes require using **VSX Scalar Round to Quad-Precision Integer *(xsrqpi)*** instruction.

Note

> The *xsrqpi* instruction allows for overriding the rounding mode as an immediate operand. So a two instruction sequence can implement any of the four **FPSCR$_{RN}$** rounding modes plus the fifth (Round to Nearest Away) mode specific to floating point integer instructions.

For this example we will look at Convert with Round to Zero Quad-Precision to Unsigned Quadword. The POWER10 operation can be implemented as a single xscvqpuqz instruction. For example:

```
static inline vui128_t
vec_xscvqpuqz (__binary128 f128)
{
  vui128_t result;
#if defined (_ARCH_PWR10)  && (__GNUC__ >= 10)
  __asm__(
      "xscvqpuqz %0,%1"
      : "=v" (result)
      : "v" (f128)
      : );
#else
...
#endif
  return result;
}
```

We use in-line assembler here as there are no current or planed compiler intrinsics for this and the C language only supports conversions between __float128 and __int128 scalars. The scalar conversions returns the __int128 result in GPR pair, while we need the result in vector register.

Note

> We could try to implement the POWER9 convert to quadword operation using two xscvqpudz instructions (at 12-cycles each). But this also requires two QP-multiplies (at 24-cycles each), plus xscvudqp/xssubqp (at 12-cycles each). So far it looks like using the POWER8 implementation for POWER9 will actually perform better.

The POWER8 implementation looks like this:

```
vui64_t q_exp, q_delta, x_exp;
vui128_t q_sig;
vb128_t b_sign;
const vui128_t q_zero = { 0 };
const vui128_t q_ones = (vui128_t) vec_splat_s32 (-1);
const vui64_t exp_low = (vui64_t) CONST_VINT64_DW ( 0x3fff, 0x3fff );
const vui64_t exp_high = (vui64_t) CONST_VINT64_DW ( (0x3fff+128), (0x3fff+128) );
const vui64_t exp_127 = (vui64_t) CONST_VINT64_DW ( (0x3fff+127), (0x3fff+127) );
const vui64_t q_naninf = (vui64_t) CONST_VINT64_DW ( 0x7fff, 0x7fff );
result = q_zero;
q_exp = vec_xsxexpqp (f128);
q_sig = vec_xsxsigqp (f128);
x_exp = vec_splatd (q_exp, VEC_DW_H);
b_sign = vec_setb_qp (f128);
if (__builtin_expect (!vec_cmpud_all_eq (x_exp, q_naninf), 1))
  {
    if (vec_cmpud_all_ge (x_exp, exp_low)
     && vec_cmpud_all_eq ((vui64_t)b_sign, (vui64_t)q_zero))
      { // Greater than or equal to 1.0
    if (vec_cmpud_all_lt (x_exp, exp_high))
     { // Less than 2**128-1
      q_sig = vec_slqi (q_sig, 15);
      q_delta = vec_subudm (exp_127, x_exp);
      result = vec_srq (q_sig, (vui128_t) q_delta);
     }
    else
     { // set result to 2**128-1
      result = (vui128_t) q_ones;
     }
   }
  else
   { // less than 1.0 or negative
    result = (vui128_t) q_zero;
   }
 }
else
 { // isinf or isnan.
  vb128_t is_inf;
  // Positive Inf returns all ones
  // else NaN or -Infinity returns zero
  is_inf = vec_cmpequq (q_sig, (vui128_t) q_zero);
  // result = ~NaN | (pos & Inf) -> Inf & (pos & Inf) -> pos & Inf
  result = (vui128_t) vec_andc ((vui32_t) is_inf, (vui32_t) b_sign);
 }
```

As is the usual for floating-point conversions, we extract the sign, significand, and exponent then test for class and range. We compare the extracted exponent directly using vector doubleword compares. These are faster (on POWER8) than quadword compares but require doubleword splatting the QP exponent and compare constants for correct results. This only requires one additional instruction (xxpermdi) as the vector constants will be loaded as quadwords either way.

The outer test is for NaN/Infinity. These should be rare so we use __builtin_expect(). The implementation returns special values to match the instruction definition.

Once we know the value is finite, we check for greater than or equal to +1.0. Negative or fractional values return quadword zero. Then we check for less than 2^{128}. If not we return all ones (2^{128} -1).

If the input is in the valid range for unsigned quadword we left-justify the significand then shift the quadword right by (127 - $_{unbiased}$exp). The right shift truncates (round toward zero) any fractional bits. See vec_xscvqpuqz().

The signed operation follows similar logic with appropriate adjustments for negative values and reduced magnitude range. The doubleword versions of the convert operation follows the same outline with different range constants. See vec_xscvqpsqz(), vec_xscvqpudz() and vec_xscvqpsdz().

7.4.2.4.4 Convert Quadword integer to Quad-Precision

Conversions from doubleword integer to quad-precision float can be represented exactly and do not require any rounding. But conversions from quadword integer to quad-precision float may overflow the 113-bit significand which does require rounding.

For POWER9 we have the **VSX Scalar Convert Signed/Unsigned Doubleword to Quad-Precision format *(xscvsdqp/xscvudqp)*** instructions. For POWER10 we have the **VSX Scalar Convert with Round Signed/Unsigned Quadword to Quad-Precision format *(xscvsqqp/xscvuqqp)*** instructions. One of four rounding modes is selected from the 2-bit **FPSCR.$_{RN}$** field. The default rounding mode is *Round to Nearest Even* which we will use in this example. Convert using other rounding modes by changing the **FPSCR.$_{RN}$** field.

For example:

```
__binary128
static inline vec_xscvuqqp (vui128_t int128)
{
 __binary128 result;
#if defined (_ARCH_PWR10)  && (__GNUC__ >= 10)
 __asm__(
   "xscvuqqp %0,%1"
   : "=v" (result)
   : "v" (int128)
   : );
#elif defined (_ARCH_PWR9) && defined (__FLOAT128__) && (__GNUC__ > 7)
  vui64_t int64 = (vui64_t) int128;
  __binary128 hi64, lo64;
  __binary128 two64 = 0x1.0p64;
  hi64 = int64[VEC_DW_H];
  lo64 = int64[VEC_DW_L];
  result = (hi64 * two64) + lo64;
#elif  defined (_ARCH_PWR8)
...
#endif
  return result;
}
```

The POWER10 implementation uses the **xscvuqqp** instruction. While POWER9 implementation uses **xscvudqp** instructions to convert the high/low 64-bit halves of the quadword integer. To complete the conversion we need to multiply the converted high 64-bits by 2**64 than add the lower converted 64-bits. The compiler should generate something like this:

```
<test_vec_xscvuqqp_PWR9>:
     addis   r9,r2,0 ## R_PPC64_TOC16_HA   .rodata."0x1.0p64"
     addi    r9,r9,0 ## R_PPC64_TOC16_LO   .rodata."0x1.0p64"
     xxspltd v0,v2,1
     xscvudqp v2,v2
     xscvudqp v0,v0
     lxv     v1,0(r9)
     xsmaddqp v2,v0,v1
     blr
```

The POWER8 implementation looks like this:

```
#elif  defined (_ARCH_PWR8)
  vui64_t q_exp;
  vui128_t q_sig;
  const vui128_t q_zero = (vui128_t) { 0 };
  const vui32_t lowmask = CONST_VINT128_W ( 0, 0, 0, 1);
  q_sig = int128;
  // Quick test for 0UL as this case requires a special exponent.
  if (vec_cmpuq_all_eq (q_sig, q_zero))
    {
      result = vec_xfer_vui128t_2_bin128 (q_zero);
    }
  else
    { // We need to produce a normal QP, so we treat the QW integer
      // like a denormal, then normalize it.
      // Start with the quad exponent bias + 127 then subtract the count of
      // leading '0's. The 128-bit sig can have 0-127 leading '0's.
      vui64_t q_expm = (vui64_t) CONST_VINT64_DW (0, (0x3fff + 127));
      vui64_t i64_clz = (vui64_t) vec_clzq (q_sig);
      q_sig = vec_slq (q_sig, (vui128_t) i64_clz);
      q_exp = vec_subudm (q_expm, i64_clz);
      // This is the part that might require rounding.
      // For example Round to Zero
      // Shift right 15-bits to normalize and truncate
      q_sig = vec_srqi ((vui128_t) q_sig, 15);
      //...
      q_exp = vec_swapd (q_exp);
      result = vec_xsiexpqp (q_sig, q_exp);
    }
```

In this example the significand (including the L-bit) is right justified in the high-order 113-bits of q_sig. The guard, round, and sticky (GRX) bits are in the low-order 15 bits. The sticky-bits are the last 13 bits and are logically ORed (or added to 0x1fff) to produce the X-bit.

The signed quadword conversion is bit more complicated for both POWER9/8. For example:

```
__binary128
static inline vec_xscvsqqp (vi128_t int128)
{
  __binary128 result;
#if defined (_ARCH_PWR10)  && (__GNUC__ >= 10)
  __asm__(
      "xscvsqqp %0,%1"
      : "=v" (result)
      : "v" (int128)
      : );
#elif defined (_ARCH_PWR9) && defined (__FLOAT128__) && (__GNUC__ > 7)
  __binary128 hi64, lo64, i_sign;
  __binary128 two64 = 0x1.0p64;
  vui128_t q_sig;
  vui32_t q_sign;
  vui128_t q_neg;
  vb128_t b_sign;
  const vui32_t signmask = CONST_VINT128_W (0x80000000, 0, 0, 0);
  // Collect the sign bit of the input value.
  q_sign = vec_and ((vui32_t) int128, signmask);
  // Convert 2s complement to unsigned magnitude form.
  q_neg  = (vui128_t) vec_negsq (int128);
  b_sign = vec_setb_sq (int128);
  q_sig = vec_seluq ((vui128_t) int128, q_neg, b_sign);
  // generate a signed 0.0 to use with vec_copysignf128
  i_sign = vec_xfer_vui32t_2_bin128 (q_sign);
  // Convert the unsigned int128 magnitude to __binary128
  vui64_t int64 = (vui64_t) q_sig;
  hi64 = int64[VEC_DW_H];
  lo64 = int64[VEC_DW_L];
  result = (hi64 * two64) + lo64;
  // copy the __int128's sign into the __binary128 result
  result = vec_copysignf128 (i_sign, result);
#elif  defined (_ARCH_PWR8)
...
#endif
  return result;
}
```

For POWER9 we can not just use the signed doubleword conversions for this case. First we split the signed quadword into a 128-bit boolean (representing the sign) and an unsigned quadword magnitude. Then perform the unsigned conversion to QP format as for vec_xscvuqqp(), And finally use vec_copysignf128() to insert the original sign into the QP result.

Similarly for POWER8:

```
#elif  defined (_ARCH_PWR8)
 vui64_t q_exp;
 vui128_t q_sig;
 vui128_t q_neg;
 vui32_t q_sign;
 vb128_t b_sign;
 const vui128_t q_zero = (vui128_t) { 0 };
 const vui32_t lowmask = CONST_VINT128_W ( 0, 0, 0, 1);
 const vui32_t signmask = CONST_VINT128_W (0x80000000, 0, 0, 0);
 // Quick test for 0UL as this case requires a special exponent.
 if (vec_cmpuq_all_eq ((vui128_t) int128, q_zero))
   {
     result = vec_xfer_vui128t_2_bin128 (q_zero);
   }
 else
   { // We need to produce a normal QP, so we treat the integer like a
     // denormal, then normalize it.
     // Collect the sign bit of the input value.
     q_sign = vec_and ((vui32_t) int128, signmask);
     // Convert 2s complement to signed magnitude form.
     q_neg  = (vui128_t) vec_negsq (int128);
     b_sign = vec_setb_sq (int128);
     q_sig = vec_seluq ((vui128_t) int128, q_neg, b_sign);
     // Start with the quad exponent bias + 127 then subtract the count of
     // leading '0's. The 128-bit sig can have 0-127 leading '0's.
     vui64_t q_expm = (vui64_t) CONST_VINT64_DW (0, (0x3fff + 127));
     vui64_t i64_clz = (vui64_t) vec_clzq (q_sig);
     q_sig = vec_slq (q_sig, (vui128_t) i64_clz);
     q_exp = vec_subudm (q_expm, i64_clz);
     // This is the part that might require rounding.
     // For example Round to Zero
     // Shift right 15-bits to normalize and truncate
     q_sig = vec_srqi ((vui128_t) q_sig, 15);
     q_exp = vec_swapd (q_exp);
     // Copy Sign-bit to QP significand before insert.
     q_sig = (vui128_t) vec_or ((vui32_t) q_sig, q_sign);
     result = vec_xsiexpqp (q_sig, q_exp);
   }
```

7.4.2.4.5 Convert Quad-Precision to Double-Precision

TBD

7.4.2.4.6 Round to Quad-Precision Integer

TBD

7.4.2.5 Quad-Precision Arithmetic

The POWER9 (PowerISA 3.0B) processor provides a full set of Quad-Precision arithmetic operations; add, divide, multiply, multiply-add/sub, and subtract. The compilers (that support IEEE128) provide normal C-language arithmetic operators for the __float128 (ISO _Float128) data type. The compiler will generate in-line quad-precision instructions for the (-mcpu=power9) target and calls to the (soft-float) runtime for earlier processor targets.

POWER9 also provides a round-to-odd override for these operations. This helps software avoid *double rounding* errors when rounding to smaller precision. These are supported with compile built-ins (or in-line assembler).

The compilers (that support IEEE128) provide a soft-float implementation for POWER8. However the runtime does not support round-to-odd as a rounding mode. Also the compiler built-ins for round-to-odd arithmetic are disabled for POWER8 and earlier.

As a minimum the PVECLIB implementation should implement POWER8 equivalents to the compiler built-ins supported for ISA 3.0. This would include the explicit round-to-odd operations.

Note

> PVECLIB supports 128-bit and 64-bit vector integer operations for POWER7 but is limited by the original VMX integer word (32-bit) element instructions. This will not provide any performance advantage over a Fixed-Point 64-bit implementation using GPRs.

7.4.2.5.1 Multiply Quad-Precision with Round-to-Odd. The PVECLIB implementation of **Multiply Quad-Precision with Round-to-Odd** will use the POWER9 xsmulqpo instruction if the compile target supports it. Otherwise provide a POWER8 VSX implementation using operations from vec_int128_ppc.h and vec_int64_ppc. For example:

```
__binary128
test_mulqpo_PWR9 (__binary128 vfa, __binary128 vfb)
{
  __binary128 result;
#if defined (_ARCH_PWR9) && (__GNUC__ > 6)
#if defined (__FLOAT128__) && (__GNUC__ > 7)
  // Earlier GCC versions may not support this built-in.
  result = __builtin_mulf128_round_to_odd (vfa, vfb);
#else
  // If the compiler supports _ARCH_PWR9, must support mnemonics.
  __asm__(
      "xsmulqpo %0,%1,%2"
      : "=v" (result)
      : "v" (vfa), "v" (vfb)
      : );
#endif
#else
  // Soft-float implementation
#endif
  return result;
}
```

We prefer the compiler built-in (if available) but can substitute in-line assembler if needed. The built-in is subject to additional compiler optimizations (like instruction scheduling) while in-line assembler is not.

The PVECLIB soft-float implementation can leverage the 128-bit vector registers and operations supported by POWER8. The implementation starts with the usual exponent and significand extraction and ends with merging the computed sign-bit with the significand and inserting the computed exponent. For example:

```
  // Working variables and constants
  vui64_t q_exp, a_exp, b_exp, x_exp;
  vui128_t q_sig, a_sig, b_sig, p_sig_h, p_sig_l, p_odd;
  vui32_t q_sign,  a_sign,  b_sign;
  const vui32_t signmask = CONST_VINT128_W(0x80000000, 0, 0, 0);
  const vui64_t q_zero = { 0, 0 };
  const vui64_t q_ones = { -1, -1 };
  const vui64_t exp_bias = (vui64_t) CONST_VINT64_DW( 0x3fff, 0x3fff );
  const vi64_t exp_min = (vi64_t) CONST_VINT64_DW( 1, 1 );
  const vui64_t exp_dnrm = (vui64_t) CONST_VINT64_DW( 0, 0 );
  const vui64_t exp_naninf = (vui64_t) CONST_VINT64_DW( 0x7fff, 0x7fff );
  const vui64_t exp_max = (vui64_t) CONST_VINT64_DW( 0x7ffe, 0x7ffe );
  const vui32_t sigov = CONST_VINT128_W(0x0001ffff, -1, -1, -1);
  // Extract the exponent, significand, and sign-bit of each operand
  a_exp = vec_xsxexpqp (vfa);
  a_sig = vec_xsxsigqp (vfa);
  a_sign = vec_and_bin128_2_vui32t (vfa, signmask);
  b_exp = vec_xsxexpqp (vfb);
  b_sig = vec_xsxsigqp (vfb);
  b_sign = vec_and_bin128_2_vui32t (vfb, signmask);
  // Vectorize the DW biased exponents
  x_exp = vec_mrgahd ((vui128_t) a_exp, (vui128_t) b_exp);
  // Generate the product sign-bit
  q_sign = vec_xor (a_sign, b_sign);
// test for (vec_all_isfinitef128 (vfa) && vec_all_isfinitef128 (vfb))
  if (vec_cmpud_all_lt (x_exp, exp_naninf))
    { // Both operands are finite (normal, denormal, or zero)
    }
  else
    { // One or both operands are NaN or Infinity
    }
  // Merge sign, significand, and exponent into final result
  q_sig = (vui128_t) vec_or ((vui32_t) q_sig, q_sign);
  result = vec_xsiexpqp (q_sig, q_exp);
```

Finite operands should be to the most common case. So it may help the compiler to use __builtin_expect(). For example:

```
if (__builtin_expect (vec_cmpud_all_lt (x_exp, exp_naninf), 1))
  { // Both operands are finite (normal, denormal, or zero)
  }
else
  { // One or both operands are NaN or Infinity
  }
```

For finite operands we need to multiply the significands, sum the (unbiased) exponents, normalize the product, round, and check for exponent under/overflow. We can use vec_muludq() to multiply two 128-bit unsigned values returning 256-bit product in two (high/low) 128-bit vectors.

The product of two 113-bit significands is up to up to 226-bits (depending on inputs). From this we only need 117-bits (C-bit, L-bit, 112 Fraction bits, G-bit, R-bit, and X (sticky) bit. The X-bit is the logical OR of all 110-bits to the right of the R-bit. But we should not discard (or collapse) any sticky bits until after normalization.

We have some latitude on how we represent this product in vector registers. It is convenient for rounding if the high order 114-bits (C, L, Fraction -bits) are right justified in the high vector. While the low order 112-bit are left justified in in the low vector. One way to accomplish this is:

```
a_sig = vec_slqi (a_sig, 8);
b_sig = vec_slqi (b_sig, 8);
p_sig_l = vec_muludq (&p_sig_h, a_sig, b_sig);
```

There are some special considerations for denormal and zero (+- 0.0) operands. Both have a biased exponent of 0x0000 but we can't use that to compute the product exponent. If the either significand is zero then the product is zero. We can short circuit this by returning the product sign-bit (A^{sign} XOR B^{sign}) followed by 127 0b0s.

Otherwise a denormal is *encoded* as a biased exponent of 0x0000 and a nonzero (112-bit) fraction. But the architecture defines a denormal as 2^{Emin} x (0.fraction). For Quad-Precision, Emin is defined as -16382 which is a biased exponent of 0x0001. So which value (0x0000 or 0x0001) is used to compute the product exponent for the intermediate result?

For denormal values we use Emin to compute the product exponent before normalization. This requires a small fix-up before computing the exponent. We can optimize things by nesting the denormal fix-up under the zero check.

```
      // Splat exponents to both DWs
      a_exp = vec_splatd (a_exp, VEC_DW_H);
      b_exp = vec_splatd (b_exp, VEC_DW_H);
      // x_exp is the vectorized DW a_exp/b_exp
      if (vec_cmpud_any_eq (x_exp, exp_dnrm))
        { // Involves zeros or denormals
          // check for zero significands in multiply
          if (vec_cmpuq_all_eq (a_sig, (vui128_t) q_zero)
              || vec_cmpuq_all_eq (b_sig, (vui128_t) q_zero))
            { // Multiply by zero, return QP signed zero
              result = vec_xfer_vui32t_2_bin128 (q_sign);
              return result;
            }
          else
            { // Denormals present
              vb64_t exp_mask;
              // Convert zero encode to Emin for any denormal
              exp_mask = vec_cmpequd (x_exp, exp_dnrm);
              x_exp = (vui64_t) vec_sel (x_exp, (vui64_t) exp_min, exp_mask);
              // Propagate Emin back to a_exp/b_exp
              a_exp = vec_splatd (x_exp, VEC_DW_H);
              b_exp = vec_splatd (x_exp, VEC_DW_L);
            }
        }
//...
      // sum biased exponents for multiply
      q_exp = vec_addudm (a_exp, b_exp);
      q_exp = vec_subudm (q_exp, exp_bias);
```

The exponent vector are splatted to both doublewords. We need the exponent in the high order doubleword for input to vec_xsiexpqp(). We need the exponent in the low order doubleword for computing shift amounts as input to vec_sldq(), vec_slq(), vec_srq(). Here we use biased exponents in the computation and result. The addition will double the bias so we need to correct it by subtracting the constant exp_bias.

Again the zero multiply case is rare, so we can help the compiler by using __builtin_expect(). Also we can replace the **else** block with boolean select logic set exponents to **Emin** for denormal operands. This also eliminates a redundant compare for **exp_dnrm**.

```
// check for zero significands in multiply
if (__builtin_expect (
     (vec_all_eq ((vui32_t) a_sig, (vui32_t) q_zero)
   || vec_all_eq ((vui32_t) b_sig, (vui32_t) q_zero)), 0))
  { // Multiply by zero, return QP signed zero
    result = vec_xfer_vui32t_2_bin128 (q_sign);
    return result;
  }
// Convert zero encode to Emin for any denormal
{
  vb64_t exp_mask;
  exp_mask = vec_cmpequd (x_exp, exp_dnrm);
  x_exp = vec_selud (x_exp, exp_min, (vb64_t) exp_mask);
  a_exp = vec_splatd (x_exp, VEC_DW_H);
  b_exp = vec_splatd (x_exp, VEC_DW_L);
}
// sum exponents
q_exp = vec_addudm (a_exp, b_exp);
q_exp = vec_subudm (q_exp, exp_bias);
```

The multiply may generate a carry in the intermediate result. This can occur when both operands are normal and sufficiently large. For example 1.5 x 1.5 = 2.25 (0x1.8 x 0x1.8 = 0x2.4). Any product with the C-bit set requires normalization by shifting one bit right and incrementing the exponent. For example:

```
// Check for carry and adjust
if (vec_cmpuq_all_gt (p_sig_h, (vui128_t) sigov))
  {
    p_tmp = vec_sldqi (p_sig_h, p_sig_l, 120);
    p_sig_h = vec_srqi (p_sig_h, 1);
    p_sig_l = vec_slqi (p_tmp, 7);
    q_exp = vec_addudm (q_exp, q_one);
  }
```

The shift sequence above is optimized for the case of a 1 bit shift right double quadword for POWER8.

This case is not rare at all so __builtin_expect() is not helpful for this case. Also quadword compare is an expensive test and requires a load of the sigov constant value. We only need to compare the high order 16-bits of the significand to detect the Carry/Leading bits.

An alternative uses splat halfword to replicate the CL-bits across the vector and use splat immediate halfword for the compare value. For example:

```
vb128_t carry_mask;
```

```
vui16_t sig_l_mask = vec_splat_u16 (1);
vui16_t t_sig = vec_splat ((vui16_t) p_sig_h, VEC_HW_H);
// Detect Carry bit
carry_mask =  (vb128_t) vec_cmpgt (t_sig, sig_l_mask);
```

Then use boolean select logic to propagate the right shifted significand and incremented exponent if carry detected. For example:

```
// Shift double quadword right 1 bit
p_tmp = vec_sldqi (p_sig_h, p_sig_l, 120);
sig_h = vec_srqi (p_sig_h, 1);
sig_l = vec_slqi (p_tmp, 7);
// Increment the exponent
x_exp = vec_addudm (q_exp, exp_one);
// Select original or normalized exp/sig
p_sig_h = vec_seluq (p_sig_h, sig_h, carry_mask);
p_sig_l = vec_seluq (p_sig_l, sig_l, carry_mask);
q_exp = vec_selud (q_exp, x_exp, (vb64_t) carry_mask);
```

Next we may need to deal with denormal results. A intermediate result is considered *tiny* if the exponent is less than Emin. If the result is still *tiny* after normalization and rounding the result is denormal and we can set the product exponent to 0x0000 before inserting it into the sign/significand for the final result.

The normalization of a *tiny* intermediate result is a little complicated. The PowerUSA states:

> If the intermediate result is Tiny (i.e., the unbiased exponent is less than -16382) and UE=0, the significand is shifted right N bits, where N is the difference between -16382 and the unbiased exponent of the intermediate result. The exponent of the intermediate result is set to the value -16382.

This requires two tests. For example:

```
if (vec_cmpsd_all_lt ((vi64_t) q_exp, (vi64_t) exp_min))
  {
    // Intermediate result is tiny, unbiased exponent < -16382
    //
    // ...
  }
if (vec_cmpuq_all_le (p_sig_h, (vui128_t) sigovt))
  { // Not tiny but ...
    // Signicand is below normal range. This can happen when
    // multiplying a denormal by a normal.
    // So try to normalize the significand.
    // ...
  }
```

Again both cases are rare, so we can help the compiler by using __builtin_expect(). Also the second test is a quadword compare requiring another quadword constant. Both expensive compared the alternative.

We only need to compare the high order 16-bits of the significand to detect the a zero Leading bit. Again use splat halfword to replicate the CL-bits across the vector and compare to zero. For example:

```
// Isolate sig CL bits and compare
vui16_t t_sig = vec_splat ((vui16_t) p_sig_h, VEC_HW_H);
if (__builtin_expect ((vec_all_eq (t_sig, (vui16_t) q_zero)), 0))
```

There are some issues that need to be addressed as we (attempt to) normalize tiny results. We need to assure as we shift bits right, any bits shifted out of the low order (112-bit) of the product, are (effectively) accumulated in the sticky-bit (X-bit). This is a consequence of deferred X-bit (accumulated) as we pass the whole 226-bit product (p_sig_h and p_sig_l) into the rounding step. This breaks down in to two cases:

- N is less than or equal to 116.
- N is greater than 116.

The first case may leave significant bits in the fraction and the GR-bits. But we must preserve any bits below GR in the sticky-bit. In this case we accumulate and clear the low-order 109-bits of the product and OR them into the X-bit.

```
const vui32_t xmask = CONST_VINT128_W(0x1fffffff, -1, -1, -1);
// Propagate low order bits into the sticky bit
// GRX is left adjusted in p_sig_l
// Isolate bits below GDX (bits 3-128).
tmp = vec_and ((vui32_t) p_sig_l, xmask);
// generate a carry into bit-2 for any nonzero bits 3-127
tmp = (vui32_t) vec_adduqm ((vui128_t) tmp, (vui128_t) xmask);
// Or this with the X-bit to propagate any sticky bits into X
p_sig_l = (vui128_t) vec_or ((vui32_t) p_sig_l, tmp);
p_sig_l = (vui128_t) vec_andc ((vui32_t) p_sig_l, xmask);
```

Now we can perform a double quadword shift right of up to 116 bits without losing any bits from the intermediate representation. For example:

```
// Need a Double Quadword shift here, so convert right
// shift into shift left double quadword for p_sig_l.
l_exp = vec_subudm (exp_128, x_exp);
p_sig_l = vec_sldq (p_sig_h, p_sig_l, (vui128_t) l_exp);
// Complete right shift for p_sig_h
p_sig_h = vec_srq (p_sig_h, (vui128_t) x_exp);
q_sig = p_sig_h;
```

The second case implies that all bits of the intermediate representation (bits CL through GR) will be shifted away and can only be accumulated in the sticky (X) bit.

```
// Intermediate result is too tiny, the shift will
// zero the fraction and the GR-bit leaving only the
// Sticky bit. The X-bit needs to include all bits
// from p_sig_h and p_sig_l
p_sig_l = vec_srqi (p_sig_l, 8);
p_sig_l = (vui128_t) vec_or ((vui32_t) p_sig_l, (vui32_t) p_sig_h);
// generate a carry into bit-2 for any nonzero bits 3-127
p_sig_l = vec_adduqm (p_sig_l, (vui128_t) xmask);
q_sig = (vui128_t) q_zero;
p_sig_l = (vui128_t) vec_andc ((vui32_t) p_sig_l, xmask);
```

There is another special case where the product exponent is greater than or equal to **Emin** but the significand is not in normal range. Otherwise we can continue to the rounding process.

This case can happen if a normal value is multiplied by a denormal. The normalization process for this case is to:

If the exponent is **Emin** then the result is denormal. Set the biased exponent to zero and continue to rounding.

Otherwise shift the significand left and decrement the exponent, until the exponent equals **Emin** or the significant is in normal range.

For the second case we use count leading zeros to (vec_clzq()) to compute the number bits (*c_exp*) we need to shift left to get a normalized significand. We also compute the number of bits (*d_exp*) we need to decrement the exponent to **Emin**. Then we take the minimum (of *c_exp* and *d_exp*) as the shift count. For example:

```
const vui64_t exp_15 = { 15, 15 };
vui64_t c_exp, d_exp;
vui128_t c_sig;
c_sig = vec_clzq (p_sig_h);
c_exp = vec_splatd ((vui64_t) c_sig, VEC_DW_L);
// Adjust for leading bits before L-bit
c_exp = vec_subudm (c_exp, exp_15);
d_exp = vec_subudm (q_exp, (vui64_t) exp_min);
d_exp = vec_minud (c_exp, d_exp);
```

We use the minimum value (*d_exp*) to shift the significand left. Then check if the result is still denormal. If so set the result biased exponent to zero. Otherwise compute the normalized exponent.

```
// Try to normalize the significand.
p_sig_h = vec_sldq (p_sig_h, p_sig_l, (vui128_t) d_exp);
p_sig_l = vec_slq (p_sig_l, (vui128_t) d_exp);
// Compare computed exp to shift count to normalize.
if (vec_cmpud_all_le (q_exp, c_exp))
  { // exp less than shift count to normalize so
```

```
    // result is still denormal.
    q_exp = q_zero;
  }
else // Adjust exp after normalize shift left.
  q_exp = vec_subudm (q_exp, d_exp);
```

Of course we can replace the exponent compare if-then-else with select logic. For example:

```
vb64_t exp_mask;
exp_mask = vec_cmpgtud (q_exp, c_exp);
q_exp = vec_subudm (q_exp, d_exp);
q_exp = vec_selud (exp_dnrm, q_exp, exp_mask);
```

The round-to-odd case to has some special considerations.

- Round-to-odd will never generate a Carry, so we don't need to check.
- Exponent overflow does not return infinity. Instead it returns **FLT128_MAX** of the appropriate sign.

```
// Round to odd from lower product bits
p_odd = vec_addcuq (p_sig_l, (vui128_t) q_ones);
q_sig = (vui128_t)  vec_or ((vui32_t) q_sig, (vui32_t) p_odd);
// Check for exponent overflow -> __FLT128_MAX__
if  (vec_cmpud_all_gt ( q_exp, exp_max))
{
  // Intermediate result is huge, unbiased exponent > 16383
  // so return __FLT128_MAX__
  q_exp = exp_max;
  q_sig = (vui128_t) sigov;
}
```

Round-to-odd is a simple operation. We already have the significand in a separate vector register from the GRX-bits. So we use vec_addcuq() to generate a carry-bit if there are any non-zero bits in the extended fraction. This carry bit is simply ORed into the low order bit of the significand.

Returning **FLT128_MAX** for exponent overfloat is special case for round-to-odd. Other rounding modes would return infinity. Again this is a rare case and we can help the compiler by using __builtin_expect(). For example:

```
if (__builtin_expect ((vec_cmpud_all_gt ( q_exp, exp_max)), 0))
{
  // Intermediate result is huge, unbiased exponent > 16383
  // so return __FLT128_MAX__ with the appropriate sign.
  const vui32_t f128_max = CONST_VINT128_W(0x7ffeffff, -1, -1, -1);
  vui32_t f128_smax = vec_or ((vui32_t) f128_max, q_sign);
  return vec_xfer_vui32t_2_bin128 (f128_smax);
}
```

7.4.2.5.1.1 NaN and Infinity handling for Multiply When operands are not finite (infinite or Not-a-Number) we have to deal with a matrix of operand pairs and return specific result values.

Note

> The PowerISA provides a table with specified results for each combination of operands. See PowerISA 3.0B Table 82 Actions for xsmulqp[o].

Implementing this matrix of results does not require much computation, but does require conditional logic to separate all the cases. Another complication is that the matrix above specifies a *Default Quiet NaN* to be returned for any multiply of Infinity by zero (ignoring the operand's sign). In this case vec_const_nanf128() is returned while bypassing normal vec_xsiexpqp() exit sequence. All other cases must consider the operand's sign in the result. For example:

```
{ // One or both operands are NaN or Infinity
  if (vec_cmpuq_all_eq (a_sig, (vui128_t) q_zero)
      && vec_cmpuq_all_eq (b_sig, (vui128_t) q_zero))
    { // Both operands either infinity or zero
      if (vec_cmpud_any_eq (x_exp, q_zero))
        { // Infinity x Zero is Default Quiet NaN
          return vec_const_nanf128 ();
        }
      else
        { // Infinity x Infinity == signed Infinity
          q_sign = vec_xor (a_sign, b_sign);
          q_exp = a_exp;
          q_sig = a_sig;
        }
    }
  else
    { // One or both operands are NaN
      const vui32_t q_nan = CONST_VINT128_W(0x00008000, 0, 0, 0);
      if (vec_all_isnanf128 (vfa))
        { // vfa is NaN, return vfa as quiet NAN
          q_sign = a_sign;
          q_sig = (vui128_t) vec_or ((vui32_t) a_sig, q_nan);
          q_exp = a_exp;
        }
      else if (vec_all_isnanf128 (vfb))
        { // vfb is NaN, return vfb as quiet NAN
          q_sign = b_sign;
          q_sig = (vui128_t) vec_or ((vui32_t) b_sig, q_nan);
          q_exp = b_exp;
        }
      else  // Not NaN, so Infinity and a Nonzero finite number
        { // Return signed Infinity
          q_sign = vec_xor (a_sign, b_sign);
          q_sig = (vui128_t) q_zero;
          q_exp = exp_naninf;
        }
    }
}
```

7.4.2.5.2 Add Quad-Precision with Round-to-Odd.

The PVECLIB implementation of **Add Quad-Precision with Round-to-Odd** will use the POWER9 xsaddqpo instruction if the compile target supports it. Otherwise provide a POWER8 VSX implementation using operations from vec_int128_ppc.h and vec_int64_ppc. For example:

```
__binary128
test_addqpo (__binary128 vfa, __binary128 vfb)
{
  __binary128 result;
#if defined (_ARCH_PWR9) && (__GNUC__ > 7)
#if defined (__FLOAT128__) && (__GNUC__ > 8)
  // earlier GCC versions generate extra data moves for this.
  result = __builtin_addf128_round_to_odd (vfa, vfb);
#else
  // No extra data moves here.
  __asm__(
      "xsaddqpo %0,%1,%2"
      : "=v" (result)
      : "v" (vfa), "v" (vfb)
      : );
#endif
#else
  // Soft-float implementation
#endif
  return result;
}
```

We prefer the compiler built-in (if available) but can substitute in-line assembler if needed. The built-in is subject to additional compiler optimizations (like instruction scheduling) while in-line assembler is not.

The PVECLIB soft-float implementation can leverage the 128-bit vector registers and operations supported by POWER8. The implementation starts with the usual sign-bit, exponent and significand extraction and ends with merging the computed sign-bit with the significand and inserting the computed exponent. For example:

```
  vui64_t q_exp, a_exp, b_exp, x_exp;
  vui128_t q_sig, a_sig, b_sig, p_tmp, p_odd;
  vui128_t a_mag, b_mag;
  vui128_t s_sig, x_bits;
  vui32_t q_sign,  a_sign,  b_sign;
  vb128_t a_lt_b;
  const vui32_t signmask = CONST_VINT128_W(0x80000000, 0, 0, 0);
  const vui64_t q_zero = { 0, 0 };
  const vui64_t q_ones = { -1, -1 };
  const vi64_t exp_min = (vi64_t) CONST_VINT64_DW( 1, 1 );
  const vui64_t exp_dnrm = (vui64_t) CONST_VINT64_DW( 0, 0 );
  const vui64_t q_expnaninf = (vui64_t) CONST_VINT64_DW( 0x7fff, 0x7fff );
  const vui64_t q_expmax = (vui64_t) CONST_VINT64_DW( 0x7ffe, 0x7ffe );
  const vui32_t sigov = CONST_VINT128_W(0x000fffff, -1, -1, -1);
  const vui32_t sigovt = CONST_VINT128_W(0x0007ffff, -1, -1, -1);
  const vui32_t xbitmask = CONST_VINT128_W(0, 0, 0, 1);
  a_exp = vec_xsxexpqp (vfa);
  a_sig = vec_xsxsigqp (vfa);
  a_sign = vec_and_bin128_2_vui32t (vfa, signmask);
  b_exp = vec_xsxexpqp (vfb);
  b_sig = vec_xsxsigqp (vfb);
  b_sign = vec_and_bin128_2_vui32t (vfb, signmask);
  // Vectorize the DW biased exponents
  x_exp = vec_mrgahd ((vui128_t) a_exp, (vui128_t) b_exp);
//  if (vec_all_isfinitef128 (vfa) && vec_all_isfinitef128 (vfb))
//  The above can be optimized to the following
  if (__builtin_expect (vec_cmpud_all_lt (x_exp, q_expnaninf), 1))
    { // Both operands are finite (normal, denormal, or zero)
    }
  else
    { // One or both operands are NaN or Infinity
    }
  // Merge sign, significand, and exponent into final result
  q_sig = (vui128_t) vec_or ((vui32_t) q_sig, q_sign);
  result = vec_xsiexpqp (q_sig, q_exp);
```

Floating-point addition requires that operands are represented with the same exponent before the add operation. This means shifting the significand of the smaller magnitude right by the absolute difference between the exponents. To reduce the effect of round-off error this shift must preserve any low order bits (shifted away) as the Guard, Round and Sticky bits for internal *Intermediate Results (**IR**)*. For addition the **IR** only needs 117-bits which can be accommodated in a single 128-vector. Significands (which includes the leading/implicit bit) can be converted to **IR** form by shifting left 3-bits. This still leaves room on the left for the carry-bit. For example:

```
// Precondition the significands before add so the GRX bits
// are in the least significant 3 bit.
a_sig = vec_slqi (a_sig, 3);
b_sig = vec_slqi (b_sig, 3);
```

(See: Representing Intermediate results for Quad-Precision)

Note

> The sign bits are maintained as separate local variables (a_sign, b_sign, q_sign) for the power8 VSX software implementation.

Floating-point addition will add or subtract magnitudes depending the signs of the operands. If the signs are the same, simply add (unsigned quadword) the two (**IR** format) operands. Otherwise if the signs differ, simply subtract (unsigned quadword) the smaller magnitude from the larger. In this case the sign of the result is the sign of the larger magnitude.

This discussion implies that knowing the relative absolute magnitude up front can simplify the implementation. As we saw in Quad-Precision compares for POWER8 we can use quadword integer compares if we know that QP values are finite. By masking off sign-bit from the operands we can use unsigned quadword compare to determine relative magnitude. For example:

```
// Mask off sign bits so can use integers for magnitude compare.
a_mag = (vui128_t) vec_andc_bin128_2_vui32t (vfa, signmask);
b_mag = (vui128_t) vec_andc_bin128_2_vui32t (vfb, signmask);
// If magnitude(b) >  magnitude(a) will need to swap a/b, later
a_lt_b = vec_cmpltuq (a_mag, b_mag);
```

We can also generate a sign difference mask we can use later. For example:

```
vui32_t diff_sign;
// If sign(vfa) != sign(vfb) will need to:
// 1) Subtract instead of add significands
// 2) Generate signed zeros
q_sign = vec_xor (a_sign, b_sign);
diff_sign = (vui32_t) vec_setb_sq ((vi128_t) q_sign);
```

Again a denormal is *encoded* as a biased exponent of 0x0000 and a nonzero (112-bit) fraction. But the architecture defines a denormal as 2^{Emin} x (0.fraction). For Quad-Precision, Emin is defined as -16382 which is a biased exponent of 0x0001. We need to use Emin in the **IR** to to compute shift values. This requires a small fix-up before computing the **IR**. It is simpler at this point to apply Emin to both denormal and QP 0.0 values.

```
// Vectorize the DW biased exponents
x_exp = vec_mrgahd ((vui128_t) a_exp, (vui128_t) b_exp);
//...
    // Correct exponent for zeros or denormals to E_min
      {
        vb64_t exp_mask;
        exp_mask = vec_cmpequd (x_exp, exp_dnrm);
        x_exp = (vui64_t) vec_sel (x_exp, (vui64_t) exp_min, exp_mask);
      }
```

We vectorize this fixup by merging a_exp/b_exp into a vector doubleword x_exp and applying compare/select.

Before diving into the add operation we simplify the code that follows by swapping the internal representation of vfa/vfb if if absolute magnitude of vfa is less than vfb.

```
// Now swap operands a/b if necessary so a has greater magnitude.
  {
    vui128_t a_tmp = a_sig;
    vui128_t b_tmp = b_sig;
    vui64_t x_tmp = vec_swapd (x_exp);
    q_sign = vec_sel (a_sign, b_sign, (vui32_t) a_lt_b);
    x_exp = vec_sel (x_exp, x_tmp, (vui64_t) a_lt_b);
    a_exp = vec_splatd (x_exp, VEC_DW_H);
    b_exp = vec_splatd (x_exp, VEC_DW_L);
    q_exp = a_exp;
    a_sig = vec_seluq (a_tmp, b_tmp, (vb128_t) a_lt_b);
    b_sig = vec_seluq (b_tmp, a_tmp, (vb128_t) a_lt_b);
  }
```

At this point we can assume that:

- Exponents (a_exp, b_exp) are in the range E_min -> E_max
- And a_exp >= b_exp
  - If a_exp == b_exp then a_sig >= b_sig
  - If a_exp > b_exp then
    - shift (b_sig) right by (a_exp - b_exp)
    - any bits shifted out of b_sig are ORed into the X-bit
- The result exponent is usually a_exp +/-1

```
if (vec_cmpud_all_lt (b_exp, a_exp))
  {
    vui64_t d_exp, l_exp;
    vui128_t t_sig;
    const vui64_t exp_128 = (vui64_t) CONST_VINT64_DW( 128, 128 );
    // d_exp = a_exp - b_exp
    d_exp = vec_subudm (a_exp, b_exp);
    if (vec_cmpud_all_lt (d_exp, exp_128))
      { // if (d_exp < 128) then shift b_sig right
        t_sig = vec_srq (b_sig, (vui128_t) d_exp);
        // retain bits shifted way for X-bits
        l_exp = vec_subudm (exp_128, d_exp);
        x_bits = vec_slq (b_sig, (vui128_t) l_exp);
      }
    else
      { // else all significant bits shifted away
        t_sig = (vui128_t) q_zero;
        // retain original b_sig for X-bits
        x_bits = b_sig;
      }
    // reduce and X-bits for X-bit in IR
    p_odd = vec_addcuq (x_bits, (vui128_t) q_ones);
    // merge shifted b_sig with accumlated X-bit
    b_sig = (vui128_t) vec_or ((vui32_t) t_sig, (vui32_t) p_odd);
  }
```

The inner if/then/else can be replaced with select logic. For example:

```
vb128_t exp_mask = (vb128_t) vec_cmpltud (d_exp, exp_128);
l_exp = vec_subudm (exp_128, d_exp);
t_sig = vec_srq (b_sig, (vui128_t) d_exp);
x_bits = vec_slq (b_sig, (vui128_t) l_exp);
t_sig = vec_seluq ((vui128_t) q_zero, t_sig, exp_mask);
x_bits = vec_seluq (b_sig, x_bits, exp_mask);
```

Now we are ready to compute the significand. For example:

```
// If operands have the same sign then s_sig = a_sig + b_sig
// Otherwise s_sig = a_sig - b_sig
add_sig = vec_adduqm (a_sig, b_sig);
sub_sig = vec_subuqm (a_sig, b_sig);
s_sig = vec_seluq (add_sig, sub_sig, (vb128_t) diff_sign);
```

Now handle the special case of a zero (0.0) result. Either added two zero operands or a subtraction (equal operands with different signs) produced an exact zero result. Exact zero results always have an positive sign, otherwise return return the result with same sign as vfa. For example:

```
if (vec_cmpuq_all_eq (s_sig, (vui128_t) q_zero))
  { // Special case of both zero with different sign
    q_sign = vec_sel (a_sign, (vui32_t) q_zero, diff_sign);
    return vec_xfer_vui32t_2_bin128 (q_sign);
  }
```

Next check for overflow/carry. In this case we need to shift the significand right one bit and increment the exponent by one. Also insure that and X-bit is not lost (shifted away) before we get to the rounding stage. For example:

```
// Check for carry and adjust
else if (vec_cmpuq_all_gt (s_sig, (vui128_t) sigov))
  {
    p_odd = (vui128_t) vec_and ((vui32_t) s_sig, xbitmask);
    s_sig = vec_srqi (s_sig, 1);
    s_sig = (vui128_t) vec_or ((vui32_t) s_sig, (vui32_t) p_odd);
    q_exp = vec_addudm (q_exp, q_one);
  }
```

Otherwise check for underflow (C- and L-bits are 0b0) which requires normalization and may result in a denormal value. Use count leading zeros to estimate the required normalizing left shift. The **IR** internal representation with normally have 12 leading zeros and so we need to adjust for that. Also we need to prevent shifting beyond denormal range (q_exp <= E_min), so use the minimum of leading zero count and the delta between the current (q_exp) exponent and E_min. For example:

```
// Or the significand is below normal range.
// This can happen with subtraction (different signs).
else if (vec_cmpuq_all_le (s_sig, (vui128_t) sigovt))
  {
    const vui64_t exp_12 = { 12, 12 };
    vui64_t c_exp, d_exp;
    vui128_t c_sig;
    c_sig = vec_clzq (s_sig);
    c_exp = vec_splatd ((vui64_t) c_sig, VEC_DW_L);
    c_exp = vec_subudm (c_exp, exp_12);
    d_exp = vec_subudm (q_exp, (vui64_t) exp_min);
    d_exp = vec_minud (c_exp, d_exp);
    // ...
  }
```

The overflow/underflow test above are expensive quadword integer compares and quadword constants. However all that is required is testing the C/L-bits. We can convert these tests into byte compares. For example:

```
      // The C/L-bits are shifted 3-bit left like the significand
      // So the C/L-bits would be constants 16 and 8.
      // But splat-immediate can't generate a +16, so we subtract 1
      // from each and adjust the compares to compensate.
      const vui8_t t_sig_L = vec_splat_u8 (7);
      const vui8_t t_sig_C = vec_splat_u8 (15);
      // ...
      // Issolate CL bits from significand too simplify the compare
#if __BYTE_ORDER__ == __ORDER_LITTLE_ENDIAN__
      vui8_t t_sig = vec_splat ((vui8_t) s_sig, 14);
#else
      vui8_t t_sig = vec_splat ((vui8_t) s_sig, 1);
#endif
        // Check for carry and adjust
        if (vec_all_gt (t_sig, t_sig_C))
        {
          // ...
        }
      // else check for underflow
      else if (vec_all_le (t_sig, t_sig_L))
        {
          // ...
        }
```

If the current exponent is greater then E_min then we can safely use the minimum shift count to:

- Normalize the significand by left shift and adjust the exponent.
- If the adjusted exponent is <= E_min then return denormal by setting the biased exponent to zero (q_zero).

For example:

```
// ...
    if (vec_cmpsd_all_gt ((vi64_t) q_exp, exp_min))
      {
        s_sig = vec_slq (s_sig, (vui128_t) d_exp);
        if (vec_cmpud_all_le (q_exp, c_exp))
          {
            // Exponent is less than the required normalization
            // so return denormal result
            q_exp = q_zero;
          }
        else
          q_exp = vec_subudm (q_exp, d_exp);
      }
    else
      {
        // Exponent is less than or equal to E_min
        // so return denormal result
        q_exp = q_zero;
      }
```

This can be simplified using boolean select logic. Here we replace vec_cmpud_all_le() with vec_cmpgtud() and vec_sel() which generates smaller faster code. For example:

```
// ...
    if (vec_cmpsd_all_gt ((vi64_t) q_exp, exp_min))
      {
        vb64_t exp_mask = vec_cmpgtud (q_exp, c_exp);
        s_sig = vec_slq (s_sig, (vui128_t) d_exp);
        q_exp = vec_subudm (q_exp, d_exp);
        q_exp = (vui64_t) vec_sel (q_zero, (vui64_t) q_exp, exp_mask);
      }
    else // ...
```

After normalization we are ready to round the **IR**, where round-to-odd is the simplest case. For example:

```
// Round to odd from low order GRX-bits
p_tmp = (vui128_t) vec_and ((vui32_t) s_sig, grx_mask);
p_odd = vec_addcuq (p_tmp, (vui128_t) q_ones);
q_sig = vec_srqi (s_sig, 3);
q_sig = (vui128_t) vec_or ((vui32_t) q_sig, (vui32_t) p_odd);
```

After rounding we need one more check for overflow. For round-to-odd we will not see an significand overflow (none-zero C-bit) but we may have overflowed the exponent range.

For normal rounding modes, exponent overflow would generate an infinity. However round-to-odd is a special case that returns the maximum finite value **__FLT128_MAX__**. For example:

```
// Check for exponent overflow -> __FLT128_MAX__
if (vec_cmpud_all_gt (q_exp, q_expmax))
  {
    // return maximum finite exponent and significand
    q_exp = q_expmax;
    q_sig = (vui128_t) sigov;
  }
```

This ends the overall path for finite operands. the last step will merge the sign bit with the significand then insert the exponent. For example:

```
// Merge sign, significand, and exponent into final result
q_sig = (vui128_t) vec_or ((vui32_t) q_sig, q_sign);
result = vec_xsiexpqp (q_sig, q_exp);
```

7.4.2.5.2.1 NaN and Infinity handling for Add

If either operand is a NaN or infinity special handling is required. See PowerISA 3.0B Table 53 Actions for xsaddqp[o].

The PowerISA specifies that the xsaddqpo operation returns a quiet NaN if either operand is a NaN:

- **src1** is a NaN, convert **src1** to a quiet NaN and return that value.
- **src1** is not a NaN, convert **src2** to a quiet NaN and return that value.

For example:

```
// One or both operands are NaN
const vui32_t q_nan = CONST_VINT128_W(0x00008000, 0, 0, 0);
if (vec_all_isnanf128 (vfa))
  {
    // vfa is NaN
    q_sign = a_sign;
    q_sig = (vui128_t) vec_or ((vui32_t) a_sig, q_nan);
    q_exp = a_exp;
  }
else if (vec_all_isnanf128 (vfb))
  {
    // vfb is NaN
    q_sign = b_sign;
    q_sig = (vui128_t) vec_or ((vui32_t) b_sig, q_nan);
    q_exp = b_exp;
  }
else  //
```

The sign. significand, and exponent are combined into a __float128 quiet NaN and returned.

Otherwise one or both operands are infinity. Addition of an infinity and a finite or two infinities (of the same sign) returns infinity. But subtraction of infinities (additional of different signs) returns a default quiet NaN. For example:

```
// else one or both operands are Infinity
  {
    if (vec_cmpud_all_eq (x_exp, q_expnaninf)
        && vec_cmpud_any_ne ((vui64_t) a_sign, (vui64_t) b_sign))
      { // Both operands infinity and opposite sign
        // Inifinty + Infinity (opposite sign) is Default Quiet NaN
        return vec_const_nanf128 ();
      }
    else
      { // Either both operands infinity and same sign
        // Or one infinity and one finite
        if (vec_cmpud_any_eq (a_exp, q_expnaninf))
          {
            // return infinity
            return vfa;
          }
        else
          {
            // return infinity
            return vfb;
          }
      }
  }
```

7.4.2.5.3 Subtract Quad-Precision with Round-to-Odd.

The PVECLIB implementation of **Subtract Quad-Precision with Round-to-Odd** will use the POWER9 xssubqpo instruction if the compile target supports it. Otherwise provide a POWER8 VSX implementation based on the PVECLIB **Add Quad-Precision** implementation. This is based on the observation that;

- floating-point addition becomes subtraction if the signs differ, and
- negation of the second operand allows use of the existing add operation to perform subtraction.

For example:

```
 __binary128
test_vec_subqpo_V0 (__binary128 src1, __binary128 src2)
{
  __binary128 result;
#if defined (_ARCH_PWR9) && (__GNUC__ > 7)
#if defined (__FLOAT128__) && (__GNUC__ > 8)
  // earlier GCC versions generate extra data moves for this.
  result = __builtin_subf128_round_to_odd (src1, src2);
#else
  // No extra data moves here.
  __asm__(
      "xssubqpo %0,%1,%2"
      : "=v" (result)
      : "v" (src1), "v" (src2)
      : );
#endif
#else  // defined (_ARCH_PWR8)
  __binary128 nsrc2 = vec_negf128 (src2);
  return vec_xsaddqpo (src1, nsrc2);
}
```

Unfortunately it is not that simple. The PowerISA specifies that the xssubqpo operation must return the original **src2** operand as a quiet NaN if:

- **src2** is a NaN, and
- **src1** is not a NaN.

See PowerISA 3.0B Table 95 Actions for xssubqp[o].

To insure compatible results for POWER8 and POWER9 implementations we need to avoid negation if **src2** is a NaN. For example:

```
 __binary128
test_vec_subqpo_V1 (__binary128 src1, __binary128 src2)
{
// ...
#else  // defined (_ARCH_PWR8)
  __binary128 nsrc2;
  nsrc2 = vec_self128 (vec_negf128 (src2), src2, vec_isnanf128(src2));
  return vec_xsaddqpo (src1, nsrc2);
}
```

This requires a relatively expensive (∼14 cycle) vec_isnanf128() test and adds ∼12 instructions to the vec_xssubqpo() operation. But this does product correct results for all operand combinations.

Note

> This assumes that vec_xsaddqpo() is fully inlined into vec_xssubqpo().

To get POWER8 an implementation of **Subtract Quad-Precision with Round-to-Odd** with better performance we need to inject the **src2** sign negation strategically into a copy of the **Add Quad-Precision** implementation. This should be after;

- extracting the sign, exponent and significand from the **Quad-Precision** source operands, and
- testing for finite, NaN, and infinity

Once we know that neither operand is NaN we can simply flip the sign bit for source operand 2. For example once we know that both operands are finite:

```
  const vui32_t signmask = CONST_VINT128_W(0x80000000, 0, 0, 0);
//  if (vec_all_isfinitef128 (vfa) && vec_all_isfinitef128 (vfb))
//  The above can be optimized to the following
  if (__builtin_expect (vec_cmpud_all_lt (x_exp, q_expnaninf), 1))
    {
      // Negate sign for subtract, then use add logic
      b_sign = vec_xor (signmask, b_sign);
      // ...
```

7.4.2.5.3.1 NaN and Infinity handling for Subtract

The other case is the **else** leg of the finite case where at least one of the operands is NaN or infinity. For example once we know the none of the operands are NaN:

```
// ... NaNs
else  // Or one or both operands are Infinity
  {
    if (vec_cmpud_all_eq (x_exp, q_expnaninf)
        && vec_cmpud_all_eq ((vui64_t) a_sign, (vui64_t) b_sign))
      { // Both operands infinity and same sign
        // Inifinty - Infinity (same sign) is Default Quiet NaN
        return vec_const_nanf128 ();
      }
    else
      { // Either both operands infinity and different sign
        // Or one infinity and one finite
        if (vec_cmpud_any_eq (a_exp, q_expnaninf))
          {
            // return infinity
            return src1;
          }
        else
          {
            // return -src2 (infinity)
            return vec_negf128(src2);
          }
      }
  }
```

The changes for vec_xssubqpo() from vec_xsaddqpo() are:

- When both operands are infinity and the same sign, then return the default quiet NaN. For vec_xsaddqpo() this tests for signs not equal via vec_cmpud_any_ne().
- Otherwise if **src1** is finite and **src2** is infinity then return negative **src2**.

7.4.2.6 Constants and Masks for Quad-Precision Soft-float

The implementation examples above require a number of __binary128, vector __int128, and vector long long constants. These are used as (special) return values, binary masks, and integer constants for comparison and arithmetic.

In this case finite __float128 (AKA __binary128) constants are provided by the compiler (via -mfloat128) if supported. Literal constants can be defined using the **'q'**/**'Q'** suffix. The compile also provides macros for the IEEE range values; __FLT128_MAX__, __FLT128_MIN__, __FLT128_EPSILON__, and __FLT128_DENORM_MIN__. These macros define the appropriate literal constant using the 'Q' suffix.

The compiler also provides build-in functions for some special constant values; __builtin_infq (), __builtin_huge_valq (), _builtin_nanq (), and __builtin_nansq (). This assumes that you are using a compiler that supports __float128 types and operations and is enabled (via -mfloat128).

PVECLIB has to operated in an environment where the compiler's support for IEEE Float128 is disabled, missing, or incomplete. So this implementation may construct any __binary128 values it may need as vector constants then use the appropriate *xfer* function to create the required scalar value. For example:

```
static inline __binary128
vec_const_inff128 ()
{
  const vui32_t posinf = CONST_VINT128_W (0x7fff0000, 0, 0, 0);
  return vec_xfer_vui32t_2_bin128 (posinf);
}
```

PVECLIB provides splat operations vec_splat_s64() and vec_splat_s128() for small integer constants. For integers in the range -16 <-> 15 the implementation can use vec_splat_s32() then sign extent word elements to doubleword or quadword integer elements. This has the advantage of shorter instruction latency and avoids a vector load from storage. Any vector load has a chance of cache miss and associated cache reload latencies.

For constants outside of this range the implementation uses vec_splats() for which the compiler is expected to load a vector constant from storage. See: Loading small Doubleword constants and Loading small Quadword constants

7.4.2.6.1 Special Constants for Quad-Precision Soft-float

Some special values and masks don't qualify as small integers and need to be constructed as vector constants in hexadecimal. For example:

```
const vui32_t signmask = CONST_VINT128_W(0x80000000, 0, 0, 0);
const vui32_t magmask = CONST_VINT128_W (0x7fffffff, -1, -1, -1);
const vui32_t sigmask = CONST_VINT128_W (0x0000ffff, -1, -1, -1);
const vui32_t expmask = CONST_VINT128_W (0x7fff0000, 0, 0, 0);
const vui32_t hiddenbit = CONST_VINT128_W (0x00010000, 0, 0, 0);
```

For most of these values the compiler will generate a quadword constant in the read-only data segment and generate a vector load for any function that references that constant.

One exception is the quadword sign mask which the GCC compiler recognizes as special case (See↩ : Some special quadword constants). This is a 4 instruction sequence of vector splat immediate words and shift lefts. The nominal execution latency is 6 cycles. The quadword sign mask can be used for both signed quadword integer and quad-precision float values.

It would be useful if we could identify similar (none load) sequences for other mask values. The Quad-Precision Quiet NaN bit mask *QNaNbit*) is similar to the *signmask*. Quiet NaNs **QNaNs** represent the result of specific invalid operations or when *Signaling NaNs* are (Operation Exception) disabled. Quad-Precision Soft-float needs this mask to generate default QNaNs or convert an operand of an invalid operations to return a QNaN result. See↩ : NaN and Infinity handling for Multiply, NaN and Infinity handling for Add, and NaN and Infinity handling for Subtract.

Like the *signmask* it requires the generation of a 0x80000000 constant that can be shifted into position. For example:

```
vui32_t
test_mask128_f128Qbit (void)
{
  //  const vui32_t QNaNbit = CONST_VINT128_W (0x00008000, 0, 0, 0);
  const vui32_t q_zero = CONST_VINT128_W (0, 0, 0, 0);
  const vui32_t q_ones = CONST_VINT128_W (-1, -1, -1, -1);
  vui32_t QNaNbit;
  QNaNbit = vec_sl (q_ones, q_ones);
  return vec_sld (QNaNbit, q_zero, 10);
}
```

Unfortunately the compiler does not recognize this vector constant as a special case. So PVECLIB provides it own implementation. See: vec_mask128_f128Qbit().

7.4.2.6.1.1 Tricks with one's complement

For example the sign and magnitude masks are one's complements of each other. This allows us to get the effect of the *magmask* by reusing the *signmask* and using Vector AND with Complement. For example:

```
// Mask off sign bits so can use quadword integer magnitude compare.
a_mag = (vui128_t) vec_andc_bin128_2_vui32t (vfa, signmask);
b_mag = (vui128_t) vec_andc_bin128_2_vui32t (vfb, signmask);
// Extract the sign bit.
a_sign = vec_and_bin128_2_vui32t (vfa, signmask);
b_sign = vec_and_bin128_2_vui32t (vfb, signmask);
```

The alternative is to use the *magmask* and use Vector AND with Complement to extact the sign bits. For example:

```
// Mask off sign bits so can use quadword integer magnitude compare.
a_mag = (vui128_t) vec_and_bin128_2_vui32t (vfa, magmask);
b_mag = (vui128_t) vec_and_bin128_2_vui32t (vfb, magmask);
// Extract the sign bit.
a_sign = vec_andc_bin128_2_vui32t (vfa, magmask);
b_sign = vec_andc_bin128_2_vui32t (vfb, magmask);
```

This also applies to the vector select (vec_sel()) operation. Vector selects is defined as bit-wise (vra & ∼vrc) | (vrb & vrc) where vrc is the select mask. So the complement of the select mask can be used by swapping select inputs vra/vrb.

Using the complement has no advantage unless we can generate the *magmask* with a shorter/faster (than *signmask*) sequence. Or if the complement is used elsewhere (in the function) and using the complement (for both/multiple operations) allows the implementation to eliminated an expensive constant load.

So lets look at what we can do to generate the magnitude mask (*magmask*). For example we can shift the *all ones* vector right one bit:

```
vui32_t
test_mask128_f128mag (void)
{
  //  const vui32_t magmask = CONST_VINT128_W (0x7fffffff, -1, -1, -1);
  const vui32_t q_ones = CONST_VINT128_W (-1, -1, -1, -1);
  return (vui32_t) vec_srqi ((vui128_t) q_ones, 1);
}
```

which generates:

```
0000000000001560 <test_mask128_f128mag>:
     vspltisw v0,-1
     vspltisb v2,1
     vsr     v2,v0,v2
     blr
```

This sequence only requires 3 instructions and a nominal execution latency of 4 cycles. The vector splat immediate instructions are independent and can execute in parallel. So this technique for generating *magmask* and using it instead of *signmask* is advantageous. Also a clear advantage versus loading *magmask* as a vector constant (at 9+ cycles latency).

7.4.2.6.1.2 Other examples for special constants

Now lets look at how we might generate the significand mask (*sigmask*.) For example we can shift the *all ones* vector right 16-bits:

```
  vui32_t
  test_mask128_f128sig (void)
{
  //  const vui32_t sigmask = CONST_VINT128_W (0x0000ffff, -1, -1, -1);
  const vui32_t q_zero = CONST_VINT128_W (0, 0, 0, 0);
  const vui32_t q_ones = CONST_VINT128_W (-1, -1, -1, -1);
  // Equivalent to shift right 16-bits immediate.
  return vec_sld (q_zero, q_ones, 14);
}
```

which generates:

```
0000000000001580 <test_mask128_f128sig>:
     vspltisw v0,0
     vspltisw v2,-1
     vsldoi  v2,v0,v2,14
     blr
```

Again this sequence only requires 3 instructions and a nominal execution latency of 4 cycles. The vector splat immediate instructions are independent and can execute in parallel. So generating the *sigmask* this way has an advantage over loading a vector constant.

Note

> The quadword constant q_zero and q_ones are already recognized by the compiler as a special case and commonly used in mask generation. As such these constants should be subject to common subexpression optimization and generated once per function.

More examples of special constants are single-bit masks used to compare/test/set the carry/leading bits of the significand. These constants are easy to generate as halfword/word immediate values but need to be shifted left into the high order halfword of the quadword. For example:

```
  vui32_t
  test_mask128_f128Lbit (void)
{
  //  const vui32_t hidden = CONST_VINT128_W (0x00010000, 0, 0, 0);
  const vui32_t q_zero = CONST_VINT128_W (0, 0, 0, 0);
  vui32_t hidden = vec_splat_u32 (1);
  return vec_sld (hidden, q_zero, 14);
}
```

which generates:

```
0000000000001600 <test_mask128_f128Lbit>:
     vspltisw v2,0
     vspltisw v0,1
     vsldoi  v2,v0,v2,14
     blr
```

Again this sequence only requires 3 instructions and a nominal execution latency of 4 cycles. The vector splat immediate instructions are independent and can execute in parallel. We could use the splat halfword immediate for '1' here but word constants are more commonly used and there is no advantage (in instruction count or latency). See: vec_splat_s64() and vec_splat_s128().

Similarly for the carry bit:

```
vui32_t
test_mask128_f128Cbit (void)
{
  //  const vui32_t carry = CONST_VINT128_W (0x00020000, 0, 0, 0);
  const vui32_t q_zero = CONST_VINT128_W (0, 0, 0, 0);
  vui32_t carry = vec_splat_u32 (2);
  return vec_sld (carry, q_zero, 14);
}
```

7.4.2.6.1.3 Some constants are trickier

Some constants are just more difficult to construct. For example:

```
const vui32_t expmask = CONST_VINT128_W (0x7fff0000, 0, 0, 0);
const vui64_t exp_naninf = CONST_VINT128_W (0, 0x7fff, 0, 0x7fff);
```

These constants are used to extract/insert exponents and to test operands for NaN/Infinity. They are used early in many Quad-precision operations, so any delay in getting these constants ready for use, will impact the performance of the whole operation.

Note

> The *expmask* is also used to construct NaN/Infinity values as special return values. But these cases should be rare and so any delays in loading these should not cause significant performance impact.

The tricky part is generating any value with sequence of '1's with '0's on both sides (not left or right justified). Especially within the constraints imposed by POWER8 (PowerISA 2.07B) and earlier machines. Any (none load) sequence require both a left and right shift and any bit (versus octet) shifts will require additional vector constants as shift counts. For example:

```
vui32_t
test_mask128_f128exp_V0 (void)
{
  //  const vui32_t expmask = CONST_VINT128_W (0x7fff0000, 0, 0, 0);
  const vui32_t q_zero = CONST_VINT128_W (0, 0, 0, 0);
  const vui32_t q_ones = CONST_VINT128_W (-1, -1, -1, -1);
  vui32_t expmask;
  // generate {0xffff0000, 0, 0, 0} from q_ones/q_zero
  expmask = vec_sld (q_ones, q_zero, 14);
  // Then shift right halfword to get  {0x7fff0000, 0, 0, 0}
  expmask = (vui32_t) vec_srhi ((vui16_t) expmask, 1);
  return expmask;
}
```

Which generates:

```
0000000000001590 <test_mask128_f128exp_V0>:
    vspltisw v1,0
    vspltisw v0,-1
    vspltish v2,1
    vsldoi  v0,v0,v1,14
    vsrh    v2,v0,v2
    blr
```

This is 5 instructions with a nominal latency of 6 cycles. The halfword shift right requires its own halfword shift count which is unlikely to find a common subexpression within Quad-precision codes.

Another possibility is use Vector Pack Pixel (**vpkpx**) to convert a sequences of words (4x8-bit pixel channels) into a sequence of halfwords (a leading 1-bit channel followed by 3x5-bit channels). The high order channel copies word bit 7 to halfword bit 0. The next three channels are copied 5-bits at a time from the high-order 5-bits of word bytes 1-3. The pack pixel operation requires 8 word elements (256-bits from 2 VRs) to generate 8 halfword elements (one 128-bit vector result).

For example the word element constant 0x00ffffff generates the halfword element 0x7fff. The trick becomes to generate the vectors {0x00ffffff, 0, 0, 0} and {0, 0, 0, 0} using only vector immediate instructions. Where possible we should use built-ins and values common in other operations within PVECLIB and vec_f128_ppc.h. This effectively limits us to the vector splat immediate and Vector Splat Immediate Signed Word (vspltisw) and Vector Shift Left Double by Octet Immediate (vsldoi). For example:

```
vui32_t
 test_mask128_f128exp_V1 (void)
 {
   //  const vui32_t expmask = CONST_VINT128_W (0x7fff0000, 0, 0, 0);
   const vui32_t q_zero = CONST_VINT128_W (0, 0, 0, 0);
   const vui32_t q_ones = CONST_VINT128_W (-1, -1, -1, -1);
   vui32_t expmask;
   // Genrate {0x00ffffff, 0, 0, 0} from q_ones/q_zeros
   expmask = vec_sld (q_ones, q_zero, 13);
   expmask = vec_sld (expmask, expmask, 15);
   // Pack to Pixel to generate {0x7fff0000, 0, 0, 0}
 #if __BYTE_ORDER__ == __ORDER_LITTLE_ENDIAN__
   return (vui32_t) vec_packpx (q_zero, expmask);
 #else
   return (vui32_t) vec_packpx (expmask, q_zero);
 #endif
 }
```

Which generates:

```
0000000000001580 <test_mask128_f128exp_V1>:
    vspltisw v2,0
    vspltisw v0,-1
    vsldoi  v0,v0,v2,13
    vsldoi  v0,v0,v0,15
    vpkpx   v2,v0,v2
    blr
```

This is actually worse than the previous example. Still 5 instructions with a nominal latency of 6-8 cycles. The last 3 instructions are serially dependent on both of the q-ones/q_zero generation operations. The second vsldoi in addition to vpkpx did not improve the performance.

Note

> The vec_packpx() intrinsic is is defined as endian sensitive in the *Power Vector Intrinsic Programming Reference*. So we need the __BYTE_ORDER__ conditional to generate the correct quadword magnitude for both endians.

Time to look more closely at the **vpkpx** operation. Perhaps we can take advantage of different handling of the *auxiliary channel* versus the *color channels*. If we can find a byte value where the low-order bit (bit-7) is '0' while the high-order five bits are 0b11111. For example the byte value 0xf8 (signed char -8) meets these requirements. The word element 0xf8f8f8f8 will pack to the halfword pixel 0x7fff. A vector of these word elements can be generated with vec_splat_s8(-8).

We need the q_zero constant to clear the right three word elements of the left vector and provide zeros to the right 128-bits of the 256-bits required for **vpkpx**. The q_ones constant is not required in this version. For example:

```
vui32_t
test_mask128_f128exp_V2 (void)
{
  //  const vui32_t expmask = CONST_VINT128_W (0x7fff0000, 0, 0, 0);
  const vui32_t q_zero = CONST_VINT128_W (0, 0, 0, 0);
  vui32_t expmask;
  // Genrate {0xf8f8f8f8, 0, 0, 0}
  expmask = (vui32_t) vec_splat_s8 (-8);
  expmask = vec_sld (expmask, q_zero, 12);
  // Pack to Pixel and generate {0x7fff0000, 0, 0, 0}
#if __BYTE_ORDER__ == __ORDER_LITTLE_ENDIAN__
  return (vui32_t) vec_packpx (q_zero, expmask);
#else
  return (vui32_t) vec_packpx (expmask, q_zero);
#endif
}
```

Which generates:

```
0000000000001560 <test_mask128_f128exp_V2>:
    vspltisw v2,0
    vspltisb v0,-8
    vsldoi  v0,v0,v2,12
    vpkpx   v2,v0,v2
    blr
```

This is better at 4 instructions with a nominal latency of 4-6 cycles. The first two (splat immediate) instructions are independent and can execute in parallel, but the vsldoi and vpkpx are serially dependent on the results of both instructions. The q-zero constant is likely to be shared with the generation of other quadword constants. The vec_splat_s8(-8) may be shared with the doubleword exp_naninf constant. See vec_mask64_f128exp().

The doubleword form of the exponent mask (*exp_naninf*) is used as a comparand when we need to test two Quad-precision operands for main path (both finite) or for special case handling (either is NaN/Infinity). We can use a similar technique to that above.

The differences are that the mask is right (versus left) justified and we are splating the mask to both doubleword. For example:

```
vui64_t
test_mask64_f128exp (void)
{
  //const vui32_t exp_naninf = CONST_VINT128_W (0, 0x7fff, 0, 0x7fff);
  const vui32_t q_zero = CONST_VINT128_W (0, 0, 0, 0);
  vui32_t expmask;
  // Genrate {0, 0, 0, 0xf8f8f8f8}
  expmask = (vui32_t) vec_splat_u8 (-8);
  expmask = vec_sld (q_zero, expmask, 4);
  // Pack to Pixel and generate {0, 0x7fff, 0, 0x7fff}
  return (vui64_t) vec_packpx (expmask, expmask);
}
```

Here we can skip the __BYTE_ORDER__ conditional as the vec_packpx() operands are symmetrical. The example above generates:

```
0000000000001560 <test_mask128_f128exp_V2>:
    vspltisw v2,0
    vspltisb v0,-8
    vsldoi  v2,v2,v0,4
    vpkpx   v2,v2,v2
    blr
```

Again 4 instructions with a nominal latency of 4-6 cycles.

Another set of difficult constants to generate are the small powers of 2 (greater than 8). Values 64 and 128 are used for range checks before normalization. These values do not fit into the signed 5-bit (vector splat) immediate fields available to POWER8 and earlier.

The sign-bit constant can be converted to the value 128 with a Vector Shift Left Double by Octet Immediate (vsldoi). For example:

```
static inline vui32_t
test_const128_f128_128(void)
{
  //  const vui32_t signmask = CONST_VINT128_W(0, 0, 0, 128);
  const vui32_t q_zero = CONST_VINT128_W (0, 0, 0, 0);
  const vui32_t q_ones = CONST_VINT128_W (-1, -1, -1, -1);
  vui32_t signmask;
  signmask = vec_sl (q_ones, q_ones);
  return vec_sld (q_zero, signmask, 1);
}
```

This runs 4 instruction with the q_zero/q_ones immediate constants subject to common subexpression evaluation and sharing. This yields a latency of 4 to 6 cycles.

The doubleword form can be based on the above with an additional instruction to splat the low doubleword. For example:

```
static inline vui64_t
test_const64_f128_128(void)
{
  //  const vui32_t dw_128 = CONST_VINT128_W(0, 0, 0, 128);
  const vui32_t q_zero = CONST_VINT128_W (0, 0, 0, 0);
  const vui32_t q_ones = CONST_VINT128_W (-1, -1, -1, -1);
  vui32_t signmask;
  signmask = vec_sl (q_ones, q_ones);
  signmask = vec_sld (q_zero, signmask, 1);
  return vec_mrgald ((vui128_t) signmask, (vui128_t) signmask);
}
```

This works for VSX enable platforms (POWER7 and later) but bumps the latency to 6-8 cycles.

Another possibility is to use the Vector Count Leading Zeros instructions to generate constants for the number of bits in the element. For example:

```
static inline vui64_t
test_const64_f128_64(void)
{
  //  const vui32_t dw_128 = CONST_VINT128_W(0, 64, 0, 64);
  const vui32_t q_zero = CONST_VINT128_W (0, 0, 0, 0);
  // Generate {64, 64} from count leading zeros of {0, 0}
  return  vec_clzd((vui64_t) q_zero);
}
```

This requires only 2 instruction and the q_zero immediate constant is subject to common subexpression evaluation and sharing. This yields a latency of 2 to 4 cycles.

Note

> This technique applies to all element sizes where **Vector Count Leading Zeros** (vclz) are supported. For POWER7 and earlier support byte/halfword/word (constants 8,16, and 32). POWER8 supports Vector Count Leading Zeros Doubleword (vclzd) for constant 64. So far (as of POWER10) there is not quadword (constant 128) form, but for POWER9 we can use **VSX Vector Splat Immediate Byte** and **Vector Extend Sign Byte To Doubleword**.

To generate the doubleword constant 128 we can generate the constant 64 and double it. For example:

```
static inline vui64_t
test_const64_f128_128(void)
{
  //  const vui32_t dw_128 = CONST_VINT128_W(0, 128, 0,128);
  const vui32_t q_zero = CONST_VINT128_W (0, 0, 0, 0);
  // Generate {64, 64} from count leading zeros of {0, 0}
  vui64_t dw64 = vec_clzd((vui64_t) q_zero);
  // Generate {128, 128}
  return vec_addudm (dw64, dw64);
}
```

This runs 3 instruction with the q_zero immediate constant subject to common subexpression evaluation and sharing. This yields a latency of 4 to 6 cycles.

7.4.3 Examples

For example: using the the classification functions for implementing the math library function sine and cosine. The Posix specification requires that special input values are processed without raising extraneous floating point exceptions and return specific floating point values in response. For example the sin() function.

- If the input *value* is NaN then return a NaN.
- If the input *value* is +-0.0 then return *value*.

- If the input *value* is subnormal then return *value*.
- If the input *value* is +-Inf then return a NaN.
- Otherwise compute and return sin(value).

The following code example uses functions from this header to address the POSIX requirements for special values input to sinf128():

```
__binary128
test_sinf128 (__binary128 value)
{
  __binary128 result;
  if (vec_all_isnormalf128 (value))
    {
      // body of taylor series.
      ...
    }
  else
    {
      if (vec_all_isinff128 (value))
        result = vec_const_nanf128 ();
      else
        result = value;
    }
  return result;
}
```

For another example the cos() function.

- If the input *value* is NaN then return a NaN.
- If the input *value* is +-0.0 then return *1.0*.
- If the input *value* is +-Inf then return a NaN.
- Otherwise compute and return cos(value).

The following code example uses functions from this header to address the Posix requirements for special values input to cosf128():

```
__binary128
test_cosf128 (__binary128 value)
{
  __binary128 result;
  if (vec_all_isfinitef128 (value))
    {
      if (vec_all_iszerof128 (value))
        result = 1.0Q;
      else
        {
          // body of taylor series ...
        }
    }
  else
    {
      if (vec_all_isinff128 (value))
        result = vec_const_nanf128 ();
      else
        result = value;
    }
  return result;
}
```

Neither example raises floating point exceptions or sets **errno**, as appropriate for a vector math library.

7.4.4 Performance data

High level performance estimates are provided as an aid to function selection when evaluating algorithms. For background on how *Latency* and *Throughput* are derived see: Performance data.

7.4.5 Function Documentation

7.4.5.1 vec_absf128()

```
static __binary128 vec_absf128 (
             __binary128 f128 )  [inline], [static]
```

Absolute Quad-Precision.

Clear the sign bit of the __float128 input and return the resulting positive __float128 value.

processor	Latency	Throughput
power8	2-11	2/cycle
power9	2	4/cycle

Parameters

f128	a __float128 value containing a signed value.

Returns

a __float128 value with magnitude from f128 and a positive sign of f128.

7.4.5.2 vec_all_isfinitef128()

```
static int vec_all_isfinitef128 (
             __binary128 f128 )  [inline], [static]
```

Return true if the __float128 value is Finite (Not NaN nor Inf).

A IEEE Binary128 finite value has an exponent between 0x0000 and 0x7ffe (a 0x7fff indicates NaN or Inf). The significand can be any value. Using the !vec_all_eq compare conditional verify this condition and avoids a vector -> GPR transfer for platforms before PowerISA-2.07. The sign bit is ignored.

processor	Latency	Throughput
power8	4-20	2/cycle
power9	3	2/cycle

Note

This function will not raise VXSNAN or VXVC (FE_INVALID) exceptions. A normal __float128 compare can.

Parameters

f128	a __float128 value in vector.

Returns

an int containing 0 or 1.

7.4.5.3 vec_all_isinff128()

```
static int vec_all_isinff128 (
             __binary128 f128 )  [inline], [static]
```

Return true if the __float128 value is infinity.

A IEEE Binary128 infinity has a exponent of 0x7fff and significand of all zeros. Using the vec_all_eq compare conditional verifies both conditions and avoids a vector -> GPR transfer for platforms before PowerISA-2.07.

processor	Latency	Throughput
power8	4-20	2/cycle
power9	3	2/cycle

Note

This function will not raise VXSNAN or VXVC (FE_INVALID) exceptions. A normal __float128 compare can.

Parameters

f128	a __float128 value in vector.

Returns

an int containing 0 or 1.

7.4.5.4 vec_all_isnanf128()

```
static int vec_all_isnanf128 (
             __binary128 f128 )  [inline], [static]
```

Return true if the __float128 value is Not a Number (NaN).

A IEEE Binary128 NaN has a exponent of 0x7fff and nonzero significand. Using the combined vec_all_eq / vec_any_gt compare conditional verify both conditions and avoids a vector -> GPR transfer for platforms before PowerISA-2.07. The sign bit is ignored.

processor	Latency	Throughput
power8	6-14	1/cycle
power9	3	2/cycle

Note

This function will not raise VXSNAN or VXVC (FE_INVALID) exceptions. A normal __float128 compare can.

Parameters

f128	a __float128 value in vector.

Returns

an int containing 0 or 1.

7.4.5.5 vec_all_isnormalf128()

```
static int vec_all_isnormalf128 (
             __binary128 f128 )  [inline], [static]
```

Return true if the __float128 value is normal (Not NaN, Inf, denormal, or zero).

A IEEE Binary128 normal value has an exponent between 0x0001 and 0x7ffe (a 0x7fff indicates NaN or Inf). The significand can be any value (expect 0 if the exponent is zero). Using the combined vec_all_ne compares conditional verify both conditions and avoids a vector -> GPR transfer for platforms before PowerISA-2.07. The sign bit is ignored.

processor	Latency	Throughput
power8	4-29	1/cycle
power9	3	2/cycle

Note

This function will not raise VXSNAN or VXVC (FE_INVALID) exceptions. A normal __float128 compare can.

Parameters

f128	a __float128 value in vector.

Returns

an int containing 0 or 1.

7.4.5.6 vec_all_issubnormalf128()

```
static int vec_all_issubnormalf128 (
             __binary128 f128 )  [inline], [static]
```

Return true if the __float128 value is subnormal (denormal).

A IEEE Binary128 subnormal has an exponent of 0x0000 and a nonzero significand. The sign bit is ignored.

Note

This function will not raise VXSNAN or VXVC (FE_INVALID) exceptions. A normal __float128 compare can.

processor	Latency	Throughput
power8	8-29	1/cycle
power9	3	2/cycle

Parameters

f128	a vector of __binary128 values.

Returns

a boolean int, true if the __float128 value is subnormal.

7.4.5.7 vec_all_isunorderedf128()

```
static int vec_all_isunorderedf128 (
             __binary128 vfa,
             __binary128 vfb )  [inline], [static]
```

Return true if either __float128 value (vra, vrb) is NaN.

A IEEE Binary128 NaN has a exponent of 0x7fff and nonzero significand. The sign bit is ignored. For POWER9 and later we use scalar_test_data_class(). Otherwise mask off the sign bit and compare greater than unsigned quadword to the integer equivalent of Quad-Precision infinity.

Note

This function will not raise VXSNAN or VXVC (FE_INVALID) exceptions. A normal __float128 compare can.

processor	Latency	Throughput
power8	16-25	1/cycle
power9	6	1/cycle

Parameters

vfa	128-bit vector treated as a scalar __binary128.
vfb	128-bit vector treated as a scalar __binary128.

Returns

a boolean int, true if either __float128 value (vra, vrb) is NaN.

7.4.5.8 vec_all_iszerof128()

```
static int vec_all_iszerof128 (
             __binary128 f128 )  [inline], [static]
```

Return true if the __float128 value is +-0.0.

A IEEE Binary128 zero has an exponent of 0x0000 and a zero significand. The sign bit is ignored.

Note

This function will not raise VXSNAN or VXVC (FE_INVALID) exceptions. A normal __float128 compare can.

processor	Latency	Throughput
power8	4-20	1/cycle
power9	3	2/cycle

Parameters

f128	a vector of __binary64 values.

Returns

a boolean int, true if the __float128 value is +/- zero.

7.4.5.9 vec_and_bin128_2_vui32t()

```
static vui32_t vec_and_bin128_2_vui32t (
             __binary128 f128,
             vui32_t mask )  [inline], [static]
```

Transfer a quadword from a __binary128 scalar to a vector int and logical AND with a mask.

The compiler does not allow direct transfer (assignment or type cast) between __binary128 (__float128) scalars and vector types. This despite the fact the the ABI and ISA require __binary128 in vector registers (VRs).

Note

this function uses a union to effect the (logical) transfer. The compiler should not generate any code for this.

Parameters

f128	a __binary128 floating point scalar value.
mask	a vector unsigned int

Returns

The original value ANDed with mask as a 128-bit vector int.

7.4.5.10 vec_andc_bin128_2_vui128t()

```
static vui128_t vec_andc_bin128_2_vui128t (
             __binary128 f128,
             vui128_t mask )  [inline], [static]
```

Transfer a quadword from a __binary128 scalar to a vector __int128 and logical AND Compliment with mask.

The compiler does not allow direct transfer (assignment or type cast) between __binary128 (__float128) scalars and vector types. This despite the fact the the ABI and ISA require __binary128 in vector registers (VRs).

Note

this function uses a union to effect the (logical) transfer. The compiler should not generate any code for this.

Parameters

f128	a __binary128 floating point scalar value.
mask	a vector unsigned __int128

Returns

The original value ANDed with mask as a 128-bit vector int.

7.4.5.11 vec_andc_bin128_2_vui32t()

```
static vui32_t vec_andc_bin128_2_vui32t (
             __binary128 f128,
             vui32_t mask )  [inline], [static]
```

Transfer a quadword from a __binary128 scalar to a vector int and logical AND Compliment with mask.

The compiler does not allow direct transfer (assignment or type cast) between __binary128 (__float128) scalars and vector types. This despite the fact the the ABI and ISA require __binary128 in vector registers (VRs).

Note

this function uses a union to effect the (logical) transfer. The compiler should not generate any code for this.

Parameters

f128	a __binary128 floating point scalar value.
mask	a vector unsigned int

Returns

The original value ANDed with mask as a 128-bit vector int.

7.4.5.12 vec_cmpeqtoqp()

```
static vb128_t vec_cmpeqtoqp (
             __binary128 vfa,
             __binary128 vfb )  [inline], [static]
```

Vector Compare Equal (Total-order) Quad-Precision.

Compare Binary-float 128-bit values and return all '1's, if vfa == vfb, otherwise all '0's. Zeros, Infinities and NaN of the same sign compare equal.

For POWER9 (PowerISA 3.0B) or later, use a VSX Scalar Compare Unordered Quad-Precision or a VSX Scalar Compare Equal Quad-Precision instruction. Otherwise use vector __int128 arithmetic and logical operations to implement the equivalent Quad-precision floating-point operation. This leverages operations from vec_int128_ppc.h.

Note

This operation *may not* follow the IEEE standard relative to signed zero, or NaN comparison. However if the hardware target includes an instruction that does implement the IEEE standard, the implementation may use that. This relaxed implementation may be useful for implementations on POWER8 and earlier. Especially for soft-float implementations where it is known these special cases do not occur.

processor	Latency	Throughput
power8	6	2/cycle
power9	3	2/cycle

Parameters

vfa	128-bit vector treated as a scalar __binary128.
vfb	128-bit vector treated as a scalar __binary128.

Returns

128-bit vector boolean reflecting __binary128 compare equal.

7.4.5.13 vec_cmpequqp()

```
static vb128_t vec_cmpequqp (
             __binary128 vfa,
             __binary128 vfb )  [inline], [static]
```

Vector Compare Equal (Unordered) Quad-Precision.

Compare Binary-float 128-bit values and return all '1's, if vfa == vfb, otherwise all '0's. Zeros of either sign compare equal. Infinities of the same sign compare equal. A NaN in either or both operands compare unequal.

For POWER9 (PowerISA 3.0B) or later, use a VSX Scalar Compare Unordered Quad-Precision or (POWER10) VSX Scalar Compare Equal Quad-Precision instruction. Otherwise use vector __int128 arithmetic and logical operations to implement the equivalent Quad-precision floating-point operation. This leverages operations from vec_int128_ppc.h.

Note

This operation *may not* follow the PowerISA relative to NaN comparison and setting the FPSCR. However if the hardware target includes an instruction that does implement the IEEE standard, the implementation may use that. This relaxed implementation may be useful for implementations on POWER8 and earlier. Especially for soft-float implementations where it is known these special cases do not occur.

processor	Latency	Throughput
power8	18-30	1/cycle
power9	3	2/cycle

Parameters

vfa	128-bit vector treated as a scalar __binary128.
vfb	128-bit vector treated as a scalar __binary128.

Returns

128-bit vector boolean reflecting __binary128 compare equal.

7.4.5.14 vec_cmpequzqp()

```
static vb128_t vec_cmpequzqp (
             __binary128 vfa,
             __binary128 vfb )  [inline], [static]
```

Vector Compare Equal (Zero-unordered) Quad-Precision.

Compare Binary-float 128-bit values and return all '1's, if vfa == vfb, otherwise all '0's. Zeros of either sign compare equal. Infinities and NaNs of the same sign compare equal.

For POWER9 (PowerISA 3.0B) or later, use a VSX Scalar Compare Unordered Quad-Precision or a VSX Scalar Compare Equal Quad-Precision instruction. Otherwise use vector __int128 arithmetic and logical operations to implement the equivalent Quad-precision floating-point operation. This leverages operations from vec_int128_ppc.h.

Note

> This operation *may not* follow the IEEE standard relative NaN comparison. However if the hardware target includes an instruction that does implement the IEEE standard, the implementation may use that. This relaxed implementation may be useful for implementations on POWER8 and earlier. Especially for soft-float implementations where it is known these special cases do not occur.

processor	Latency	Throughput
power8	10	1/cycle
power9	3	2/cycle

Parameters

vfa	128-bit vector treated as a scalar __binary128.
vfb	128-bit vector treated as a scalar __binary128.

Returns

128-bit vector boolean reflecting __binary128 compare equal.

7.4.5.15 vec_cmpgetoqp()

```
static vb128_t vec_cmpgetoqp (
             __binary128 vfa,
             __binary128 vfb )  [inline], [static]
```

Vector Compare Greater Than or Equal (Total-order) Quad-Precision.

Compare Binary-float 128-bit values and return all '1's, if vfa >= vfb, otherwise all '0's. Zeros, Infinities and NaNs are compared as signed values. Infinities and NaNs have the highest/lowest magnitudes.

For POWER9 (PowerISA 3.0B) or later, use a VSX Scalar Compare Unordered Quad-Precision or (POWER10) VSX Scalar Compare Greater Than Quad-Precision instruction. Otherwise comparands are converted to unsigned integer magnitudes before using vector __int128 comparison to implement the equivalent Quad-precision floating-point operation. This leverages operations from vec_int128_ppc.h.

Note

This operation *may not* follow the IEEE standard relative to signed zero, or NaN comparison. However if the hardware target includes an instruction that does implement the IEEE standard, the implementation may use that. This relaxed implementation may be useful for implementations on POWER8 and earlier. Especially for soft-float implementations where it is known these special cases do not occur.

processor	Latency	Throughput
power8	26-35	2/cycle
power9	3	2/cycle

Parameters

vfa	128-bit vector treated as a scalar __binary128.
vfb	128-bit vector treated as a scalar __binary128.

Returns

128-bit vector boolean reflecting __binary128 compare greater than or equal.

7.4.5.16 vec_cmpgeuqp()

```
static vb128_t vec_cmpgeuqp (
            __binary128 vfa,
            __binary128 vfb )  [inline], [static]
```

Vector Compare Greater Than or Equal (Unordered) Quad-Precision.

Compare Binary-float 128-bit values and return all '1's, if vfa >= vfb, otherwise all '0's. Zeros of either sign are converted to +0. Infinities of different signs compare ordered. A NaN in either or both operands compare unordered.

For POWER9 (PowerISA 3.0B) or later, use a VSX Scalar Compare Unordered Quad-Precision or (POWER10) VSX Scalar Compare Greater Than Quad-Precision instruction. Otherwise comparands are converted to unsigned integer magnitudes before using vector __int128 comparison to implement the equivalent Quad-precision floating-point operation. This leverages operations from vec_int128_ppc.h.

Note

> This operation *may not* follow the PowerISA relative to NaN comparison and setting the FPSCR. However if the hardware target includes an instruction that does implement the IEEE standard, the implementation may use that. This relaxed implementation may be useful for implementations on POWER8 and earlier. Especially for soft-float implementations where it is known these special cases do not occur.

processor	Latency	Throughput
power8	28-37	2/cycle
power9	3	2/cycle

Parameters

vfa	128-bit vector treated as a scalar __binary128.
vfb	128-bit vector treated as a scalar __binary128.

Returns

> 128-bit vector boolean reflecting __binary128 compare greater than or equal.

7.4.5.17 vec_cmpgeuzqp()

```
static vb128_t vec_cmpgeuzqp (
             __binary128 vfa,
             __binary128 vfb )  [inline], [static]
```

Vector Compare Greater Than Or Equal (Zero-unordered) Quad-Precision.

Compare Binary-float 128-bit values and return all '1's, if vfa >= vfb, otherwise all '0's. Zeros of either sign are converted to +0. Infinities and NaNs are compared as signed values. Infinities and NaNs have the highest/lowest magnitudes.

For POWER9 (PowerISA 3.0B) or later, use a VSX Scalar Compare Unordered Quad-Precision or (POWER10) VSX Scalar Compare Greater Than Quad-Precision instruction. Otherwise comparands are converted to unsigned integer magnitudes before using vector __int128 comparison to implement the equivalent Quad-precision floating-point operation. This leverages operations from vec_int128_ppc.h.

Note

> This operation *may not* follow the IEEE standard relative to NaN comparison. However if the hardware target includes an instruction that does implement the IEEE standard, the implementation may use that. This relaxed implementation may be useful for implementations on POWER8 and earlier. Especially for soft-float implementations where it is known these special cases do not occur.

processor	Latency	Throughput
power8	28-37	2/cycle
power9	3	2/cycle

Parameters

vfa	128-bit vector treated as a scalar __binary128.
vfb	128-bit vector treated as a scalar __binary128.

Returns

128-bit vector boolean reflecting __binary128 compare greater than or equal.

7.4.5.18 vec_cmpgttoqp()

```
static vb128_t vec_cmpgttoqp (
             __binary128 vfa,
             __binary128 vfb )  [inline], [static]
```

Vector Compare Greater Than (Total-order) Quad-Precision.

Compare Binary-float 128-bit values and return all '1's, if vfa $>$ vfb, otherwise all '0's. Zeros, Infinities and NaNs are compared as signed values. Infinities and NaNs have the highest/lowest magnitudes.

For POWER9 (PowerISA 3.0B) or later, use a VSX Scalar Compare Unordered Quad-Precision or (POWER10) VSX Scalar Compare Greater Than Quad-Precision instruction. Otherwise comparands are converted to unsigned integer magnitudes before using vector __int128 comparison to implement the equivalent Quad-precision floating-point operation. This leverages operations from vec_int128_ppc.h.

Note

This operation *may not* follow the IEEE standard relative to signed zero, or NaN comparison. However if the hardware target includes an instruction that does implement the IEEE standard, the implementation may use that. This relaxed implementation may be useful for implementations on POWER8 and earlier. Especially for soft-float implementations where it is known these special cases do not occur.

processor	Latency	Throughput
power8	26-35	2/cycle
power9	3	2/cycle

Parameters

vfa	128-bit vector treated as a scalar __binary128.
vfb	128-bit vector treated as a scalar __binary128.

Returns

128-bit vector boolean reflecting __binary128 compare greater than.

7.4.5.19 vec_cmpgtuqp()

```
static vb128_t vec_cmpgtuqp (
             __binary128 vfa,
             __binary128 vfb )  [inline], [static]
```

Vector Compare Greater Than (Unordered) Quad-Precision.

Compare Binary-float 128-bit values and return all '1's, if vfa $>$ vfb, otherwise all '0's. Zeros of either sign are converted to +0. Infinities of different signs compare ordered. A NaN in either or both operands compare unordered.

For POWER9 (PowerISA 3.0B) or later, use a VSX Scalar Compare Unordered Quad-Precision or (POWER10) VSX Scalar Compare Greater Than Quad-Precision instruction. Otherwise comparands are converted to unsigned integer magnitudes before using vector __int128 comparison to implement the equivalent Quad-precision floating-point operation. This leverages operations from vec_int128_ppc.h.

Note

This operation *may not* follow the PowerISA relative to NaN comparison and setting the FPSCR. However if the hardware target includes an instruction that does implement the IEEE standard, the implementation may use that. This relaxed implementation may be useful for implementations on POWER8 and earlier. Especially for soft-float implementations where it is known these special cases do not occur.

processor	Latency	Throughput
power8	28-37	2/cycle
power9	3	2/cycle

Parameters

vfa	128-bit vector treated as a scalar __binary128.
vfb	128-bit vector treated as a scalar __binary128.

Returns

128-bit vector boolean reflecting __binary128 compare greater than.

7.4.5.20 vec_cmpgtuzqp()

```
static vb128_t vec_cmpgtuzqp (
             __binary128 vfa,
             __binary128 vfb )  [inline], [static]
```

Vector Compare Greater Than (Zero-unordered) Quad-Precision.

Compare Binary-float 128-bit values and return all '1's, if vfa > vfb, otherwise all '0's. Zeros of either sign are converted to +0. Infinities and NaNs are compared as signed values. Infinities and NaNs have the highest/lowest magnitudes.

For POWER9 (PowerISA 3.0B) or later, use a VSX Scalar Compare Unordered Quad-Precision or (POWER10) VSX Scalar Compare Greater Than Quad-Precision instruction. Otherwise comparands are converted to unsigned integer magnitudes before using vector __int128 comparison to implement the equivalent Quad-precision floating-point operation. This leverages operations from vec_int128_ppc.h.

Note

> This operation *may not* follow the IEEE standard relative to NaN comparison. However if the hardware target includes an instruction that does implement the IEEE standard, the implementation may use that. This relaxed implementation may be useful for implementations on POWER8 and earlier. Especially for soft-float implementations where it is known these special cases do not occur.

processor	Latency	Throughput
power8	28-37	2/cycle
power9	3	2/cycle

Parameters

vfa	128-bit vector treated as a scalar __binary128.
vfb	128-bit vector treated as a scalar __binary128.

Returns

> 128-bit vector boolean reflecting __binary128 compare greater than.

7.4.5.21 vec_cmpletoqp()

```
static vb128_t vec_cmpletoqp (
            __binary128 vfa,
            __binary128 vfb )  [inline], [static]
```

Vector Compare Less Than or Equal (Total-order) Quad-Precision.

Compare Binary-float 128-bit values and return all '1's, if vfa <= vfb, otherwise all '0's. Zeros, Infinities and NaNs are compared as signed values. Infinities and NaNs have the highest/lowest magnitudes.

For POWER9 (PowerISA 3.0B) or later, use a VSX Scalar Compare Unordered Quad-Precision or (POWER10) VSX Scalar Compare Greater Than Quad-Precision instruction. Otherwise comparands are converted to unsigned integer magnitudes before using vector __int128 comparison to implement the equivalent Quad-precision floating-point operation. This leverages operations from vec_int128_ppc.h.

Note

This operation *may not* follow the IEEE standard relative to signed zero, or NaN comparison. However if the hardware target includes an instruction that does implement the IEEE standard, the implementation may use that. This relaxed implementation may be useful for implementations on POWER8 and earlier. Especially for soft-float implementations where it is known these special cases do not occur.

processor	Latency	Throughput
power8	26-35	2/cycle
power9	3	2/cycle

Parameters

vfa	128-bit vector treated as a scalar __binary128.
vfb	128-bit vector treated as a scalar __binary128.

Returns

128-bit vector boolean reflecting __binary128 compare less than or equal.

7.4.5.22 vec_cmpleuqp()

```
static vb128_t vec_cmpleuqp (
            __binary128 vfa,
            __binary128 vfb )  [inline], [static]
```

Vector Compare Less Than or Equal (Unordered) Quad-Precision.

Compare Binary-float 128-bit values and return all '1's, if vfa <= vfb, otherwise all '0's. Zeros of either sign are converted to +0. Infinities of different signs compare ordered. A NaN in either or both operands compare unordered.

For POWER9 (PowerISA 3.0B) or later, use a VSX Scalar Compare Unordered Quad-Precision or (POWER10) VSX Scalar Compare Greater Than Quad-Precision instruction. Otherwise comparands are converted to unsigned integer magnitudes before using vector __int128 comparison to implement the equivalent Quad-precision floating-point operation. This leverages operations from vec_int128_ppc.h.

Note

This operation *may not* follow the PowerISA relative to NaN comparison and setting the FPSCR. However if the hardware target includes an instruction that does implement the IEEE standard, the implementation may use that. This relaxed implementation may be useful for implementations on POWER8 and earlier. Especially for soft-float implementations where it is known these special cases do not occur.

processor	Latency	Throughput
power8	28-37	2/cycle
power9	3	2/cycle

Parameters

vfa	128-bit vector treated as a scalar __binary128.
vfb	128-bit vector treated as a scalar __binary128.

Returns

128-bit vector boolean reflecting __binary128 compare less than or equal.

7.4.5.23 vec_cmpleuzqp()

```
static vb128_t vec_cmpleuzqp (
            __binary128 vfa,
            __binary128 vfb )  [inline], [static]
```

Vector Compare Less Than or Equal (Zero-unordered) Quad-Precision.

Compare Binary-float 128-bit values and return all '1's, if vfa <= vfb, otherwise all '0's. Zeros of either sign are converted to +0. Infinities and NaNs are compared as signed values. Infinities and NaNs have the highest/lowest magnitudes.

For POWER9 (PowerISA 3.0B) or later, use a VSX Scalar Compare Unordered Quad-Precision or (POWER10) VSX Scalar Compare Greater Than Quad-Precision instruction. Otherwise comparands are converted to unsigned integer magnitudes before using vector __int128 comparison to implement the equivalent Quad-precision floating-point operation. This leverages operations from vec_int128_ppc.h.

Note

This operation *may not* follow the IEEE standard relative to NaN comparison. However if the hardware target includes an instruction that does implement the IEEE standard, the implementation may use that. This relaxed implementation may be useful for implementations on POWER8 and earlier. Especially for soft-float implementations where it is known these special cases do not occur.

processor	Latency	Throughput
power8	28-37	2/cycle
power9	3	2/cycle

Parameters

vfa	128-bit vector treated as a scalar __binary128.
vfb	128-bit vector treated as a scalar __binary128.

Returns

128-bit vector boolean reflecting __binary128 compare less than or equal.

7.4.5.24 vec_cmplttoqp()

```
static vb128_t vec_cmplttoqp (
             __binary128 vfa,
             __binary128 vfb )  [inline], [static]
```

Vector Compare Less Than (Total-order) Quad-Precision.

Compare Binary-float 128-bit values and return all '1's, if vfa < vfb, otherwise all '0's. Zeros, Infinities and NaNs are compared as signed values. Infinities and NaNs have the highest/lowest magnitudes.

For POWER9 (PowerISA 3.0B) or later, use a VSX Scalar Compare Unordered Quad-Precision or (POWER10) VSX Scalar Compare Greater Than Quad-Precision instruction. Otherwise comparands are converted to unsigned integer magnitudes before using vector __int128 comparison to implement the equivalent Quad-precision floating-point operation. This leverages operations from vec_int128_ppc.h.

Note

This operation *may not* follow the IEEE standard relative to signed zero, or NaN comparison. However if the hardware target includes an instruction that does implement the IEEE standard, the implementation may use that. This relaxed implementation may be useful for implementations on POWER8 and earlier. Especially for soft-float implementations where it is known these special cases do not occur.

processor	Latency	Throughput
power8	26-35	2/cycle
power9	3	2/cycle

Parameters

vfa	128-bit vector treated as a scalar __binary128.
vfb	128-bit vector treated as a scalar __binary128.

Returns

128-bit vector boolean reflecting __binary128 compare less than.

7.4.5.25 vec_cmpltuqp()

```
static vb128_t vec_cmpltuqp (
             __binary128 vfa,
             __binary128 vfb )  [inline], [static]
```

Vector Compare Less Than (Unordered) Quad-Precision.

Compare Binary-float 128-bit values and return all '1's, if vfa < vfb, otherwise all '0's. Zeros of either sign are converted to +0. Infinities of different signs compare ordered. A NaN in either or both operands compare unordered.

For POWER9 (PowerISA 3.0B) or later, use a VSX Scalar Compare Unordered Quad-Precision or (POWER10) VSX Scalar Compare Greater Than Quad-Precision instruction. Otherwise comparands are converted to unsigned integer magnitudes before using vector __int128 comparison to implement the equivalent Quad-precision floating-point operation. This leverages operations from vec_int128_ppc.h.

Note

> This operation *may not* follow the PowerISA relative to NaN comparison and setting the FPSCR. However if the hardware target includes an instruction that does implement the IEEE standard, the implementation may use that. This relaxed implementation may be useful for implementations on POWER8 and earlier. Especially for soft-float implementations where it is known these special cases do not occur.

processor	Latency	Throughput
power8	28-37	2/cycle
power9	3	2/cycle

Parameters

vfa	128-bit vector treated as a scalar __binary128.
vfb	128-bit vector treated as a scalar __binary128.

Returns

> 128-bit vector boolean reflecting __binary128 compare less than.

7.4.5.26 vec_cmpltuzqp()

```
static vb128_t vec_cmpltuzqp (
            __binary128 vfa,
            __binary128 vfb )  [inline], [static]
```

Vector Compare Less Than (Zero-unordered) Quad-Precision.

Compare Binary-float 128-bit values and return all '1's, if vfa < vfb, otherwise all '0's. Zeros of either sign are converted to +0. Infinities and NaNs are compared as signed values. Infinities and NaNs have the highest/lowest magnitudes.

For POWER9 (PowerISA 3.0B) or later, use a VSX Scalar Compare Unordered Quad-Precision or (POWER10) VSX Scalar Compare Greater Than Quad-Precision instruction. Otherwise comparands are converted to unsigned integer magnitudes before using vector __int128 comparison to implement the equivalent Quad-precision floating-point operation. This leverages operations from vec_int128_ppc.h.

Note

This operation *may not* follow the IEEE standard relative to NaN comparison. However if the hardware target includes an instruction that does implement the IEEE standard, the implementation may use that. This relaxed implementation may be useful for implementations on POWER8 and earlier. Especially for soft-float implementations where it is known these special cases do not occur.

processor	Latency	Throughput
power8	28-37	2/cycle
power9	3	2/cycle

Parameters

vfa	128-bit vector treated as a scalar __binary128.
vfb	128-bit vector treated as a scalar __binary128.

Returns

128-bit vector boolean reflecting __binary128 compare less than.

7.4.5.27 vec_cmpnetoqp()

```
static vb128_t vec_cmpnetoqp (
            __binary128 vfa,
            __binary128 vfb )  [inline], [static]
```

Vector Compare Not Equal (Total-order) Quad-Precision.

Compare Binary-float 128-bit values and return all '1's, if vfa != vfb, otherwise all '0's. Zeros, Infinities and NaN of the same sign compare equal.

For POWER9 (PowerISA 3.0B) or later, use a VSX Scalar Compare Unordered Quad-Precision or a VSX Scalar Compare Equal Quad-Precision instruction. Otherwise use vector __int128 arithmetic and logical operations to implement the equivalent Quad-precision floating-point operation. This leverages operations from vec_int128_ppc.h.

Note

This operation *may not* follow the IEEE standard relative to signed zero, or NaN comparison. However if the hardware target includes an instruction that does implement the IEEE standard, the implementation may use that. This relaxed implementation may be useful for implementations on POWER8 and earlier. Especially for soft-float implementations where it is known these special cases do not occur.

processor	Latency	Throughput
power8	6	2/cycle
power9	3	2/cycle

Parameters

vfa	128-bit vector treated as a scalar __binary128.
vfb	128-bit vector treated as a scalar __binary128.

Returns

128-bit vector boolean reflecting __binary128 compare not equal.

7.4.5.28 vec_cmpneuqp()

```
static vb128_t vec_cmpneuqp (
             __binary128 vfa,
             __binary128 vfb )  [inline], [static]
```

Vector Compare Not Equal (Unordered) Quad-Precision.

Compare Binary-float 128-bit values and return all '1's, if vfa == vfb, otherwise all '0's. Zeros of either sign compare equal. Infinities of the same sign compare equal. A NaN in either or both operands compare unequal.

For POWER9 (PowerISA 3.0B) or later, use a VSX Scalar Compare Unordered Quad-Precision or (POWER10) VSX Scalar Compare Equal Quad-Precision instruction. Otherwise use vector __int128 arithmetic and logical operations to implement the equivalent Quad-precision floating-point operation. This leverages operations from vec_int128_ppc.h.

Note

This operation *may not* follow the PowerISA relative to NaN comparison and setting the FPSCR. However if the hardware target includes an instruction that does implement the IEEE standard, the implementation may use that. This relaxed implementation may be useful for implementations on POWER8 and earlier. Especially for soft-float implementations where it is known these special cases do not occur.

processor	Latency	Throughput
power8	18-30	1/cycle
power9	3	2/cycle

Parameters

vfa	128-bit vector treated as a scalar __binary128.
vfb	128-bit vector treated as a scalar __binary128.

Returns

128-bit vector boolean reflecting __binary128 compare not equal.

7.4.5.29 vec_cmpneuzqp()

```
static vb128_t vec_cmpneuzqp (
             __binary128 vfa,
             __binary128 vfb )  [inline], [static]
```

Vector Compare Not Equal (Zero-unordered) Quad-Precision.

Compare Binary-float 128-bit values and return all '1's, if vfa != vfb, otherwise all '0's. Zeros of either sign compare equal. Infinities and NaNs of the same sign compare equal.

For POWER9 (PowerISA 3.0B) or later, use a VSX Scalar Compare Unordered Quad-Precision or a VSX Scalar Compare Equal Quad-Precision instruction. Otherwise use vector __int128 arithmetic and logical operations to implement the equivalent Quad-precision floating-point operation. This leverages operations from vec_int128_ppc.h.

Note

This operation *may not* follow the IEEE standard relative to NaN comparison. However if the hardware target includes an instruction that does implement the IEEE standard, the implementation may use that. This relaxed implementation may be useful for implementations on POWER8 and earlier. Especially for soft-float implementations where it is known these special cases do not occur.

processor	Latency	Throughput
power8	10	1/cycle
power9	3	2/cycle

Parameters

vfa	128-bit vector treated as a scalar __binary128.
vfb	128-bit vector treated as a scalar __binary128.

Returns

128-bit vector boolean reflecting __binary128 compare not equal.

7.4.5.30 vec_cmpqp_all_eq()

```
static int vec_cmpqp_all_eq (
             __binary128 vfa,
             __binary128 vfb )  [inline], [static]
```

Vector Compare all Equal (Unordered) Quad-Precision.

Compare Binary-float 128-bit values and return 1, if vfa == vfb, otherwise 0. Zeros of either sign compare equal. Infinities of the same sign compare equal. A NaN in either or both operands compare unequal.

For POWER9 (PowerISA 3.0B) or later, use a VSX Scalar Compare Unordered Quad-Precision. Otherwise use vector __int128 arithmetic and logical operations to implement the equivalent Quad-precision floating-point operation. This leverages operations from vec_int128_ppc.h.

Note

This operation *may not* follow the PowerISA relative to NaN comparison and setting the FPSCR. However if the hardware target includes an instruction that does implement the IEEE standard, the implementation may use that. This relaxed implementation may be useful for implementations on POWER8 and earlier. Especially for soft-float implementations where it is known these special cases do not occur.

processor	Latency	Throughput
power8	18-30	1/cycle
power9	3	2/cycle

Parameters

vfa	128-bit vector treated as a scalar __binary128.
vfb	128-bit vector treated as a scalar __binary128.

Returns

int boolean reflecting __binary128 compare equal.

7.4.5.31 vec_cmpqp_all_ge()

```
static int vec_cmpqp_all_ge (
             __binary128 vfa,
             __binary128 vfb )  [inline], [static]
```

Vector Compare all Greater Than Or Equal (Unordered) Quad-Precision.

Compare Binary-float 128-bit values and return 1 (true), if vfa >= vfb, otherwise 0 (false). Zeros of either sign are converted to +0. Infinities of different signs compare ordered. A NaN in either or both operands compare unordered.

For POWER9 (PowerISA 3.0B) or later, use a VSX Scalar Compare Unordered Quad-Precision. Otherwise comparands are converted to unsigned integer magnitudes before using vector __int128 comparison to implement the equivalent Quad-precision floating-point operation. This leverages operations from vec_int128_ppc.h.

Note

This operation *may not* follow the PowerISA relative to NaN comparison and setting the FPSCR. However if the hardware target includes an instruction that does implement the IEEE standard, the implementation may use that. This relaxed implementation may be useful for implementations on POWER8 and earlier. Especially for soft-float implementations where it is known these special cases do not occur.

processor	Latency	Throughput
power8	28-37	2/cycle
power9	3	2/cycle

Parameters

vfa	128-bit vector treated as a scalar __binary128.
vfb	128-bit vector treated as a scalar __binary128.

Returns

int boolean reflecting __binary128 greater than.

7.4.5.32 vec_cmpqp_all_gt()

```
static int vec_cmpqp_all_gt (
            __binary128 vfa,
            __binary128 vfb )  [inline], [static]
```

Vector Compare all Greater Than (Unordered) Quad-Precision.

Compare Binary-float 128-bit values and return 1 (true), if vfa $>$ vfb, otherwise 0 (false). Zeros of either sign are converted to +0. Infinities of different signs compare ordered. A NaN in either or both operands compare unordered.

For POWER9 (PowerISA 3.0B) or later, use a VSX Scalar Compare Unordered Quad-Precision. Otherwise comparands are converted to unsigned integer magnitudes before using vector __int128 comparison to implement the equivalent Quad-precision floating-point operation. This leverages operations from vec_int128_ppc.h.

Note

This operation *may not* follow the PowerISA relative to NaN comparison and setting the FPSCR. However if the hardware target includes an instruction that does implement the IEEE standard, the implementation may use that. This relaxed implementation may be useful for implementations on POWER8 and earlier. Especially for soft-float implementations where it is known these special cases do not occur.

processor	Latency	Throughput
power8	28-37	2/cycle
power9	3	2/cycle

Parameters

vfa	128-bit vector treated as a scalar __binary128.
vfb	128-bit vector treated as a scalar __binary128.

Returns

int boolean reflecting __binary128 greater than.

7.4.5.33 vec_cmpqp_all_le()

```
static int vec_cmpqp_all_le (
             __binary128 vfa,
             __binary128 vfb )  [inline], [static]
```

Vector Compare all Less Than Or Equal (Unordered) Quad-Precision.

Compare Binary-float 128-bit values and return 1 (true), if vfa <= vfb, otherwise 0 (false). Zeros of either sign are converted to +0. Infinities of different signs compare ordered. A NaN in either or both operands compare unordered.

For POWER9 (PowerISA 3.0B) or later, use a VSX Scalar Compare Unordered Quad-Precision. Otherwise comparands are converted to unsigned integer magnitudes before using vector __int128 comparison to implement the equivalent Quad-precision floating-point operation. This leverages operations from vec_int128_ppc.h.

Note

> This operation *may not* follow the PowerISA relative to NaN comparison and setting the FPSCR. However if the hardware target includes an instruction that does implement the IEEE standard, the implementation may use that. This relaxed implementation may be useful for implementations on POWER8 and earlier. Especially for soft-float implementations where it is known these special cases do not occur.

processor	Latency	Throughput
power8	28-37	2/cycle
power9	3	2/cycle

Parameters

vfa	128-bit vector treated as a scalar __binary128.
vfb	128-bit vector treated as a scalar __binary128.

Returns

> int boolean reflecting __binary128 less than or equal.

7.4.5.34 vec_cmpqp_all_lt()

```
static int vec_cmpqp_all_lt (
             __binary128 vfa,
             __binary128 vfb )  [inline], [static]
```

Vector Compare all Less Than (Unordered) Quad-Precision.

Compare Binary-float 128-bit values and return 1 (true), if vfa < vfb, otherwise 0 (false). Zeros of either sign are converted to +0. Infinities of different signs compare ordered. A NaN in either or both operands compare unordered.

For POWER9 (PowerISA 3.0B) or later, use a VSX Scalar Compare Unordered Quad-Precision. Otherwise comparands are converted to unsigned integer magnitudes before using vector __int128 comparison to implement the equivalent Quad-precision floating-point operation. This leverages operations from vec_int128_ppc.h.

Note

This operation *may not* follow the PowerISA relative to NaN comparison and setting the FPSCR. However if the hardware target includes an instruction that does implement the IEEE standard, the implementation may use that. This relaxed implementation may be useful for implementations on POWER8 and earlier. Especially for soft-float implementations where it is known these special cases do not occur.

processor	Latency	Throughput
power8	28-37	2/cycle
power9	3	2/cycle

Parameters

vfa	128-bit vector treated as a scalar __binary128.
vfb	128-bit vector treated as a scalar __binary128.

Returns

int boolean reflecting __binary128 less than.

7.4.5.35 vec_cmpqp_all_ne()

```
static int vec_cmpqp_all_ne (
             __binary128 vfa,
             __binary128 vfb )  [inline], [static]
```

Vector Compare all Not-Equal (Unordered) Quad-Precision.

Compare Binary-float 128-bit values and return 1, if vfa == vfb, otherwise 0. Zeros of either sign compare equal. Infinities of the same sign compare equal. A NaN in either or both operands compare unequal.

For POWER9 (PowerISA 3.0B) or later, use a VSX Scalar Compare Unordered Quad-Precision. Otherwise use vector __int128 arithmetic and logical operations to implement the equivalent Quad-precision floating-point operation. This leverages operations from vec_int128_ppc.h.

Note

This operation *may not* follow the PowerISA relative to NaN comparison and setting the FPSCR. However if the hardware target includes an instruction that does implement the IEEE standard, the implementation may use that. This relaxed implementation may be useful for implementations on POWER8 and earlier. Especially for soft-float implementations where it is known these special cases do not occur.

processor	Latency	Throughput
power8	18-30	1/cycle
power9	3	2/cycle

Parameters

vfa	128-bit vector treated as a scalar __binary128.
vfb	128-bit vector treated as a scalar __binary128.

Returns

int boolean reflecting __binary128 compare not-equal.

7.4.5.36 vec_cmpqp_all_toeq()

```
static int vec_cmpqp_all_toeq (
             __binary128 vfa,
             __binary128 vfb )  [inline], [static]
```

Vector Compare all Equal (Total-order) Quad-Precision.

Compare Binary-float 128-bit values and return 1, if vfa == vfb, otherwise 0. Zeros, Infinities and NaN of the same sign compare equal.

For POWER9 (PowerISA 3.0B) or later, use a VSX Scalar Compare Unordered Quad-Precision. Otherwise use vector __int128 arithmetic and logical operations to implement the equivalent Quad-precision floating-point operation. This leverages operations from vec_int128_ppc.h.

Note

This operation *may not* follow the IEEE standard relative to signed zero, or NaN comparison. However if the hardware target includes an instruction that does implement the IEEE standard, the implementation may use that. This relaxed implementation may be useful for implementations on POWER8 and earlier. Especially for soft-float implementations where it is known these special cases do not occur.

processor	Latency	Throughput
power8	6	2/cycle
power9	3	2/cycle

Parameters

vfa	128-bit vector treated as a scalar __binary128.
vfb	128-bit vector treated as a scalar __binary128.

Returns

int boolean reflecting __binary128 compare equal.

7.4.5.37 vec_cmpqp_all_toge()

```
static int vec_cmpqp_all_toge (
             __binary128 vfa,
             __binary128 vfb )  [inline], [static]
```

Vector Compare all Greater Than Or Equal (Total-order) Quad-Precision.

Compare Binary-float 128-bit values and return 1 (true), if vfa >= vfb, otherwise 0 (false). Zeros, Infinities and NaNs are compared as signed values. Infinities and NaNs have the highest/lowest magnitudes.

For POWER9 (PowerISA 3.0B) or later, use a VSX Scalar Compare Unordered Quad-Precision. Otherwise comparands are converted to unsigned integer magnitudes before using vector __int128 comparison to implement the equivalent Quad-precision floating-point operation. This leverages operations from vec_int128_ppc.h.

Note

> This operation *may not* follow the IEEE standard relative to signed zero, or NaN comparison. However if the hardware target includes an instruction that does implement the IEEE standard, the implementation may use that. This relaxed implementation may be useful for implementations on POWER8 and earlier. Especially for soft-float implementations where it is known these special cases do not occur.

processor	Latency	Throughput
power8	26-35	2/cycle
power9	3	2/cycle

Parameters

vfa	128-bit vector treated as a scalar __binary128.
vfb	128-bit vector treated as a scalar __binary128.

Returns

> int boolean reflecting __binary128 greater than or equal.

7.4.5.38 vec_cmpqp_all_togt()

```
static int vec_cmpqp_all_togt (
             __binary128 vfa,
             __binary128 vfb )  [inline], [static]
```

Vector Compare all Greater Than (Total-order) Quad-Precision.

Compare Binary-float 128-bit values and return 1 (true), if vfa > vfb, otherwise 0 (false). Zeros, Infinities and NaNs are compared as signed values. Infinities and NaNs have the highest/lowest magnitudes.

For POWER9 (PowerISA 3.0B) or later, use a VSX Scalar Compare Unordered Quad-Precision. Otherwise comparands are converted to unsigned integer magnitudes before using vector __int128 comparison to implement the equivalent Quad-precision floating-point operation. This leverages operations from vec_int128_ppc.h.

Note

This operation *may not* follow the IEEE standard relative to signed zero, or NaN comparison. However if the hardware target includes an instruction that does implement the IEEE standard, the implementation may use that. This relaxed implementation may be useful for implementations on POWER8 and earlier. Especially for soft-float implementations where it is known these special cases do not occur.

processor	Latency	Throughput
power8	26-35	2/cycle
power9	3	2/cycle

Parameters

vfa	128-bit vector treated as a scalar __binary128.
vfb	128-bit vector treated as a scalar __binary128.

Returns

int boolean reflecting __binary128 greater than.

7.4.5.39 vec_cmpqp_all_tole()

```
static int vec_cmpqp_all_tole (
             __binary128 vfa,
             __binary128 vfb )  [inline], [static]
```

Vector Compare All Less Than Or Equal (Total-order) Quad-Precision.

Compare Binary-float 128-bit values and return 1 (true), if vfa <= vfb, otherwise 0 (false). Zeros, Infinities and NaNs are compared as signed values. Infinities and NaNs have the highest/lowest magnitudes.

For POWER9 (PowerISA 3.0B) or later, use a VSX Scalar Compare Unordered Quad-Precision. Otherwise comparands are converted to unsigned integer magnitudes before using vector __int128 comparison to implement the equivalent Quad-precision floating-point operation. This leverages operations from vec_int128_ppc.h.

Note

This operation *may not* follow the IEEE standard relative to signed zero, or NaN comparison. However if the hardware target includes an instruction that does implement the IEEE standard, the implementation may use that. This relaxed implementation may be useful for implementations on POWER8 and earlier. Especially for soft-float implementations where it is known these special cases do not occur.

processor	Latency	Throughput
power8	26-35	2/cycle
power9	3	2/cycle

Parameters

vfa	128-bit vector treated as a scalar __binary128.
vfb	128-bit vector treated as a scalar __binary128.

Returns

int boolean reflecting __binary128 compare less than or equal.

7.4.5.40 vec_cmpqp_all_tolt()

```
static int vec_cmpqp_all_tolt (
             __binary128 vfa,
             __binary128 vfb )  [inline], [static]
```

Vector Compare All Less Than (Total-order) Quad-Precision.

Compare Binary-float 128-bit values and return 1 (true), if vfa $<$ vfb, otherwise 0 (false). Zeros, Infinities and NaNs are compared as signed values. Infinities and NaNs have the highest/lowest magnitudes.

For POWER9 (PowerISA 3.0B) or later, use a VSX Scalar Compare Unordered Quad-Precision. Otherwise comparands are converted to unsigned integer magnitudes before using vector __int128 comparison to implement the equivalent Quad-precision floating-point operation. This leverages operations from vec_int128_ppc.h.

Note

This operation *may not* follow the IEEE standard relative to signed zero, or NaN comparison. However if the hardware target includes an instruction that does implement the IEEE standard, the implementation may use that. This relaxed implementation may be useful for implementations on POWER8 and earlier. Especially for soft-float implementations where it is known these special cases do not occur.

processor	Latency	Throughput
power8	26-35	2/cycle
power9	3	2/cycle

Parameters

vfa	128-bit vector treated as a scalar __binary128.
vfb	128-bit vector treated as a scalar __binary128.

Returns

int boolean reflecting __binary128 compare less than.

7.4.5.41 vec_cmpqp_all_tone()

```
static int vec_cmpqp_all_tone (
             __binary128 vfa,
             __binary128 vfb )  [inline], [static]
```

Vector Compare all Not-Equal (Total-order) Quad-Precision.

Compare Binary-float 128-bit values and return 1, if vfa != vfb, otherwise 0. Zeros, Infinities and NaN of the same sign compare equal.

For POWER9 (PowerISA 3.0B) or later, use a VSX Scalar Compare Unordered Quad-Precision. Otherwise use vector __int128 arithmetic and logical operations to implement the equivalent Quad-precision floating-point operation. This leverages operations from vec_int128_ppc.h.

Note

> This operation *may not* follow the IEEE standard relative to signed zero, or NaN comparison. However if the hardware target includes an instruction that does implement the IEEE standard, the implementation may use that. This relaxed implementation may be useful for implementations on POWER8 and earlier. Especially for soft-float implementations where it is known these special cases do not occur.

processor	Latency	Throughput
power8	6	2/cycle
power9	3	2/cycle

Parameters

vfa	128-bit vector treated as a scalar __binary128.
vfb	128-bit vector treated as a scalar __binary128.

Returns

> int boolean reflecting __binary128 compare not-equal.

7.4.5.42 vec_cmpqp_all_uzeq()

```
static int vec_cmpqp_all_uzeq (
             __binary128 vfa,
             __binary128 vfb )  [inline], [static]
```

Vector Compare all Equal (Zero-unordered) Quad-Precision.

Compare Binary-float 128-bit values and return 1, if vfa == vfb, otherwise 0. Zeros of either sign compare equal. Infinities and NaNs of the same sign compare equal.

For POWER9 (PowerISA 3.0B) or later, use a VSX Scalar Compare Unordered Quad-Precision. Otherwise use vector __int128 arithmetic and logical operations to implement the equivalent Quad-precision floating-point operation. This leverages operations from vec_int128_ppc.h.

Note

This operation *may not* follow the IEEE standard relative NaN comparison. However if the hardware target includes an instruction that does implement the IEEE standard, the implementation may use that. This relaxed implementation may be useful for implementations on POWER8 and earlier. Especially for soft-float implementations where it is known these special cases do not occur.

processor	Latency	Throughput
power8	10	1/cycle
power9	3	2/cycle

Parameters

vfa	128-bit vector treated as a scalar __binary128.
vfb	128-bit vector treated as a scalar __binary128.

Returns

int boolean reflecting __binary128 compare equal.

7.4.5.43 vec_cmpqp_all_uzge()

```
static int vec_cmpqp_all_uzge (
             __binary128 vfa,
             __binary128 vfb )  [inline], [static]
```

Vector Compare all Greater Than Or Equal (Zero-unordered) Quad-Precision.

Compare Binary-float 128-bit values and return 1 (true), if vfa >= vfb, otherwise 0 (false). Zeros of either sign are converted to +0. Infinities and NaNs are compared as signed values. Infinities and NaNs have the highest/lowest magnitudes.

For POWER9 (PowerISA 3.0B) or later, use a VSX Scalar Compare Unordered Quad-Precision. Otherwise comparands are converted to unsigned integer magnitudes before using vector __int128 comparison to implement the equivalent Quad-precision floating-point operation. This leverages operations from vec_int128_ppc.h.

Note

This operation *may not* follow the IEEE standard relative to NaN comparison. However if the hardware target includes an instruction that does implement the IEEE standard, the implementation may use that. This relaxed implementation may be useful for implementations on POWER8 and earlier. Especially for soft-float implementations where it is known these special cases do not occur.

processor	Latency	Throughput
power8	28-37	2/cycle
power9	3	2/cycle

Parameters

vfa	128-bit vector treated as a scalar __binary128.
vfb	128-bit vector treated as a scalar __binary128.

Returns

int boolean reflecting __binary128 greater than or equal.

7.4.5.44 vec_cmpqp_all_uzgt()

```
static int vec_cmpqp_all_uzgt (
             __binary128 vfa,
             __binary128 vfb )  [inline], [static]
```

Vector Compare all Greater Than (Zero-unordered) Quad-Precision.

Compare Binary-float 128-bit values and return 1 (true), if vfa > vfb, otherwise 0 (false). Zeros of either sign are converted to +0. Infinities and NaNs are compared as signed values. Infinities and NaNs have the highest/lowest magnitudes.

For POWER9 (PowerISA 3.0B) or later, use a VSX Scalar Compare Unordered Quad-Precision. Otherwise comparands are converted to unsigned integer magnitudes before using vector __int128 comparison to implement the equivalent Quad-precision floating-point operation. This leverages operations from vec_int128_ppc.h.

Note

This operation *may not* follow the IEEE standard relative to NaN comparison. However if the hardware target includes an instruction that does implement the IEEE standard, the implementation may use that. This relaxed implementation may be useful for implementations on POWER8 and earlier. Especially for soft-float implementations where it is known these special cases do not occur.

processor	Latency	Throughput
power8	28-37	2/cycle
power9	3	2/cycle

Parameters

vfa	128-bit vector treated as a scalar __binary128.
vfb	128-bit vector treated as a scalar __binary128.

Returns

int boolean reflecting __binary128 greater than.

7.4.5.45 vec_cmpqp_all_uzle()

```
static int vec_cmpqp_all_uzle (
             __binary128 vfa,
             __binary128 vfb )  [inline], [static]
```

Vector Compare all Less Than Or Equal (Zero-unordered) Quad-Precision.

Compare Binary-float 128-bit values and return 1 (true), if vfa <= vfb, otherwise 0 (false). Zeros of either sign are converted to +0. Infinities and NaNs are compared as signed values. Infinities and NaNs have the highest/lowest magnitudes.

For POWER9 (PowerISA 3.0B) or later, use a VSX Scalar Compare Unordered Quad-Precision. Otherwise comparands are converted to unsigned integer magnitudes before using vector __int128 comparison to implement the equivalent Quad-precision floating-point operation. This leverages operations from vec_int128_ppc.h.

Note

This operation *may not* follow the IEEE standard relative to NaN comparison. However if the hardware target includes an instruction that does implement the IEEE standard, the implementation may use that. This relaxed implementation may be useful for implementations on POWER8 and earlier. Especially for soft-float implementations where it is known these special cases do not occur.

processor	Latency	Throughput
power8	28-37	2/cycle
power9	3	2/cycle

Parameters

vfa	128-bit vector treated as a scalar __binary128.
vfb	128-bit vector treated as a scalar __binary128.

Returns

int boolean reflecting __binary128 less than or equal.

7.4.5.46 vec_cmpqp_all_uzlt()

```
static int vec_cmpqp_all_uzlt (
             __binary128 vfa,
             __binary128 vfb )  [inline], [static]
```

Vector Compare all Less Than (Zero-unordered) Quad-Precision.

Compare Binary-float 128-bit values and return 1 (true), if vfa $<$ vfb, otherwise 0 (false). Zeros of either sign are converted to +0. Infinities and NaNs are compared as signed values. Infinities and NaNs have the highest/lowest magnitudes.

For POWER9 (PowerISA 3.0B) or later, use a VSX Scalar Compare Unordered Quad-Precision. Otherwise comparands are converted to unsigned integer magnitudes before using vector __int128 comparison to implement the equivalent Quad-precision floating-point operation. This leverages operations from vec_int128_ppc.h.

Note

This operation *may not* follow the IEEE standard relative to NaN comparison. However if the hardware target includes an instruction that does implement the IEEE standard, the implementation may use that. This relaxed implementation may be useful for implementations on POWER8 and earlier. Especially for soft-float implementations where it is known these special cases do not occur.

processor	Latency	Throughput
power8	28-37	2/cycle
power9	3	2/cycle

Parameters

vfa	128-bit vector treated as a scalar __binary128.
vfb	128-bit vector treated as a scalar __binary128.

Returns

int boolean reflecting __binary128 less than.

7.4.5.47 vec_cmpqp_all_uzne()

```
static int vec_cmpqp_all_uzne (
            __binary128 vfa,
            __binary128 vfb ) [inline], [static]
```

Vector Compare all Not-Equal (Zero-unordered) Quad-Precision.

Compare Binary-float 128-bit values and return 1, if vfa != vfb, otherwise 0. Zeros of either sign compare equal. Infinities and NaNs of the same sign compare equal.

For POWER9 (PowerISA 3.0B) or later, use a VSX Scalar Compare Unordered Quad-Precision. Otherwise use vector __int128 arithmetic and logical operations to implement the equivalent Quad-precision floating-point operation. This leverages operations from vec_int128_ppc.h.

Note

This operation *may not* follow the IEEE standard relative NaN comparison. However if the hardware target includes an instruction that does implement the IEEE standard, the implementation may use that. This relaxed implementation may be useful for implementations on POWER8 and earlier. Especially for soft-float implementations where it is known these special cases do not occur.

processor	Latency	Throughput
power8	10	1/cycle
power9	3	2/cycle

Parameters

vfa	128-bit vector treated as a scalar __binary128.
vfb	128-bit vector treated as a scalar __binary128.

Returns

int boolean reflecting __binary128 compare equal.

7.4.5.48 vec_cmpqp_exp_eq()

```
static int vec_cmpqp_exp_eq (
             __binary128 vfa,
             __binary128 vfb )  [inline], [static]
```

Vector Compare Quad-Precision Exponents for Equal.

Compare the exponents of two Binary-float 128-bit values and return 1, if vfa^{exp} == vfb^{exp}, otherwise 0. A NaN in either or both operands compare unequal.

For POWER9 (PowerISA 3.0B) or later, use the VSX Scalar Compare Exponents Quad-Precision instruction. Otherwise use vector __int128 arithmetic and logical operations to implement the equivalent Quad-precision floating-point operation. This leverages operations from vec_int128_ppc.h.

Note

This operation *may not* follow the PowerISA relative to setting the FPSCR. However if the hardware target includes an instruction that does implement the IEEE standard, the implementation may use that. This relaxed implementation may be useful for implementations on POWER8 and earlier. Especially for soft-float implementations where it is known these special cases do not occur.

processor	Latency	Throughput
power8	8-17	1/cycle
power9	3	2/cycle

Parameters

vfa	128-bit vector treated as a scalar __binary128.
vfb	128-bit vector treated as a scalar __binary128.

Returns

int boolean reflecting __binary128 exponent compare equal.

7.4.5.49 vec_cmpqp_exp_gt()

```
static int vec_cmpqp_exp_gt (
             __binary128 vfa,
             __binary128 vfb )  [inline], [static]
```

Vector Compare Exponents Quad-Precision for Greater Than.

Compare the exponents of two Binary-float 128-bit values and return 1, if $vfa^{exp} > vfb^{exp}$, otherwise 0. A NaN in either or both operands returns 0.

For POWER9 (PowerISA 3.0B) or later, use the VSX Scalar Compare Exponents Quad-Precision instruction. Otherwise use vector __int128 arithmetic and logical operations to implement the equivalent Quad-precision floating-point operation. This leverages operations from vec_int128_ppc.h.

Note

This operation *may not* follow the PowerISA relative to setting the FPSCR. However if the hardware target includes an instruction that does implement the IEEE standard, the implementation may use that. This relaxed implementation may be useful for implementations on POWER8 and earlier. Especially for soft-float implementations where it is known these special cases do not occur.

processor	Latency	Throughput
power8	8-17	1/cycle
power9	3	2/cycle

Parameters

vfa	128-bit vector treated as a scalar __binary128.
vfb	128-bit vector treated as a scalar __binary128.

Returns

int boolean reflecting __binary128 exponent compare greater than.

7.4.5.50 vec_cmpqp_exp_lt()

```
static int vec_cmpqp_exp_lt (
             __binary128 vfa,
             __binary128 vfb )  [inline], [static]
```

Vector Compare Exponents Quad-Precision for Less Than.

Compare the exponents of two Binary-float 128-bit values and return 1, if vfa^{exp} $<$ vfb^{exp}, otherwise 0. A NaN in either or both operands returns 0.

For POWER9 (PowerISA 3.0B) or later, use the VSX Scalar Compare Exponents Quad-Precision instruction. Otherwise use vector __int128 arithmetic and logical operations to implement the equivalent Quad-precision floating-point operation. This leverages operations from vec_int128_ppc.h.

Note

> This operation *may not* follow the PowerISA relative to setting the FPSCR. However if the hardware target includes an instruction that does implement the IEEE standard, the implementation may use that. This relaxed implementation may be useful for implementations on POWER8 and earlier. Especially for soft-float implementations where it is known these special cases do not occur.

processor	Latency	Throughput
power8	8-17	1/cycle
power9	3	2/cycle

Parameters

vfa	128-bit vector treated as a scalar __binary128.
vfb	128-bit vector treated as a scalar __binary128.

Returns

> int boolean reflecting __binary128 exponent compare equal.

7.4.5.51 vec_cmpqp_exp_unordered()

```
static int vec_cmpqp_exp_unordered (
             __binary128 vfa,
             __binary128 vfb )  [inline], [static]
```

Vector Compare Exponents Quad-Precision for Unordered.

Compare two Binary-float 128-bit values and return 1, if either or both operands are NaN, otherwise 0.

For POWER9 (PowerISA 3.0B) or later, use the VSX Scalar Compare Exponents Quad-Precision instruction. Otherwise use vector __int128 arithmetic and logical operations to implement the equivalent Quad-precision floating-point operation. This leverages operations from vec_int128_ppc.h.

Note

> This operation *may not* follow the PowerISA relative to setting the FPSCR. However if the hardware target includes an instruction that does implement the IEEE standard, the implementation may use that. This relaxed implementation may be useful for implementations on POWER8 and earlier. Especially for soft-float implementations where it is known these special cases do not occur.

processor	Latency	Throughput
power8	8-17	1/cycle
power9	3	2/cycle

Parameters

vfa	128-bit vector treated as a scalar __binary128.
vfb	128-bit vector treated as a scalar __binary128.

Returns

int boolean reflecting __binary128 unordered.

7.4.5.52 vec_const128_f128_128()

```
static vui32_t vec_const128_f128_128 (
             void  )  [inline], [static]
```

Generate Quadword constant 128.

Load immediate the quadword constant vui32_t {0, 0, 0, 128}.

Note

See vec_mask64_f128exp() for rationale.

processor	Latency	Throughput
power8	4-6	1/cycle

Returns

The 128-bit const vui32_t {0, 0, 0, 128}.

7.4.5.53 vec_const64_f128_128()

```
static vui64_t vec_const64_f128_128 (
             void  )  [inline], [static]
```

Generate doubleword splat constant 128.

Load immediate the quadword constant vui32_t {0, 0, 0, 128}.

Note

See vec_mask64_f128exp() for rationale.

processor	Latency	Throughput
power8	4-6	1/cycle

Returns

The 128-bit const vui64_t {128, 128}.

7.4.5.54 vec_const_huge_valf128()

```
static __binary128 vec_const_huge_valf128 ( )  [inline], [static]
```

return a positive infinity.

Returns

const __float128 positive infinity.

7.4.5.55 vec_const_inff128()

```
static __binary128 vec_const_inff128 ( )  [inline], [static]
```

return a positive infinity.

Returns

a const __float128 positive infinity.

7.4.5.56 vec_const_nanf128()

```
static __binary128 vec_const_nanf128 ( )  [inline], [static]
```

return a quiet NaN.

Returns

a const __float128 quiet NaN.

7.4.5.57 vec_const_nansf128()

```
static __binary128 vec_const_nansf128 ( )  [inline], [static]
```

return a signaling NaN.

Returns

a const __float128 signaling NaN.

7.4.5.58 vec_copysignf128()

```
static __binary128 vec_copysignf128 (
              __binary128 f128x,
              __binary128 f128y )  [inline], [static]
```

Copy the sign bit from f128x and merge with the magnitude from f128y. The merged result is returned as a __float128 value.

Note

This operation was patterned after the intrinsic vec_cpsgn (altivec.h) introduced for POWER7 and VSX. It turns out the original (GCC 4.9) compiler implementation reversed the operands and does not match the PowerISA or the Vector Intrinsic Programming Reference manuals. Subsequent compilers and PVECLIB implementations replicated this (operand order) error. This has now been reported as bug against the compilers, which are in the process of applying fixes and distributing updates. This version of PVECLIB is updated to match the Vector Intrinsic Programming Reference.

processor	Latency	Throughput
power8	2-11	2/cycle
power9	2	4/cycle

Parameters

f128x	a __float128 value containing the sign bit.
f128y	a __float128 value containing the magnitude.

Returns

a __float128 value with magnitude from f128y and the sign of f128x.

7.4.5.59 vec_isfinitef128()

```
static vb128_t vec_isfinitef128 (
            __binary128 f128 )  [inline], [static]
```

Return 128-bit vector boolean true if the __float128 value is Finite (Not NaN nor Inf).

A IEEE Binary128 finite value has an exponent between 0x0000 and 0x7ffe (a 0x7fff indicates NaN or Inf). The significand can be any value. Using the vec_cmpeq conditional to generate the predicate mask for NaN / Inf and then invert this for the finite condition. The sign bit is ignored.

processor	Latency	Throughput
power8	8-17	2/cycle
power9	6	2/cycle

Note

This function will not raise VXSNAN or VXVC (FE_INVALID) exceptions. A normal __float128 compare can.

Parameters

f128	a __float128 value in vector.

Returns

a vector boolean containing all 0s or 1s.

7.4.5.60 vec_isinf_signf128()

```
static int vec_isinf_signf128 (
            __binary128 f128 )  [inline], [static]
```

Return true (nonzero) value if the __float128 value is infinity. If infinity, indicate the sign as +1 for positive infinity and -1 for negative infinity.

A IEEE Binary128 infinity has a exponent of 0x7fff and significand of all zeros. Using the vec_all_eq compare conditional verifies both conditions. A subsequent vec_any_gt checks the sign bit and set the result appropriately. The sign bit is ignored.

This sequence avoids a vector -> GPR transfer for platforms before PowerISA-2.07.

processor	Latency	Throughput
power8	12-32	1/cycle
power9	3-12	2/cycle

Note

This function will not raise VXSNAN or VXVC (FE_INVALID) exceptions. A normal __float128 compare can.

Parameters

f128	a __float128 value in vector.

Returns

an int containing 0 if not infinity and +1/-1 otherwise.

7.4.5.61 vec_isinff128()

```
static vb128_t vec_isinff128 (
            __binary128 f128 )  [inline], [static]
```

Return a 128-bit vector boolean true if the __float128 value is infinity.

A IEEE Binary128 infinity has a exponent of 0x7fff and significand of all zeros. The sign bit is ignored.

processor	Latency	Throughput
power8	8-17	2/cycle
power9	6	2/cycle

Note

This function will not raise VXSNAN or VXVC (FE_INVALID) exceptions. A normal __float128 compare can.

Parameters

f128	a __float128 value in vector.

Returns

a vector boolean containing all 0s or 1s..

7.4.5.62 vec_isnanf128()

```
static vb128_t vec_isnanf128 (
            __binary128 f128 )  [inline], [static]
```

Return 128-bit vector boolean true if the __float128 value is Not a Number (NaN).

A IEEE Binary128 NaN has a exponent of 0x7fff and nonzero significand. The sign bit is ignored. For POWER9 and later we use scalar_test_data_class(). Otherwise mask off the sign bit and compare greater than unsigned quadword to the integer equivalent of Quad-Precision infinity.

processor	Latency	Throughput
power8	14-23	1/cycle
power9	6	2/cycle

Note

This function will not raise VXSNAN or VXVC (FE_INVALID) exceptions. A normal __float128 compare can.

Parameters

f128	a __float128 value in vector.

Returns

a vector boolean containing all 0s or 1s.

7.4.5.63 vec_isnormalf128()

```
static vb128_t vec_isnormalf128 (
            __binary128 f128 )  [inline], [static]
```

Return 128-bit vector boolean true if the __float128 value is normal (Not NaN, Inf, denormal, or zero).

A IEEE Binary128 normal value has an exponent between 0x0001 and 0x7ffe (a 0x7fff indicates NaN or Inf). The significand can be any value (expect 0 if the exponent is zero). The sign bit is ignored.

processor	Latency	Throughput
power8	10-19	2/cycle
power9	6	2/cycle

Note

This function will not raise VXSNAN or VXVC (FE_INVALID) exceptions. A normal __float128 compare can.

Parameters

f128	a __float128 value in vector.

Returns

a vector boolean containing all 0s or 1s.

7.4.5.64 vec_issubnormalf128()

```
static vb128_t vec_issubnormalf128 (
            __binary128 f128 )  [inline], [static]
```

Return 128-bit vector boolean true value, if the __float128 value is subnormal (denormal).

A IEEE Binary128 subnormal has an exponent of 0x0000 and a nonzero significand. The sign bit is ignored.

Note

This function will not raise VXSNAN or VXVC (FE_INVALID) exceptions. A normal __float128 compare can.

processor	Latency	Throughput
power8	16-25	1/cycle
power9	6	1/cycle

Parameters

f128	a vector of __binary64 values.

Returns

a vector boolean long long, each containing all 0s(false) or 1s(true).

7.4.5.65 vec_isunorderedf128()

```
static vb128_t vec_isunorderedf128 (
            __binary128 vfa,
            __binary128 vfb )  [inline], [static]
```

Return 128-bit vector boolean true value, if either __float128 value (vra, vrb) is NaN.

A IEEE Binary128 NaN has a exponent of 0x7fff and nonzero significand. The sign bit is ignored. For POWER9 and later we use scalar_test_data_class(). Otherwise mask off the sign bit and compare greater than unsigned quadword to the integer equivalent of Quad-Precision infinity.

Note

This function will not raise VXSNAN or VXVC (FE_INVALID) exceptions. A normal __float128 compare can.

processor	Latency	Throughput
power8	16-25	1/cycle
power9	6	1/cycle

Parameters

vfa	128-bit vector treated as a scalar __binary128.
vfb	128-bit vector treated as a scalar __binary128.

Returns

a vector boolean _int128.

7.4.5.66 vec_iszerof128()

```
static vb128_t vec_iszerof128 (
             __binary128 f128 )  [inline], [static]
```

Return 128-bit vector boolean true value, if the value that is +-0.0.

A IEEE Binary64 zero has an exponent of 0x000 and a zero significand. The sign bit is ignored.

Note

This function will not raise VXSNAN or VXVC (FE_INVALID) exceptions. A normal __float128 compare can.

processor	Latency	Throughput
power8	8-17	2/cycle
power9	6	2/cycle

Parameters

f128	a vector of __binary32 values.

Returns

a vector boolean int, each containing all 0s(false) or 1s(true).

7.4.5.67 vec_mask128_f128Cbit()

```
static vui32_t vec_mask128_f128Cbit (
              void ) [inline], [static]
```

Generate Quadword C-bit mask Immediate.

Load immediate the quadword constant vui32_t {0x00020000, 0, 0, 0}.

Note

See vec_mask64_f128exp() for rationale.

processor	Latency	Throughput
power8	2-4	1/cycle

Returns

The 128-bit mask vui32_t {0x00020000, 0, 0, 0}.

7.4.5.68 vec_mask128_f128exp()

```
static vui32_t vec_mask128_f128exp (
              void ) [inline], [static]
```

Generate Quadword Quad-Precision exponent mask.

Load immediate the quadword constant vui32_t {0x7fff0000, 0, 0, 0}.

Note

See vec_mask64_f128exp() for rationale.

processor	Latency	Throughput
power8	6	1/cycle

Returns

The 128-bit mask vui32_t {0x7fff0000, 0, 0, 0}.

7.4.5.69 vec_mask128_f128Lbit()

```
static vui32_t vec_mask128_f128Lbit (
             void  )  [inline], [static]
```

Generate Quadword L-bit mask Immediate.

Load immediate the quadword constant vui32_t {0x00010000, 0, 0, 0}.

Note

See vec_mask64_f128exp() for rationale.

processor	Latency	Throughput
power8	2-4	1/cycle

Returns

The 128-bit mask vui32_t {0x00010000, 0, 0, 0}.

7.4.5.70 vec_mask128_f128mag()

```
static vui32_t vec_mask128_f128mag (
             void  )  [inline], [static]
```

Generate Quadword Quad-Precision magnitude mask.

Load immediate the quadword constant vui32_t {0x7fffffff, -1, -1, -1}.

Note

See vec_mask64_f128exp() for rationale.

processor	Latency	Throughput
power8	2-4	1/cycle

Returns

The 128-bit mask vui32_t {0x7fffffff, -1, -1, -1}.

7.4.5.71 vec_mask128_f128Qbit()

```
static vui32_t vec_mask128_f128Qbit (
             void  )  [inline], [static]
```

Generate Quadword QNaN-bit mask Immediate.

Load immediate the quadword constant vui32_t {0x00008000, 0, 0, 0}.

Note

See vec_mask64_f128exp() for rationale.

processor	Latency	Throughput
power8	4-6	1/cycle

Returns

The 128-bit mask vui32_t {0x00008000, 0, 0, 0}.

7.4.5.72 vec_mask128_f128sig()

```
static vui32_t vec_mask128_f128sig (
             void  )  [inline], [static]
```

Generate Quadword Quad-Precision significand mask.

Load immediate the quadword constant vui32_t {0x0000ffff, -1, -1, -1}.

Note

See vec_mask64_f128exp() for rationale.

processor	Latency	Throughput
power8	2-4	1/cycle

Returns

The 128-bit mask vui32_t {0x0000ffff, -1, -1, -1}.

7.4.5.73 vec_mask128_f128sign()

```
static vui32_t vec_mask128_f128sign (
            void ) [inline], [static]
```

Generate Quadword Quad-Precision Sign-bit mask.

Load immediate the quadword constant vui32_t {0x80000000, -1, -1, -1}.

Note

See vec_mask64_f128exp() for rationale.

processor	Latency	Throughput
power8	4-6	1/cycle

Returns

The 128-bit mask vui32_t {0x80000000, -1, -1, -1}.

7.4.5.74 vec_mask64_f128exp()

```
static vui64_t vec_mask64_f128exp (
            void ) [inline], [static]
```

Generate Doubleword Quad-Precision exponent mask.

Load the quadword constant vui32_t {0, 0x7fff, 0, 0x7fff}.

Note

The compiler, left to its own devices, will load vector constants from the read-only data section (.rodata). This requires (at least) 5 cycles assuming a L1 cache hit. This does not include any instructions needed for address calculation, alignment, or endian fix-up (another 4-8 cycles). A L1 cache miss adds more (7-27) cycles to retrieve data from the L2/L3 cache levels. A short sequence, using only vector register and immediate operands, eliminates load associated address calculation and cache misses. This provides better and more repeatable performance.

processor	Latency	Throughput
power8	4-6	1/cycle

Note

The consistent use of q_zero/q_ones constants across these operations allows the compiler to apply common subexpression elimination (CSE) optimizations.

Returns

The 128-bit mask vui32_t {0x00010000, 0, 0, 0}.

7.4.5.75 vec_mrgh_bin128_2_vui64t()

```
static vui64_t vec_mrgh_bin128_2_vui64t (
            __binary128 vfa,
            __binary128 vfb ) [inline], [static]
```

Merge High and Transfer function from a pair of __binary128 scalars to a vector long long int.

The compiler does not allow direct transfer (assignment or type cast) between __binary128 (__float128) scalars and vector types. This despite the fact the the ABI and ISA require __binary128 in vector registers (VRs).

Note

this function uses a union to effect the (logical) transfer. The compiler should generate a single xxmrghd for this.

Parameters

vfa	a __binary128 floating point scalar value.
vfb	a __binary128 floating point scalar value.

Returns

The high doublewords of vfa/vfb value merged into a vector long long int.

7.4.5.76 vec_mrgl_bin128_2_vui64t()

```
static vui64_t vec_mrgl_bin128_2_vui64t (
            __binary128 vfa,
            __binary128 vfb ) [inline], [static]
```

Merge Low and Transfer function from a pair of __binary128 scalars to a vector long long int.

The compiler does not allow direct transfer (assignment or type cast) between __binary128 (__float128) scalars and vector types. This despite the fact the the ABI and ISA require __binary128 in vector registers (VRs).

Note

this function uses a union to effect the (logical) transfer. The compiler should generate a single xxmrgld for this.

Parameters

vfa	a __binary128 floating point scalar value.
vfb	a __binary128 floating point scalar value.

Returns

The low doublewords of vfa/vfb value merged into a vector long long int.

7.4.5.77 vec_nabsf128()

```
static __binary128 vec_nabsf128 (
             __binary128 f128 )  [inline], [static]
```

Negative Absolute value Quad-Precision.

Unconditionally set sign bit of the __float128 input and return the resulting positive __float128 value.

processor	Latency	Throughput
power8	2-11	2/cycle
power9	2	4/cycle

Parameters

f128	a __float128 value containing a signed value.

Returns

a __float128 value with magnitude from f128 and a negative sign.

7.4.5.78 vec_negf128()

```
static __binary128 vec_negf128 (
             __binary128 f128 )  [inline], [static]
```

Negate the sign bit of a __float128 input and return the resulting __float128 value.

processor	Latency	Throughput
power8	2-11	2/cycle
power9	2	4/cycle

Parameters

f128	a __float128 value containing a signed value.

Returns

a __float128 value with magnitude from f128 and the opposite sign of f128.

7.4.5.79 vec_or_bin128_2_vui32t()

```
static vui32_t vec_or_bin128_2_vui32t (
             __binary128 f128,
             vui32_t mask )  [inline], [static]
```

Transfer a quadword from a __binary128 scalar to a vector int and logical OR with mask.

The compiler does not allow direct transfer (assignment or type cast) between __binary128 (__float128) scalars and vector types. This despite the fact the the ABI and ISA require __binary128 in vector registers (VRs).

Note

this function uses a union to effect the (logical) transfer. The compiler should not generate any code for this.

Parameters

f128	a __binary128 floating point scalar value.
mask	a vector unsigned int

Returns

The original value ORed with mask as a 128-bit vector int.

7.4.5.80 vec_sel_bin128_2_bin128()

```
static __binary128 vec_sel_bin128_2_bin128 (
             __binary128 vfa,
             __binary128 vfb,
             vb128_t mask )  [inline], [static]
```

Select and Transfer from one of two __binary128 scalars under a 128-bit mask. The result is a __binary128 of the selected value.

The compiler does not allow direct transfer (assignment or type cast) between __binary128 (__float128) scalars and vector types. This despite the fact the the ABI and ISA require __binary128 in vector registers (VRs).

Note

this function uses a union to effect the (logical) transfer. The compiler should not generate any code for this.

Parameters

vfa	a __binary128 floating point scalar value.
vfb	a __binary128 floating point scalar value.
mask	a vector bool __int128

Returns

The bit of vfa or vfb depending on the mask.

7.4.5.81 vec_self128()

```
static __binary128 vec_self128 (
            __binary128 vfa,
            __binary128 vfb,
            vb128_t mask )  [inline], [static]
```

Select and Transfer from one of two __binary128 scalars under a 128-bit mask. The result is a __binary128 of the selected value.

processor	Latency	Throughput
power8	2	2/cycle
power9	2	4/cycle

Parameters

vfa	a __binary128 floating point scalar value.
vfb	a __binary128 floating point scalar value.
mask	a vector bool __int128

Returns

The bit of vfa or vfb depending on the mask.

7.4.5.82 vec_setb_qp()

```
static vb128_t vec_setb_qp (
            __binary128 f128 )  [inline], [static]
```

Vector Set Bool from Quadword Floating-point.

If the quadword's sign bit is '1' then return a vector bool __int128 that is all '1's. Otherwise return all '0's.

The resulting mask can be used in vector masking and select operations.

Note

> This operation will set the sign mask regardless of data class. For POWER9 the Scalar Test Data Class instructions copy the sign bit to CR bit 0 which distinguishes between +/- NaN.

processor	Latency	Throughput
power8	4 - 6	2/cycle
power9	6	2/cycle

Parameters

f128	a 128-bit vector treated a signed __int128.

Returns

> a 128-bit vector bool of all '1's if the sign bit is '1'. Otherwise all '0's.

7.4.5.83 vec_signbitf128()

```
static int vec_signbitf128 (
            __binary128 f128 )  [inline], [static]
```

Return int boolean true if the __float128 value is negative (sign bit is '1').

For POWER9 use scalar_test_neg (a special case of scalar_test_data_class). For POWER8 and earlier, vec_and with a signmask and then vec_all_eq compare with that mask generates the boolean of the sign bit.

processor	Latency	Throughput
power8	4-10	2/cycle
power9	3	2/cycle

Parameters

f128	a __float128 value in vector.

Returns

a int boolean indicating the sign bit.

7.4.5.84 vec_xfer_bin128_2_vui128t()

```
static vui128_t vec_xfer_bin128_2_vui128t (
             __binary128 f128 )  [inline], [static]
```

Transfer function from a __binary128 scalar to a vector __int128.

The compiler does not allow direct transfer (assignment or type cast) between __binary128 (__float128) scalars and vector types. This despite the fact the the ABI and ISA require __binary128 in vector registers (VRs).

Note

this function uses a union to effect the (logical) transfer. The compiler should not generate any code for this.

Parameters

f128	a __binary128 floating point scalar value.

Returns

The original value as a 128-bit vector __int128.

7.4.5.85 vec_xfer_bin128_2_vui16t()

```
static vui16_t vec_xfer_bin128_2_vui16t (
             __binary128 f128 )  [inline], [static]
```

Transfer function from a __binary128 scalar to a vector short int.

The compiler does not allow direct transfer (assignment or type cast) between __binary128 (__float128) scalars and vector types. This despite the fact the the ABI and ISA require __binary128 in vector registers (VRs).

Note

this function uses a union to effect the (logical) transfer. The compiler should not generate any code for this.

Parameters

f128	a __binary128 floating point scalar value.

Returns

The original value as a 128-bit vector short int.

7.4.5.86 vec_xfer_bin128_2_vui32t()

```
static vui32_t vec_xfer_bin128_2_vui32t (
            __binary128 f128 )  [inline], [static]
```

Transfer function from a __binary128 scalar to a vector int.

The compiler does not allow direct transfer (assignment or type cast) between __binary128 (__float128) scalars and vector types. This despite the fact the the ABI and ISA require __binary128 in vector registers (VRs).

Note

this function uses a union to effect the (logical) transfer. The compiler should not generate any code for this.

Parameters

f128	a __binary128 floating point scalar value.

Returns

The original value as a 128-bit vector int.

7.4.5.87 vec_xfer_bin128_2_vui64t()

```
static vui64_t vec_xfer_bin128_2_vui64t (
            __binary128 f128 )  [inline], [static]
```

Transfer function from a __binary128 scalar to a vector long long int.

The compiler does not allow direct transfer (assignment or type cast) between __binary128 (__float128) scalars and vector types. This despite the fact the the ABI and ISA require __binary128 in vector registers (VRs).

Note

this function uses a union to effect the (logical) transfer. The compiler should not generate any code for this.

Parameters

f128	a __binary128 floating point scalar value.

Returns

The original value as a 128-bit vector long long int.

7.4.5.88 vec_xfer_bin128_2_vui8t()

```
static vui8_t vec_xfer_bin128_2_vui8t (
             __binary128 f128 )  [inline], [static]
```

Transfer function from a __binary128 scalar to a vector char.

The compiler does not allow direct transfer (assignment or type cast) between __binary128 (__float128) scalars and vector types. This despite the fact the the ABI and ISA require __binary128 in vector registers (VRs).

Note

this function uses a union to effect the (logical) transfer. The compiler should not generate any code for this.

Parameters

f128	a __binary128 floating point scalar value.

Returns

The original value as a 128-bit vector char.

7.4.5.89 vec_xfer_vui128t_2_bin128()

```
static __binary128 vec_xfer_vui128t_2_bin128 (
             vui128_t f128 )  [inline], [static]
```

Transfer a vector unsigned __int128 to __binary128 scalar.

The compiler does not allow direct transfer (assignment or type cast) between __binary128 (__float128) scalars and vector types. This despite the fact the the ABI and ISA require __binary128 in vector registers (VRs).

Note

this function uses a union to effect the (logical) transfer. The compiler should not generate any code for this.

Parameters

f128	a vector unsigned __int128 value.

Returns

The original value returned as a __binary128 scalar.

7.4.5.90 vec_xfer_vui16t_2_bin128()

```
static __binary128 vec_xfer_vui16t_2_bin128 (
             vui16_t f128 )  [inline], [static]
```

Transfer a vector unsigned short to __binary128 scalar.

The compiler does not allow direct transfer (assignment or type cast) between __binary128 (__float128) scalars and vector types. This despite the fact the the ABI and ISA require __binary128 in vector registers (VRs).

Note

this function uses a union to effect the (logical) transfer. The compiler should not generate any code for this.

Parameters

f128	a vector unsigned short value.

Returns

The original value returned as a __binary128 scalar.

7.4.5.91 vec_xfer_vui32t_2_bin128()

```
static __binary128 vec_xfer_vui32t_2_bin128 (
             vui32_t f128 )  [inline], [static]
```

Transfer a vector unsigned int to __binary128 scalar.

The compiler does not allow direct transfer (assignment or type cast) between __binary128 (__float128) scalars and vector types. This despite the fact the the ABI and ISA require __binary128 in vector registers (VRs).

Note

this function uses a union to effect the (logical) transfer. The compiler should not generate any code for this.

Parameters

f128	a vector unsigned int value.

Returns

The original value returned as a __binary128 scalar.

7.4.5.92 vec_xfer_vui64t_2_bin128()

```
static __binary128 vec_xfer_vui64t_2_bin128 (
             vui64_t f128 )  [inline], [static]
```

Transfer a vector unsigned long long to __binary128 scalar.

The compiler does not allow direct transfer (assignment or type cast) between __binary128 (__float128) scalars and vector types. This despite the fact the the ABI and ISA require __binary128 in vector registers (VRs).

Note

this function uses a union to effect the (logical) transfer. The compiler should not generate any code for this.

Parameters

f128	a vector unsigned long long value.

Returns

The original value returned as a __binary128 scalar.

7.4.5.93 vec_xfer_vui8t_2_bin128()

```
static __binary128 vec_xfer_vui8t_2_bin128 (
             vui8_t f128 )  [inline], [static]
```

Transfer a vector unsigned char to __binary128 scalar.

The compiler does not allow direct transfer (assignment or type cast) between __binary128 (__float128) scalars and vector types. This despite the fact the the ABI and ISA require __binary128 in vector registers (VRs).

Note

this function uses a union to effect the (logical) transfer. The compiler should not generate any code for this.

Parameters

f128	a vector unsigned char value.

Returns

The original value returned as a __binary128 scalar.

7.4.5.94 vec_xor_bin128_2_vui32t()

```
static vui32_t vec_xor_bin128_2_vui32t (
            __binary128 f128,
            vui32_t mask )  [inline], [static]
```

Transfer a quadword from a __binary128 scalar to a vector int and logical Exclusive OR with mask.

The compiler does not allow direct transfer (assignment or type cast) between __binary128 (__float128) scalars and vector types. This despite the fact the the ABI and ISA require __binary128 in vector registers (VRs).

Note

this function uses a union to effect the (logical) transfer. The compiler should not generate any code for this.

Parameters

f128	a __binary128 floating point scalar value.
mask	a vector unsigned int

Returns

The original value XORed with mask as a 128-bit vector int.

7.4.5.95 vec_xsaddqpo()

```
static __binary128 vec_xsaddqpo (
            __binary128 vfa,
            __binary128 vfb )  [inline], [static]
```

VSX Scalar Add Quad-Precision using round to Odd.

The quad-precision element of vectors vfa and vfb are added to produce the quad-precision result. The rounding mode is round to odd.

For POWER9 use the xsaddqpo instruction. For POWER8 use this soft-float implementation using vector instruction generated by PVECLIB operations. For POWER7 and earlier us the compilers soft-float implementation.

Note

This operation *may not* follow the PowerISA relative to setting the FPSCR. However if the hardware target includes the xsmulqpo instruction, the implementation may use that.

processor	Latency	Throughput
power8	54-71	1/cycle
power9	12	1/12 cycle

Parameters

vfa	128-bit vector treated as a scalar __binary128.
vfb	128-bit vector treated as a scalar __binary128.

Returns

a vector unsigned __int128 value.

7.4.5.96 vec_xscvdpqp()

```
static __binary128 vec_xscvdpqp (
             vf64_t f64 )  [inline], [static]
```

VSX Scalar Convert Double-Precision to Quad-Precision format.

The left most double-precision element of vector f64 is converted to quad-precision format.

For POWER9 use the xscvdpqp instruction. For POWER8 and earlier use vector instruction generated by PVECLIB operations.

Note

This operation *may not* follow the PowerISA relative to Signaling NaN and setting the FPSCR. However if the hardware target includes the xscvdpqp instruction, the implementation may use that.

processor	Latency	Throughput
power8	?	2/cycle
power9	3	2/cycle

Parameters

f64	a vector double. The left most element is converted.

Returns

a __binary128 value.

7.4.5.97 vec_xscvqpdpo()

```
static vf64_t vec_xscvqpdpo (
             __binary128 f128 )  [inline], [static]
```

VSX Scalar Convert with round Quad-Precision to Double-Precision (using round to odd).

The quad-precision element of vector f128 is converted to double-precision. The Floating point value is rounded to odd before conversion. The result is placed in doubleword element 0 while element 1 is set to zero.

For POWER9 use the xscvqpdpo instruction. For POWER8 and earlier use vector instructions generated by PVECLIB operations.

Note

This operation *may not* follow the PowerISA relative to setting the FPSCR. However if the hardware target includes the xscvqpdpo instruction, the implementation may use that.

processor	Latency	Throughput
power8	?	1/cycle
power9	12	1/cycle

Parameters

f128	128-bit vector treated as a scalar __binary128.

Returns

a vector unsigned long long value.

7.4.5.98 vec_xscvqpudz()

```
static vui64_t vec_xscvqpudz (
             __binary128 f128 )  [inline], [static]
```

VSX Scalar Convert with round to zero Quad-Precision to Unsigned doubleword.

The quad-precision element of vector f128 is converted to an unsigned doubleword integer. The Floating point value is rounded toward zero before conversion. The result is placed in element 0 while element 1 is set to zero.

For POWER9 use the xscvqpudz instruction. For POWER8 and earlier use vector instructions generated by PVECLIB operations.

Note

> This operation *may not* follow the PowerISA relative to setting the FPSCR. However if the hardware target includes the xscvqpudz instruction, the implementation may use that.

processor	Latency	Throughput
power8	?	2/cycle
power9	?	2/cycle

Parameters

f128	128-bit vector treated as a scalar __binary128.

Returns

> a vector unsigned long long value.

7.4.5.99 vec_xscvqpuqz()

```
static vui128_t vec_xscvqpuqz (
             __binary128 f128 )  [inline], [static]
```

VSX Scalar Convert with round to zero Quad-Precision to Unsigned Quadword.

The quad-precision element of vector f128 is converted to an unsigned quadword integer. The Floating point value is rounded toward zero before conversion.

For POWER10 use the xscvqpuqz instruction. For POWER9 and earlier use vector instruction generated by PVECLIB operations.

Note

> This operation *may not* follow the PowerISA relative to setting the FPSCR. However if the hardware target includes the xscvqpuqz instruction, the implementation may use that.

processor	Latency	Throughput
power8	?	2/cycle
power9	?	2/cycle

Parameters

f128	128-bit vector treated as a scalar __binary128.

Returns

a vector unsigned __int128 value.

7.4.5.100 vec_xscvsdqp()

```
static __binary128 vec_xscvsdqp (
            vi64_t int64 )  [inline], [static]
```

VSX Scalar Convert Signed-Doubleword to Quad-Precision format.

The left most signed doubleword element of vector int64 is converted to quad-precision format.

For POWER9 use the xscvsdqp instruction. For POWER8 and earlier use vector instruction generated by PVECLIB operations.

Note

At this point we are not trying to comply with PowerISA by setting any FPSCR bits associated with Quad-Precision convert. If such is required, FR and/or FI can be set using the Move To FPSCR Bit 0 (mtfsb0) instruction.

processor	Latency	Throughput
power8	?	2/cycle
power9	3	2/cycle

Parameters

int64	a vector signed long long. The left most element is converted.

Returns

a __binary128 value.

7.4.5.101 vec_xscvsqqp()

```
static __binary128 vec_xscvsqqp (
            vi128_t int128 )  [inline], [static]
```

VSX Scalar Convert Signed-Quadword to Quad-Precision format.

The signed quadword element of vector int128 is converted to quad-precision format. If the conversion is not exact the default rounding mode is "Round to Nearest Even".

For POWER10 use the xscvuqqp instruction. POWER9 only supports doubleword converts so use a combination of two xscvudqp and xsmaddqp instructions. For POWER8 and earlier use vector instruction generated by PVECLIB operations.

Note

The POWER8 implementation ignores the hardware rounding mode **FPSCR$_{\mathbf{RN}}$**.

At this point we are not trying to comply with PowerISA by setting any FPSCR bits associated with Quad-Precision convert. If such is required, FPFR, FR and FI can be set using the Move To FPSCR Bit 0/1 (mtfsb[0|1]) instruction.

processor	Latency	Throughput
power8	?	2/cycle
power9	44-53	1/13cycles

Parameters

int128	a vector signed __int128 which is converted to QP format.

Returns

a __binary128 value.

7.4.5.102 vec_xscvudqp()

```
static __binary128 vec_xscvudqp (
             vui64_t int64 )  [inline], [static]
```

VSX Scalar Convert Unsigned-Doubleword to Quad-Precision format.

The left most unsigned doubleword element of vector int64 is converted to quad-precision format.

For POWER9 use the xscvudqp instruction. For POWER8 and earlier use vector instruction generated by PVECLIB operations.

Note

At this point we are not trying to comply with PowerISA by setting any FPSCR bits associated with Quad-Precision convert. If such is required, FR and/or FI can be set using the Move To FPSCR Bit 0 (mtfsb0) instruction.

processor	Latency	Throughput
power8	?	2/cycle
power9	3	2/cycle

Parameters

int64	a vector unsigned long long. The left most element is converted.

Returns

a __binary128 value.

7.4.5.103 vec_xscvuqqp()

```
static __binary128 vec_xscvuqqp (
            vui128_t int128 )  [inline], [static]
```

VSX Scalar Convert Unsigned-Quadword to Quad-Precision format.

The unsigned quadword element of vector int128 is converted to quad-precision format. If the conversion is not exact the default rounding mode is "Round to Nearest Even".

For POWER10 use the xscvuqqp instruction. POWER9 only supports doubleword converts so use a combination of two xscvudqp and xsmaddqp instructions. For POWER8 and earlier use vector instruction generated by PVECLIB operations.

Note

The POWER8 implementation ignores the hardware rounding mode **FPSCR**$_{\mathbf{RN}}$.

At this point we are not trying to comply with PowerISA by setting any FPSCR bits associated with Quad-Precision convert. If such is required, FPFR, FR and FI can be set using the Move To FPSCR Bit 0/1 (mtfsb[0|1]) instruction.

processor	Latency	Throughput
power8	?	2/cycle
power9	38-47	1/13cycles

Parameters

int128	a vector unsigned __int128 which is converted to QP format.

Returns

a __binary128 value.

7.4.5.104 vec_xsiexpqp()

```
static __binary128 vec_xsiexpqp (
            vui128_t sig,
            vui64_t exp )  [inline], [static]
```

Scalar Insert Exponent Quad-Precision.

Merge the sign (bit 0) and significand (bits 16:127) from sig with the 15-bit exponent from exp (bits 49:63). The exponent is moved to bits 1:15 of the final result. The result is returned as a Quad_precision floating point value.

Note

> This operation is equivalent to the POWER9 xsiexpqp instruction, This instruction requires a POWER9-enabled compiler targeting -mcpu=power9 and is not available for older compilers nor POWER8 and earlier. We can't use the built-in scalar_insert_exp because it requires scalar (GPR) inputs and vec_insert_exp is not defined for Quad-Precision. We expect (in context) inputs will be in VRs. This operation provides implementations for all VSX-enabled platforms.

processor	Latency	Throughput
power8	6-8	2/cycle
power9	2	4/cycle

Parameters

sig	vector __int128 containing the Sign Bit and 112-bit significand.
exp	vector unsigned long long element 0 containing the 15-bit exponent.

Returns

> a __binary128 value where the exponent bits (1:15) of sig are replaced from bits 49:63 of exp.

7.4.5.105 vec_xsmulqpo()

```
static __binary128 vec_xsmulqpo (
            __binary128 vfa,
            __binary128 vfb )  [inline], [static]
```

VSX Scalar Multiply Quad-Precision using round to Odd.

The quad-precision element of vectors vfa and vfb are multiplied to produce the quad-precision result. The rounding mode is round to odd.

For POWER9 use the xsmulqpo instruction. For POWER8 use this soft-float implementation using vector instruction generated by PVECLIB operations. For POWER7 and earlier is the compilers soft-float implementation.

Note

> This operation *may not* follow the PowerISA relative to setting the FPSCR. However if the hardware target includes the xsmulqpo instruction, the implementation may use that.

processor	Latency	Throughput
power8	78-84	1/cycle
power9	24	1/12 cycle

Parameters

vfa	128-bit vector treated as a scalar __binary128.
vfb	128-bit vector treated as a scalar __binary128.

Returns

a vector unsigned __int128 value.

7.4.5.106 vec_xssubqpo()

```
static __binary128 vec_xssubqpo (
            __binary128 vfa,
            __binary128 vfb )  [inline], [static]
```

VSX Scalar Subtract Quad-Precision using round to Odd.

The quad-precision element of vector vfb is subtracted from vfa to produce the quad-precision result. The rounding mode is round to odd.

For POWER9 use the xssubqpo instruction. For POWER8 use this soft-float implementation using vector instruction generated by PVECLIB operations. For POWER7 and earlier us the compilers soft-float implementation.

Note

This operation *may not* follow the PowerISA relative to setting the FPSCR. However if the hardware target includes the xsmulqpo instruction, the implementation may use that.

processor	Latency	Throughput
power8	51-70	1/cycle
power9	12	1/12 cycle

Parameters

vfa	128-bit vector treated as a scalar __binary128.
vfb	128-bit vector treated as a scalar __binary128.

Returns

a vector unsigned __int128 value.

7.4.5.107 vec_xsxexpqp()

```
static vui64_t vec_xsxexpqp (
            __binary128 f128 )  [inline], [static]
```

Scalar Extract Exponent Quad-Precision.

Extract the quad-precision exponent (bits 1:15) and right justify it to (bits 49:63 of) doubleword 0 of the result vector. The result is returned as vector long long integer value.

Note

This operation is equivalent to the POWER9 xsxexpqp instruction. This instruction requires a POWER9-enabled compiler targeting -mcpu=power9 and is not available for older compilers nor POWER8 and earlier. We can't use the built-in scalar_extract_exp because it returns scalar (GPR) results and vec_extract_exp is not defined for Quad-Precision. We expect (in context) results are needed in VRs. This operation provides implementations for all VSX-enabled platforms.

processor	Latency	Throughput
power8	8-10	2/cycle
power9	2	4/cycle

Parameters

f128	__binary128 scalar value in a vector register.

Returns

vector unsigned long long element 0 containing the 15-bit exponent

7.4.5.108 vec_xsxsigqp()

```
static vui128_t vec_xsxsigqp (
            __binary128 f128 )  [inline], [static]
```

Scalar Extract Significand Quad-Precision.

Extract the quad-precision significand (bits 16:127) and restore the implied (hidden) bit (bit 15) if the quad-precition value is normal (not zero, subnormal, Infinity or NaN). The result is returned as vector __int128 integer value with up to 113 bits of significance.

Note

This operation is equivalent to the POWER9 xsxsigqp instruction. This instruction requires a POWER9-enabled compiler targeting -mcpu=power9 and is not available for older compilers nor POWER8 and earlier. We can't use the built-in scalar_extract_sig because it returns scalar (GPR) results and vec_extract_sig is not defined for Quad-Precision. We expect (in context) results are needed in VRs. This operation provides implementations for all VSX-enabled platforms.

processor	Latency	Throughput
power8	12-14	1/6cycles
power9	3	2/cycle

Parameters

f128	__binary128 scalar value in a vector register.

Returns

vector __int128 containing the significand.

7.4.5.109 vec_xxxexpqpp()

```
static vui64_t vec_xxxexpqpp (
            __binary128 vfa,
            __binary128 vfb )  [inline], [static]
```

Vector Extract Exponent Quad-Precision Pair.

Extract the quad-precision exponent (bits 1:15), from each member of the QP Pair, and right justify the exponents to (bits 49:63 of) doublewords 0/1 of the result vector. The result is returned as vector long long integer value.

Note

This operation is equivalent to the POWER9 instruction sequence xsxexpqp/xsxexpqp/xxmrghd.

processor	Latency	Throughput
power8	6-8	1/cycle
power9	5	2/cycle

Parameters

vfa	first __binary128 scalar value in a vector register.
vfb	first __binary128 scalar value in a vector register.

Returns

vector unsigned long long elements 0/1 containing 15-bit exponents from vfa/vfb.

7.5 src/pveclib/vec_f32_ppc.h File Reference

Header package containing a collection of 128-bit SIMD operations over 4x32-bit floating point elements.

```
#include <pveclib/vec_common_ppc.h>
#include <pveclib/vec_int128_ppc.h>
```

Typedefs

- typedef vf32_t __vbinary32

  typedef __vbinary32 to vector of 4 xfloat elements.

Functions

- static vf32_t vec_absf32 (vf32_t vf32x)

  Vector float absolute value.
- static int vec_all_isfinitef32 (vf32_t vf32)

  Return true if all 4x32-bit vector float values are Finite (Not NaN nor Inf).
- static int vec_all_isinff32 (vf32_t vf32)

  Return true if all 4x32-bit vector float values are infinity.
- static int vec_all_isnanf32 (vf32_t vf32)

  Return true if all of 4x32-bit vector float values are NaN.
- static int vec_all_isnormalf32 (vf32_t vf32)

  Return true if all of 4x32-bit vector float values are normal (Not NaN, Inf, denormal, or zero).
- static int vec_all_issubnormalf32 (vf32_t vf32)

  Return true if all of 4x32-bit vector float values is subnormal (denormal).
- static int vec_all_iszerof32 (vf32_t vf32)

  Return true if all of 4x32-bit vector float values are +-0.0.
- static int vec_any_isfinitef32 (vf32_t vf32)

  Return true if any 4x32-bit vector float values are Finite (Not NaN nor Inf).
- static int vec_any_isinff32 (vf32_t vf32)

  Return true if any 4x32-bit vector float values are infinity.
- static int vec_any_isnanf32 (vf32_t vf32)

  Return true if any of 4x32-bit vector float values are NaN.
- static int vec_any_isnormalf32 (vf32_t vf32)

  Return true if any of 4x32-bit vector float values are normal (Not NaN, Inf, denormal, or zero).
- static int vec_any_issubnormalf32 (vf32_t vf32)

  Return true if any of 4x32-bit vector float values is subnormal (denormal).
- static int vec_any_iszerof32 (vf32_t vf32)

  Return true if any of 4x32-bit vector float values are +-0.0.

- static vf32_t vec_copysignf32 (vf32_t vf32x, vf32_t vf32y)

    Copy the sign bit from vf32x merged with magnitude from vf32y and return the resulting vector float values.
- static vb32_t vec_isfinitef32 (vf32_t vf32)

    Return 4x32-bit vector boolean true values for each float element that is Finite (Not NaN nor Inf).
- static vb32_t vec_isinff32 (vf32_t vf32)

    Return 4x32-bit vector boolean true values for each float, if infinity.
- static vb32_t vec_isnanf32 (vf32_t vf32)

    Return 4x32-bit vector boolean true values, for each float NaN value.
- static vb32_t vec_isnormalf32 (vf32_t vf32)

    Return 4x32-bit vector boolean true values, for each float value, if normal (Not NaN, Inf, denormal, or zero).
- static vb32_t vec_issubnormalf32 (vf32_t vf32)

    Return 4x32-bit vector boolean true values, for each float value that is subnormal (denormal).
- static vb32_t vec_iszerof32 (vf32_t vf32)

    Return 4x32-bit vector boolean true values, for each float value that is +-0.0.
- static vb32_t vec_setb_sp (vf32_t vra)

    Vector Set Bool from Sign, Single Precision.
- static vf32_t vec_vgl4fsso (float ∗array, const long long offset0, const long long offset1, const long long offset2, const long long offset3)

    Vector Gather-Load 4 Words from scalar Offsets.
- static vf32_t vec_vgl4fswo (float ∗array, vi32_t vra)

    Vector Gather-Load 4 Words from Vector Word Offsets.
- static vf32_t vec_vgl4fswsx (float ∗array, vi32_t vra, const unsigned char scale)

    Vector Gather-Load 4 Words from Vector Word Scaled Indexes.
- static vf32_t vec_vgl4fswx (float ∗array, vi32_t vra)

    Vector Gather-Load 4 Words from Vector Word Indexes.
- static vf64_t vec_vglfsdo (float ∗array, vi64_t vra)

    Vector Gather-Load Single Floats from Vector Doubleword Offsets.
- static vf64_t vec_vglfsdsx (float ∗array, vi64_t vra, const unsigned char scale)

    Vector Gather-Load Single Floats from Vector Doubleword Scaled Indexes.
- static vf64_t vec_vglfsdx (float ∗array, vi64_t vra)

    Vector Gather-Load Single Floats from Vector Doubleword Indexes.
- static vf64_t vec_vglfsso (float ∗array, const long long offset0, const long long offset1)

    Vector Gather-Load Float Single from scalar Offsets.
- static vf64_t vec_vlxsspx (const signed long long ra, const float ∗rb)

    Vector Load Scalar Single Float Indexed.
- static void vec_vsst4fsso (vf32_t xs, float ∗array, const long long offset0, const long long offset1, const long long offset2, const long long offset3)

    Vector Scatter-Store 4 Float Singles to Scalar Offsets.
- static void vec_vsst4fswo (vf32_t xs, float ∗array, vi32_t vra)

    Vector Scatter-Store 4 Float Singles to Vector Word Offsets.
- static void vec_vsst4fswsx (vf32_t xs, float ∗array, vi32_t vra, const unsigned char scale)

    Vector Scatter-Store 4 Float Singles to Vector Word Indexes.
- static void vec_vsst4fswx (vf32_t xs, float ∗array, vi32_t vra)

    Vector Scatter-Store 4 Float Singles to Vector Word Indexes.
- static void vec_vsstfsdo (vf64_t xs, float ∗array, vi64_t vra)

    Vector Scatter-Store Floats Singles to Vector Doubleword Offsets.
- static void vec_vsstfsdsx (vf64_t xs, float ∗array, vi64_t vra, const unsigned char scale)

Vector Scatter-Store Words to Vector Doubleword Scaled Indexes.

- static void vec_vsstfsdx (vf64_t xs, float ∗array, vi64_t vra)

  Vector Scatter-Store Words to Vector Doubleword Indexes.
- static void vec_vsstfsso (vf64_t xs, float ∗array, const long long offset0, const long long offset1)

  Vector Scatter-Store Float Singles to Scalar Offsets.
- static void vec_vstxsspx (vf64_t xs, const signed long long ra, float ∗rb)

  Vector Store Scalar Single Float Indexed.
- static vf32_t vec_xviexpsp (vui32_t sig, vui32_t exp)

  Vector Insert Exponent Single-Precision.
- static vui32_t vec_xvxexpsp (vf32_t vrb)

  Vector Extract Exponent Single-Precision.
- static vui32_t vec_xvxsigsp (vf32_t vrb)

  Vector Extract Significand Single-Precision.

7.5.1 Detailed Description

Header package containing a collection of 128-bit SIMD operations over 4x32-bit floating point elements.

Most vector float (32-bit float) operations are implemented with PowerISA VMX instructions either defined by the original VMX (a.k.a. Altivec) or added to later versions of the PowerISA. POWER8 added the Vector Scalar Extended (VSX) with access to additional vector registers (64 total) and operations. Most of these operations (compiler built-ins, or intrinsics) are defined in <altivec.h> and described in the `compiler documentation`.

Note

> The compiler disables associated <altivec.h> built-ins if the **mcpu** target does not enable the specific instruction. For example if you compile with **-mcpu=power7**, some of the wordwise pack, unpack and merge operations useful for conversions are not defined and the equivalent vec_perm and permute control must be used instead. This header will provide the appropriate substitutions, will generate the minimum code, appropriate for the target, and produce correct results.
>
> Most ppc64le compilers will default to -mcpu=power8 if not specified.

Most of these operations are implemented in a single instruction on newer (POWER8/POWER9) processors. This header serves to fill in functional gaps for older (POWER7, POWER8) processors and provides an inline assembler implementation for older compilers that do not provide the built-ins.

POWER9 adds useful vector float operations, including: test data class, extract exponent, extract significand, and insert exponent. These operations are common in math library implementations.

Note

> GCC 7.3 defines vector forms of the test data class, extract significand, and extract/insert_exp for float and double. These built-ins are not defined in GCC 6.4. See `compiler documentation`. These are useful operations and can be implement in a few vector logical instruction for earlier machines.

So it is reasonable for this header to provide vector forms of the floating point classification functions (isnormal/subnormal/finite/inf/nan/zero, etc.). These functions can be implemented directly using (one or more) POWER9 instructions, or a few vector logical and integer compare instructions for POWER7/8. Each is comfortably small enough to be in-lined and inherently faster than the equivalent POSIX or compiler built-in runtime scalar functions.

This header covers operations that are any of the following:

- Implemented in hardware instructions in newer processors, but useful to programmers on slightly older processors (even if the equivalent function requires more instructions). Examples include the floating point test data class, extract exponent, extract significand, and insert exponent operations.
- Defined in the OpenPOWER ABI but *not* yet defined in <altivec.h> provided by available compilers in common use. Examples include vector float even/odd.
- Providing special vector float tests for special conditions without generating extraneous floating-point exceptions. This is important for implementing vectorized forms of ISO C99 Math functions.
- Commonly used operations, not covered by the ABI or <altivec.h>, and require multiple instructions or are not obvious.

7.5.2 Examples

For example: using the the classification functions for implementing the math library function sine and cosine. The POSIX specification requires that special input values are processed without raising extraneous floating point exceptions and return specific floating point values in response. For example the sin() function.

- If the input *value* is NaN then return a NaN.
- If the input *value* is +-0.0 then return *value*.
- If the input *value* is subnormal then return *value*.
- If the input *value* is +-Inf then return a NaN.
- Otherwise compute and return sin(value).

The following code example uses functions from this header to address the POSIX requirements for special values input to for a vectorized sinf():

```
vf32_t
test_vec_sinf32 (vf32_t value)
{
  const vf32_t vec_f0 = { 0.0, 0.0, 0.0, 0.0 };
  const vui32_t vec_f32_qnan =
    { 0x7f800001, 0x7fc00000, 0x7fc00000, 0x7fc00000 };
  vf32_t result;
  vb32_t normmask, infmask;
  normmask = vec_isnormalf32 (value);
  if (vec_any_isnormalf32 (value))
    {
      // replace non-normal input values with safe values.
      vf32_t safeval = vec_sel (vec_f0, value, normmask);
      // body of vec_sin(safeval) computation elided for this example.
    }
  else
    result = value;
  // merge non-normal input values back into result
  result = vec_sel (value, result, normmask);
  // Inf input value elements return quiet-nan
  infmask = vec_isinff32 (value);
  result = vec_sel (result, (vf32_t) vec_f32_qnan, infmask);
  return result;
}
```

The code generated for this fragment runs between 24 (-mcpu=power9) and 40 (-mcpu=power8) instructions. The normal execution path is 14 to 25 instructions respectively.

Another example the cos() function.

- If the input *value* is NaN then return a NaN.
- If the input *value* is +-0.0 then return *1.0*.
- If the input *value* is +-Inf then return a NaN.
- Otherwise compute and return cos(value).

The following code example uses functions from this header to address the POSIX requirements for special values input to vectorized cosf():

```
vf32_t
test_vec_cosf32 (vf32_t value)
{
  vf32_t result;
  const vf32_t vec_f0 = { 0.0, 0.0, 0.0, 0.0 };
  const vf32_t vec_f1 = { 1.0, 1.0, 1.0, 1.0 };
  const vui32_t vec_f32_qnan =
    { 0x7f800001, 0x7fc00000, 0x7fc00000, 0x7fc00000 };
  vb32_t finitemask, infmask, zeromask;
  finitemask = vec_isfinitef32 (value);
  if (vec_any_isfinitef32 (value))
    {
      // replace non-finite input values with safe values
      vf32_t safeval = vec_sel (vec_f0, value, finitemask);
      // body of vec_sin(safeval) computation elided for this example
    }
  else
    result = value;
  // merge non-finite input values back into result
  result = vec_sel (value, result, finitemask);
  // Set +-0.0 input elements to exactly 1.0 in result
  zeromask = vec_iszerof32 (value);
  result = vec_sel (result, vec_f1, zeromask);
  // Set Inf input elements to quiet-nan in result
  infmask = vec_isinff32 (value);
  result = vec_sel (result, (vf32_t) vec_f32_qnan, infmask);
  return result;
}
```

Neither example raises floating point exceptions or sets **errno**, as appropriate for a vector math library.

7.5.3 Performance data.

High level performance estimates are provided as an aid to function selection when evaluating algorithms. For background on how *Latency* and *Throughput* are derived see: Performance data.

7.5.4 Function Documentation

7.5.4.1 vec_absf32()

```
static vf32_t vec_absf32 (
             vf32_t vf32x )  [inline], [static]
```

Vector float absolute value.

processor	Latency	Throughput
power8	6-7	2/cycle
power9	2	2/cycle

Parameters

vf32x	vector float values containing the magnitudes.

Returns

vector absolute values of 4x float elements of vf32x.

7.5.4.2 vec_all_isfinitef32()

```
static int vec_all_isfinitef32 (
             vf32_t vf32 )  [inline], [static]
```

Return true if all 4x32-bit vector float values are Finite (Not NaN nor Inf).

A IEEE Binary32 finite value has an exponent between 0x000 and 0x7f0 (a 0x7f8 indicates NaN or Inf). The significand can be any value. The sign bit is ignored.

Note

This function will not raise VXSNAN or VXVC (FE_INVALID) exceptions. A normal float compare can.

processor	Latency	Throughput
power8	4-20	2/cycle
power9	6	1/cycle

Parameters

vf32	a vector of __binary32 values.

Returns

an int containing 0 or 1.

7.5.4.3 vec_all_isinff32()

```
static int vec_all_isinff32 (
             vf32_t vf32 )  [inline], [static]
```

Return true if all 4x32-bit vector float values are infinity.

A IEEE Binary32 infinity has a exponent of 0x7f8 and significand of all zeros. The sign bit is ignored.

Note

This function will not raise VXSNAN or VXVC (FE_INVALID) exceptions. A normal float compare can.

processor	Latency	Throughput
power8	6-20	2/cycle
power9	6	1/cycle

Parameters

vf32	a vector of __binary32 values.

Returns

boolean int, true if all 4 float values are infinity

7.5.4.4 vec_all_isnanf32()

```
static int vec_all_isnanf32 (
             vf32_t vf32 )  [inline], [static]
```

Return true if all of 4x32-bit vector float values are NaN.

A IEEE Binary32 NaN value has an exponent between 0x7f8 and the significand is nonzero. The sign bit is ignored.

Note

This function will not raise VXSNAN or VXVC (FE_INVALID) exceptions. A normal float compare can.

processor	Latency	Throughput
power8	6-20	2/cycle
power9	6	1/cycle

Parameters

vf32	a vector of __binary32 values.

Returns

a boolean int, true if all of 4 vector float values are NaN.

7.5.4.5 vec_all_isnormalf32()

```
static int vec_all_isnormalf32 (
             vf32_t vf32 )  [inline], [static]
```

Return true if all of 4x32-bit vector float values are normal (Not NaN, Inf, denormal, or zero).

A IEEE Binary32 normal value has an exponent between 0x008 and 0x7f (a 0x7f8 indicates NaN or Inf). The significand can be any value (expect 0 if the exponent is zero). The sign bit is ignored.

Note

This function will not raise VXSNAN or VXVC (FE_INVALID) exceptions. A normal float compare can.

processor	Latency	Throughput
power8	6-20	1/cycle
power9	6	1/cycle

Parameters

vf32	a vector of __binary32 values.

Returns

a boolean int, true if all of 4 vector float values are normal.

7.5.4.6 vec_all_issubnormalf32()

```
static int vec_all_issubnormalf32 (
             vf32_t vf32 )  [inline], [static]
```

Return true if all of 4x32-bit vector float values is subnormal (denormal).

A IEEE Binary32 subnormal has an exponent of 0x000 and a nonzero significand. The sign bit is ignored.

Note

This function will not raise VXSNAN or VXVC (FE_INVALID) exceptions. A normal float compare can.

processor	Latency	Throughput
power8	10-30	1/cycle
power9	6	1/cycle

Parameters

vf32	a vector of __binary32 values.

Returns

a boolean int, true if all of 4 vector float values are subnormal.

7.5.4.7 vec_all_iszerof32()

```
static int vec_all_iszerof32 (
             vf32_t vf32 )  [inline], [static]
```

Return true if all of 4x32-bit vector float values are +-0.0.

A IEEE Binary32 zero has an exponent of 0x000 and a zero significand. The sign bit is ignored.

Note

This function will not raise VXSNAN or VXVC (FE_INVALID) exceptions. A normal float compare can.

processor	Latency	Throughput
power8	6-20	2/cycle
power9	6	1/cycle

Parameters

vf32	a vector of __binary32 values.

Returns

a boolean int, true if all of 4 vector float values are +/- zero.

7.5.4.8 vec_any_isfinitef32()

```
static int vec_any_isfinitef32 (
            vf32_t vf32 )  [inline], [static]
```

Return true if any 4x32-bit vector float values are Finite (Not NaN nor Inf).

A IEEE Binary32 finite value has an exponent between 0x000 and 0x7f0 (a 0x7f8 indicates NaN or Inf). The significand can be any value. The sign bit is ignored.

Note

This function will not raise VXSNAN or VXVC (FE_INVALID) exceptions. A normal float compare can.

processor	Latency	Throughput
power8	4-20	2/cycle
power9	6	1/cycle

Parameters

vf32	a vector of __binary32 values.

Returns

an int containing 0 or 1.

7.5.4.9 vec_any_isinff32()

```
static int vec_any_isinff32 (
            vf32_t vf32 )  [inline], [static]
```

Return true if any 4x32-bit vector float values are infinity.

A IEEE Binary32 infinity has a exponent of 0x7f8 and significand of all zeros.

Note

This function will not raise VXSNAN or VXVC (FE_INVALID) exceptions. A normal float compare can.

processor	Latency	Throughput
power8	6-20	2/cycle
power9	6	2/cycle

Parameters

vf32	a vector of __binary32 values.

Returns

boolean int, true if any of 4 float values are infinity

7.5.4.10 vec_any_isnanf32()

```
static int vec_any_isnanf32 (
            vf32_t vf32 )  [inline], [static]
```

Return true if any of 4x32-bit vector float values are NaN.

A IEEE Binary32 NaN value has an exponent between 0x7f8 and the significand is nonzero. The sign bit is ignored.

Note

This function will not raise VXSNAN or VXVC (FE_INVALID) exceptions. A normal float compare can.

processor	Latency	Throughput
power8	6-20	2/cycle
power9	6	2/cycle

Parameters

vf32	a vector of __binary32 values.

Returns

a boolean int, true if any of 4 vector float values are NaN.

7.5.4.11 vec_any_isnormalf32()

```
static int vec_any_isnormalf32 (
            vf32_t vf32 )  [inline], [static]
```

Return true if any of 4x32-bit vector float values are normal (Not NaN, Inf, denormal, or zero).

A IEEE Binary32 normal value has an exponent between 0x008 and 0x7f (a 0x7f8 indicates NaN or Inf). The significand can be any value (expect 0 if the exponent is zero). The sign bit is ignored.

Note

This function will not raise VXSNAN or VXVC (FE_INVALID) exceptions. A normal float compare can.

processor	Latency	Throughput
power8	10-24	1/cycle
power9	6	1/cycle

Parameters

vf32	a vector of __binary32 values.

Returns

a boolean int, true if any of 4 vector float values are normal.

7.5.4.12 vec_any_issubnormalf32()

```
static int vec_any_issubnormalf32 (
            vf32_t vf32 )  [inline], [static]
```

Return true if any of 4x32-bit vector float values is subnormal (denormal).

A IEEE Binary32 subnormal has an exponent of 0x000 and a nonzero significand. The sign bit is ignored.

Note

This function will not raise VXSNAN or VXVC (FE_INVALID) exceptions. A normal float compare can.

processor	Latency	Throughput
power8	10-18	1/cycle
power9	6	1/cycle

Parameters

vf32	a vector of __binary32 values.

Returns

if any of 4 vector float values are subnormal.

7.5.4.13 vec_any_iszerof32()

```
static int vec_any_iszerof32 (
             vf32_t vf32 )  [inline], [static]
```

Return true if any of 4x32-bit vector float values are +-0.0.

A IEEE Binary32 zero has an exponent of 0x000 and a zero significand. The sign bit is ignored.

Note

This function will not raise VXSNAN or VXVC (FE_INVALID) exceptions. A normal float compare can.

processor	Latency	Throughput
power8	6-20	2/cycle
power9	6	1/cycle

Parameters

vf32	a vector of __binary32 values.

Returns

a boolean int, true if any of 4 vector float values are +/- zero.

7.5.4.14 vec_copysignf32()

```
static vf32_t vec_copysignf32 (
             vf32_t vf32x,
             vf32_t vf32y )  [inline], [static]
```

Copy the sign bit from vf32x merged with magnitude from vf32y and return the resulting vector float values.

Note

This operation was patterned after the intrinsic vec_cpsgn (altivec.h) introduced for POWER7 and VSX. It turns out the original (GCC 4.9) compiler implementation reversed the operands and does not match the PowerISA or the Vector Intrinsic Programming Reference manuals. Subsequent compilers and PVECLIB implementations replicated this (operand order) error. This has now been reported as bug against the compilers, which are in the process of applying fixes and distributing updates. This version of PVECLIB is updated to match the Vector Intrinsic Programming Reference. This implementation is independent of the compilers update status.

processor	Latency	Throughput
power8	6-7	2/cycle
power9	2	2/cycle

Parameters

vf32x	vector float values containing the sign bits.
vf32y	vector float values containing the magnitudes.

Returns

vector float values with magnitude from vf32y and the sign of vf32x.

7.5.4.15 vec_isfinitef32()

```
static vb32_t vec_isfinitef32 (
            vf32_t vf32 )  [inline], [static]
```

Return 4x32-bit vector boolean true values for each float element that is Finite (Not NaN nor Inf).

A IEEE Binary32 finite value has an exponent between 0x000 and 0x7f0 (a 0x7f8 indicates NaN or Inf). The significand can be any value. Using the vec_cmpeq conditional to generate the predicate mask for NaN / Inf and then invert this for the finite condition. The sign bit is ignored.

Note

This function will not raise VXSNAN or VXVC (FE_INVALID) exceptions. A normal float compare can.

processor	Latency	Throughput
power8	6-15	2/cycle
power9	5	2/cycle

Parameters

vf32	a vector of __binary32 values.

Returns

a vector boolean int, each containing all 0s(false) or 1s(true).

7.5.4.16 vec_isinff32()

```
static vb32_t vec_isinff32 (
            vf32_t vf32 )  [inline], [static]
```

Return 4x32-bit vector boolean true values for each float, if infinity.

A IEEE Binary32 infinity has a exponent of 0x7f8 and significand of all zeros.

Note

This function will not raise VXSNAN or VXVC (FE_INVALID) exceptions. A normal float compare can.

processor	Latency	Throughput
power8	4-13	2/cycle
power9	3	2/cycle

Parameters

vf32	a vector of __binary32 values.

Returns

a vector boolean int, each containing all 0s(false) or 1s(true).

7.5.4.17 vec_isnanf32()

```
static vb32_t vec_isnanf32 (
            vf32_t vf32 )  [inline], [static]
```

Return 4x32-bit vector boolean true values, for each float NaN value.

A IEEE Binary32 NaN value has an exponent between 0x7f8 and the significand is nonzero. The sign bit is ignored.

processor	Latency	Throughput
power8	4-13	2/cycle
power9	3	2/cycle

Parameters

vf32	a vector of __binary32 values.

Returns

a vector boolean int, each containing all 0s(false) or 1s(true).

7.5.4.18 vec_isnormalf32()

```
static vb32_t vec_isnormalf32 (
            vf32_t vf32 )  [inline], [static]
```

Return 4x32-bit vector boolean true values, for each float value, if normal (Not NaN, Inf, denormal, or zero).

A IEEE Binary32 normal value has an exponent between 0x008 and 0x7f (a 0x7f8 indicates NaN or Inf). The significand can be any value (expect 0 if the exponent is zero). The sign bit is ignored.

Note

This function will not raise VXSNAN or VXVC (FE_INVALID) exceptions. A normal float compare can.

processor	Latency	Throughput
power8	6-15	1/cycle
power9	5	1/cycle

Parameters

vf32	a vector of __binary32 values.

Returns

a vector boolean int, each containing all 0s(false) or 1s(true).

7.5.4.19 vec_issubnormalf32()

```
static vb32_t vec_issubnormalf32 (
            vf32_t vf32 )  [inline], [static]
```

Return 4x32-bit vector boolean true values, for each float value that is subnormal (denormal).

A IEEE Binary32 subnormal has an exponent of 0x000 and a nonzero significand. The sign bit is ignored.

Note

This function will not raise VXSNAN or VXVC (FE_INVALID) exceptions. A normal float compare can.

processor	Latency	Throughput
power8	6-16	1/cycle
power9	3	1/cycle

Parameters

vf32	a vector of __binary32 values.

Returns

a vector boolean int, each containing all 0s(false) or 1s(true).

7.5.4.20 vec_iszerof32()

```
static vb32_t vec_iszerof32 (
              vf32_t vf32 )  [inline], [static]
```

Return 4x32-bit vector boolean true values, for each float value that is +-0.0.

A IEEE Binary32 zero has an exponent of 0x000 and a zero significand. The sign bit is ignored.

Note

This function will not raise VXSNAN or VXVC (FE_INVALID) exceptions. A normal float compare can.

processor	Latency	Throughput
power8	4-13	2/cycle
power9	5	2/cycle

Parameters

vf32	a vector of __binary32 values.

Returns

a vector boolean int, each containing all 0s(false) or 1s(true).

7.5.4.21 vec_setb_sp()

```
static vb32_t vec_setb_sp (
              vf32_t vra )  [inline], [static]
```

Vector Set Bool from Sign, Single Precision.

For each float, propagate the sign bit to all 32-bits of that word. The result is vector bool int reflecting the sign bit of each 32-bit float.

The resulting mask can be used in masking and select operations.

Note

This operation will set the sign mask regardless of data class, while the Vector Test Data Class will not distinguish between +/- NaN.

processor	Latency	Throughput
power8	2-9	2/cycle
power9	2-8	2/cycle

Parameters

vra	Vector float.

Returns

vector bool int reflecting the sign bits of each float value.

7.5.4.22 vec_vgl4fsso()

```
static vf32_t vec_vgl4fsso (
            float * array,
            const long long offset0,
            const long long offset1,
            const long long offset2,
            const long long offset3 )  [inline], [static]
```

Vector Gather-Load 4 Words from scalar Offsets.

For each scalar offset[0,1,2,3], load the word from the effective address formed by ∗(char∗)array+offset[0-3]. Merge resulting float single word elements [0,1,2,3] and return the resulting vector.

processor	Latency	Throughput
power8	10	1/cycle
power9	11	1/cycle

Parameters

array	Pointer to array of integer words.
offset0	Scalar (64-bit) byte offset from &array.
offset1	Scalar (64-bit) byte offset from &array.
offset2	Scalar (64-bit) byte offset from &array.
offset3	Scalar (64-bit) byte offset from &array.

Returns

vector word containing word elements [0-3] loaded from ∗(char∗)array+offset[0-3].

7.5.4.23 vec_vgl4fswo()

```
static vf32_t vec_vgl4fswo (
             float * array,
             vi32_t vra )  [inline], [static]
```

Vector Gather-Load 4 Words from Vector Word Offsets.

For each signed word element [i] of vra, load the float single word element at ∗(char∗)array+vra[i]. Merge those word elements [0-3] and return the resulting vector.

Note

Signed word offsets are expanded (unpacked) to doublewords before transfer to GRPs for effective address calculation.

processor	Latency	Throughput
power8	14	1/cycle
power9	15	1/cycle

Parameters

array	Pointer to array of integer words.
vra	Vector of signed word (32-bit) byte offsets from &array.

Returns

vector word containing word elements [0-3], each loaded from ∗(char∗)array+vra[0-3].

7.5.4.24 vec_vgl4fswsx()

```
static vf32_t vec_vgl4fswsx (
             float * array,
             vi32_t vra,
             const unsigned char scale )  [inline], [static]
```

Vector Gather-Load 4 Words from Vector Word Scaled Indexes.

For each signed word element [i] of vra, load the float single word element at array[vra[i] << scale]. Merge those word elements [0-3] and return the resulting vector.

Note

Signed word indexes are expanded (unpacked) to doublewords before shifting left (2+scale) bits before transfer to GRPs for effective address calculation. This converts each index to an 64-bit offset.

processor	Latency	Throughput
power8	16-25	1/cycle
power9	18-27	1/cycle

Parameters

array	Pointer to array of integer words.
vra	Vector of signed word (32-bit) indexes.
scale	8-bit integer. Indexes are multiplying by 2^{scale}.

Returns

vector word containing word elements [0-3] each loaded from array[vra[0-3] $<<$ scale].

7.5.4.25 vec_vgl4fswx()

```
static vf32_t vec_vgl4fswx (
             float * array,
             vi32_t vra )  [inline], [static]
```

Vector Gather-Load 4 Words from Vector Word Indexes.

For word element [i] of vra, load the float single word element at array[vra[i]]. Merge those word elements [0-3] and return the resulting vector.

Note

Signed word indexes are expanded (unpacked) to doublewords before shifting left 2 bits. This converts each index to an 64-bit offset for effective address calculation.

processor	Latency	Throughput
power8	16-25	1/cycle
power9	18-27	1/cycle

Parameters

array	Pointer to array of integer words.
vra	Vector of signed word (32-bit) indexes.

Returns

vector word containing word elements [0-3], each loaded from array[vra[0-3]].

7.5.4.26 vec_vglfsdo()

```
static vf64_t vec_vglfsdo (
             float * array,
             vi64_t vra )  [inline], [static]
```

Vector Gather-Load Single Floats from Vector Doubleword Offsets.

For each doubleword element [0-1] of vra, load the float single word element at ∗(char∗)array+vra[i] expanding them to float double format. Merge doubleword elements [0,1] and return the resulting vector.

processor	Latency	Throughput
power8	12	1/cycle
power9	11	1/cycle

Parameters

array	Pointer to array of float singles.
vra	Vector of doubleword (64-bit) byte offsets from &array.

Returns

vector doubleword elements [0,1] loaded from expanded float single words at ∗(char∗)array+vra[i].

7.5.4.27 vec_vglfsdsx()

```
static vf64_t vec_vglfsdsx (
             float * array,
             vi64_t vra,
             const unsigned char scale )  [inline], [static]
```

Vector Gather-Load Single Floats from Vector Doubleword Scaled Indexes.

For each doubleword element [0-1] of vra, load the float single word element at array[vra[i] << scale)]. Merge doubleword elements [0,1] and return the resulting vector.

processor	Latency	Throughput
power8	14-23	1/cycle
power9	13-22	1/cycle

Parameters

array	Pointer to array of float.
vra	Vector of doubleword indexes from &array.
scale	8-bit integer. Indexes are multiplying by 2^{scale}.

Returns

vector doubleword elements [0,1] loaded from the float single words at array[vra[0,1]<<scale].

7.5.4.28 vec_vglfsdx()

```
static vf64_t vec_vglfsdx (
             float * array,
             vi64_t vra )  [inline], [static]
```

Vector Gather-Load Single Floats from Vector Doubleword Indexes.

For each doubleword element [0-1] of vra, load the float single word element at array[vra[i]]. Merge doubleword elements [0,1] and return the resulting vector.

Note

As effective address calculation is modulo 64-bits, signed or unsigned doubleword offsets are equivalent.

processor	Latency	Throughput
power8	14-23	1/cycle
power9	13-22	1/cycle

Parameters

array	Pointer to array of float.
vra	Vector of doubleword indexes from &array.

Returns

vector doubleword elements [0,1] loaded from float single words at array[vra[0,1]].

7.5.4.29 vec_vglfsso()

```
static vf64_t vec_vglfsso (
            float * array,
            const long long offset0,
            const long long offset1 )  [inline], [static]
```

Vector Gather-Load Float Single from scalar Offsets.

For each scalar offset[0|1], load the float single element at ∗(char∗)array+offset[0|1] expanding them to float double format. Merge doubleword elements [0,1] and return the resulting vector.

processor	Latency	Throughput
power8	7	2/cycle
power9	11	2/cycle

Parameters

array	Pointer to array of floats.
offset0	Scalar (64-bit) byte offsets from &array.
offset1	Scalar (64-bit) byte offsets from &array.

Returns

vector double containing elements loaded from ∗(char∗)array+offset0 and ∗(char∗)array+offset1.

7.5.4.30 vec_vlxsspx()

```
static vf64_t vec_vlxsspx (
            const signed long long ra,
            const float * rb )  [inline], [static]
```

Vector Load Scalar Single Float Indexed.

Load doubleword[0] of vector **xt** as a scalar (double float formatted) single float word from the effective address formed by **rb+ra**. The operand **rb** is a pointer to an array of float words. The operand **ra** is a doubleword integer byte offset from **rb**. The result **xt** is returned as a vf64_t vector. For best performance **rb** and **ra** should be word aligned (integer multiple of 4).

Note

The Left most doubleword is the single float value, expanded and formated as a double float. The right most doubleword of vector **xt** is left *undefined* by this operation.

This operation is an alternate form of Vector Load Element (vec_lde), with the added simplification that data is always left justified in the vector. Another advantage for Power8 and later, the lxsspx instruction can load directly into any of the 64 VSRs, while expanding the single float word value into float double format, in a single operation. Both simplify merging elements for gather operations.

Note

The lxsspx instruction was introduced in PowerISA 2.07 (POWER8). Power7 and earlier will use lfs[x] and xxpermdi to move the result from VSR/FPR range to VSR/VR range if needed.

processor	Latency	Throughput
power8	5	2/cycle
power9	8	2/cycle

Parameters

ra	const doubleword index (offset/displacement).
rb	const pointer to an array of floats.

Returns

The word stored at (ra + rb) is expanded from single to double float format and loaded into vector doubleword element 0. Element 1 is undefined.

7.5.4.31 vec_vsst4fsso()

```
static void vec_vsst4fsso (
             vf32_t xs,
             float * array,
             const long long offset0,
             const long long offset1,
             const long long offset2,
             const long long offset3 )  [inline], [static]
```

Vector Scatter-Store 4 Float Singles to Scalar Offsets.

For each float word element [0-3] of xs, store the float element xs[i] at *(char*)array+offset[i].

processor	Latency	Throughput
power8	6	1/cycle
power9	4	2/cycle

Parameters

xs	Vector float elements to scatter store.
array	Pointer to array of float words.
offset0	Scalar (64-bit) byte offset from &array.
offset1	Scalar (64-bit) byte offset from &array.
offset2	Scalar (64-bit) byte offset from &array.

Parameters

offset3	Scalar (64-bit) byte offset from &array.

7.5.4.32 vec_vsst4fswo()

```
static void vec_vsst4fswo (
            vf32_t xs,
            float * array,
            vi32_t vra ) [inline], [static]
```

Vector Scatter-Store 4 Float Singles to Vector Word Offsets.

For each float word element [0-3] of xs, store the float element xs[i] at ∗(char∗)array+vra[i].

Note

> Signed word offsets are expanded (unpacked) to doublewords before transfer to GRPs for effective address calculation.

processor	Latency	Throughput
power8	10	1/cycle
power9	12	2/cycle

Parameters

xs	Vector float elements to scatter store.
array	Pointer to array of float words.
vra	Vector of signed word (32-bit) byte offsets from &array.

7.5.4.33 vec_vsst4fswsx()

```
static void vec_vsst4fswsx (
            vf32_t xs,
            float * array,
            vi32_t vra,
            const unsigned char scale ) [inline], [static]
```

Vector Scatter-Store 4 Float Singles to Vector Word Indexes.

For each float word element [0-4] of xs, store the float element xs[i] at ∗(char∗)array[vra[i]<<scale].

Note

> Signed word indexes are expanded (unpacked) to doublewords before shifting left (2+scale) bits before transfer to GRPs for effective address calculation. This converts each index to an 64-bit offset.
>
> | processor | Latency | Throughput |
> |---|---|---|
> | power8 | 12-21 | 1/cycle |
> | power9 | 15-24 | 2/cycle |

Parameters

xs	Vector float elements to scatter store.
array	Pointer to array of float words.
vra	Vector of signed word (32-bit) indexes from array.
scale	8-bit integer. Indexes are multiplying by 2^{scale}.

7.5.4.34 vec_vsst4fswx()

```
static void vec_vsst4fswx (
            vf32_t xs,
            float * array,
            vi32_t vra )  [inline], [static]
```

Vector Scatter-Store 4 Float Singles to Vector Word Indexes.

For each float word element [0-3] of xs, store the float element xs[i] at ∗(char∗)array[vra[i]].

Note

> Signed word indexes are expanded (unpacked) to doublewords before shifting left 2 bits before transfer to GRPs for effective address calculation. This converts each index to an 64-bit offset.
>
> | processor | Latency | Throughput |
> |---|---|---|
> | power8 | 12-21 | 1/cycle |
> | power9 | 15-24 | 2/cycle |

Parameters

xs	Vector float elements to scatter store.
array	Pointer to array of float words.
vra	Vector of signed word (32-bit) indexes from array.

7.5.4.35 vec_vsstfsdo()

```
static void vec_vsstfsdo (
             vf64_t xs,
             float * array,
             vi64_t vra )  [inline], [static]
```

Vector Scatter-Store Floats Singles to Vector Doubleword Offsets.

For each doubleword element [0-1] of vra, store the doubleword float element xs[i], converted to float single word format, at ∗(char∗)array+vra[i].

processor	Latency	Throughput
power8	8	1/cycle
power9	9	2/cycle

Parameters

xs	Vector doubleword elements to scatter store as float single words.
array	Pointer to array of float words.
vra	Vector of doubleword (64-bit) byte offsets from &array.

7.5.4.36 vec_vsstfsdsx()

```
static void vec_vsstfsdsx (
             vf64_t xs,
             float * array,
             vi64_t vra,
             const unsigned char scale )  [inline], [static]
```

Vector Scatter-Store Words to Vector Doubleword Scaled Indexes.

For each doubleword element [0-1] of vra, store the doubleword float element xs[i], converted to float single word format, at array[vra[i]<<scale].

processor	Latency	Throughput
power8	10-19	1/cycle
power9	10-19	1/cycle

Parameters

xs	Vector doubleword elements to scatter store as float single words.
array	Pointer to array of float words.
vra	Vector of doubleword (64-bit) indexes from &array.

Parameters

scale	8-bit integer. Indexes are multiplying by 2^{scale}.

7.5.4.37 vec_vsstfsdx()

```
static void vec_vsstfsdx (
            vf64_t xs,
            float * array,
            vi64_t vra )  [inline], [static]
```

Vector Scatter-Store Words to Vector Doubleword Indexes.

For each doubleword element [0-1] of vra, store the doubleword float element xs[i], converted to float single word format, at array[vra[i]].

processor	Latency	Throughput
power8	10-19	1/cycle
power9	10-19	1/cycle

Parameters

xs	Vector doubleword elements to scatter store as float single words.
array	Pointer to array of float words.
vra	Vector of doubleword (64-bit) indexes from &array.

7.5.4.38 vec_vsstfsso()

```
static void vec_vsstfsso (
            vf64_t xs,
            float * array,
            const long long offset0,
            const long long offset1 )  [inline], [static]
```

Vector Scatter-Store Float Singles to Scalar Offsets.

For each scalar offset[0-1], Store the doubleword element xs[i], converted to float single word format, at *(char*)array+offset[0|1].

processor	Latency	Throughput
power8	3	1/cycle
power9	3	2/cycle

Parameters

xs	Vector doubleword elements to scatter store as float single words.
array	Pointer to array of float words.
offset0	Scalar (64-bit) byte offset from &array.
offset1	Scalar (64-bit) byte offset from &array.

7.5.4.39 vec_vstxsspx()

```
static void vec_vstxsspx (
             vf64_t xs,
             const signed long long ra,
             float * rb )  [inline], [static]
```

Vector Store Scalar Single Float Indexed.

Stores doubleword float element 0 of vector **xs** as a scalar float word at the effective address formed by **rb+ra**. The operand **rb** is a pointer to an array of float. The operand **ra** is a doubleword integer byte offset from **rb**. For best performance **rb** and **ra** should be word aligned (integer multiple of 4).

This operation is an alternate form of vector store element (vec_ste), with the added simplification that data is always left justified in the vector. Another advantage for Power8 and later, the stxsspx instruction can load directly into any of the 64 VSRs. Both simplify scatter operations.

Note

> The stxsspx instruction was introduced in PowerISA 2.07 (POWER8). Power7 and earlier will, move the source (xs) from VSR/VR range to VSR/FPR range if needed, then use stsf[x].

processor	Latency	Throughput
power8	0 - 2	2/cycle
power9	0 - 2	4/cycle

Parameters

xs	vector doubleword element 0 to be stored as single float.
ra	const doubleword index (offset/displacement).
rb	const pointer to an array of floats.

7.5.4.40 vec_xviexpsp()

```
static vf32_t vec_xviexpsp (
            vui32_t sig,
            vui32_t exp )  [inline], [static]
```

Vector Insert Exponent Single-Precision.

For each word of **sig** and **exp**, merge the sign (bit 0) and significand (bits 9:31) from **sig** with the 8-bit exponent from **exp** (bits 24:31). The exponent is merged into bits 1:8 of the final result. The result is returned as a Vector Single-Precision floating point value.

Note

> This operation is equivalent to the POWER9 xviexpsp instruction and the built-in vec_insert_exp. These require a POWER9-enabled compiler targeting -mcpu=power9 and are not available for older compilers nor POWER8 and earlier. This function provides this operation for all VSX-enabled platforms.

processor	Latency	Throughput
power8	6-15	2/cycle
power9	2	4/cycle

Parameters

sig	Vector unsigned int containing the Sign Bit and 23-bit significand.
exp	Vector unsigned int containing the 8-bit exponent.

Returns

> a vf32_t value where the exponent bits (1:8) of sig are replaced from bits 24:31 of exp.

7.5.4.41 vec_xvxexpsp()

```
static vui32_t vec_xvxexpsp (
            vf32_t vrb )  [inline], [static]
```

Vector Extract Exponent Single-Precision.

For each word of **vrb**, Extract the single-precision exponent (bits 1:8) and right justify it to (bits 24:31 of) of the result vector word. The result is returned as vector unsigned integer value.

Note

> This operation is equivalent to the POWER9 xvxexpsp instruction and the built-in vec_extract_exp. These require a POWER9-enabled compiler targeting -mcpu=power9 and are not available for older compilers nor POWER8 and earlier. This function provides this operation for all VSX-enabled platforms.

processor	Latency	Throughput
power8	6-15	2/cycle
power9	2	4/cycle

Parameters

vrb	vector double value.

Returns

vector unsigned int containing the 8-bit exponent right justified in each word

7.5.4.42 vec_xvxsigsp()

```
static vui32_t vec_xvxsigsp (
             vf32_t vrb )  [inline], [static]
```

Vector Extract Significand Single-Precision.

For each word of **vrb**, Extract the single-precision significand (bits 0:31) and restore the implied (hidden) bit (bit 8) if the single-precision value is normal (not zero, subnormal, Infinity or NaN). The result is return as vector unsigned int value with up to 24 bits of significance.

Note

This operation is equivalent to the POWER9 xvxsigsp instruction and the built-in vec_extract_sig. These require a POWER9-enabled compiler targeting -mcpu=power9 and are not available for older compilers nor POWER8 and earlier. This function provides this operation for all VSX-enabled platforms.

processor	Latency	Throughput
power8	8-17	1/cycle
power9	3	2/cycle

Parameters

vrb	vector double value.

Returns

vector unsigned int containing the significand.

7.6 src/pveclib/vec_f64_ppc.h File Reference

Header package containing a collection of 128-bit SIMD operations over 64-bit double-precision floating point elements.

```
#include <pveclib/vec_common_ppc.h>
#include <pveclib/vec_int128_ppc.h>
```

Functions

- static vf64_t vec_absf64 (vf64_t vf64x)

  Vector double absolute value.
- static int vec_all_isfinitef64 (vf64_t vf64)

  Return true if all 2x64-bit vector double values are Finite (Not NaN nor Inf).
- static int vec_all_isinff64 (vf64_t vf64)

  Return true if all 2x64-bit vector double values are infinity.
- static int vec_all_isnanf64 (vf64_t vf64)

  Return true if all 2x64-bit vector double values are NaN.
- static int vec_all_isnormalf64 (vf64_t vf64)

  Return true if all 2x64-bit vector double values are normal (Not NaN, Inf, denormal, or zero).
- static int vec_all_issubnormalf64 (vf64_t vf64)

  Return true if all 2x64-bit vector double values are subnormal (denormal).
- static int vec_all_iszerof64 (vf64_t vf64)

  Return true if all 2x64-bit vector double values are +-0.0.
- static int vec_any_isfinitef64 (vf64_t vf64)

  Return true if any of 2x64-bit vector double values are Finite (Not NaN nor Inf).
- static int vec_any_isinff64 (vf64_t vf64)

  Return true if any of 2x64-bit vector double values are infinity.
- static int vec_any_isnanf64 (vf64_t vf64)

  Return true if any of 2x64-bit vector double values are NaN.
- static int vec_any_isnormalf64 (vf64_t vf64)

  Return true if any of 2x64-bit vector double values are normal (Not NaN, Inf, denormal, or zero).
- static int vec_any_issubnormalf64 (vf64_t vf64)

  Return true if any of 2x64-bit vector double values is subnormal (denormal).
- static int vec_any_iszerof64 (vf64_t vf64)

  Return true if any of 2x64-bit vector double values are +-0.0.
- static vf64_t vec_copysignf64 (vf64_t vf64x, vf64_t vf64y)

  Copy the sign bit from vf64x merged with magnitude from vf64y and return the resulting vector double values.
- static vb64_t vec_isfinitef64 (vf64_t vf64)

  Return 2x64-bit vector boolean true values for each double element that is Finite (Not NaN nor Inf).
- static vb64_t vec_isinff64 (vf64_t vf64)

  Return 2x64-bit vector boolean true values for each double, if infinity.

- static vb64_t vec_isnanf64 (vf64_t vf64)

  Return 2x64-bit vector boolean true values, for each double NaN value.
- static vb64_t vec_isnormalf64 (vf64_t vf64)

  Return 2x64-bit vector boolean true values, for each double value, if normal (Not NaN, Inf, denormal, or zero).
- static vb64_t vec_issubnormalf64 (vf64_t vf64)

  Return 2x64-bit vector boolean true values, for each double value that is subnormal (denormal).
- static vb64_t vec_iszerof64 (vf64_t vf64)

  Return 2x64-bit vector boolean true values, for each double value that is +-0.0.
- static long double vec_pack_longdouble (vf64_t lval)

  Copy the pair of doubles from a vector to IBM long double.
- static vb64_t vec_setb_dp (vf64_t vra)

  Vector Set Bool from Sign, Double Precision.
- static vf64_t vec_unpack_longdouble (long double lval)

  Copy the pair of doubles from a IBM long double to a vector double.
- static vf64_t vec_vglfdso (double ∗array, const long long offset0, const long long offset1)

  Vector Gather-Load Float Double from scalar Offsets.
- static vf64_t vec_vglfddo (double ∗array, vi64_t vra)

  Vector Gather-Load Float Double from Doubleword Offsets.
- static vf64_t vec_vglfddsx (double ∗array, vi64_t vra, const unsigned char scale)

  Vector Gather-Load Float Double from Doubleword Scaled Indexes.
- static vf64_t vec_vglfddx (double ∗array, vi64_t vra)

  Vector Gather-Load Float Double from Doubleword indexes.
- static void vec_vsstfdso (vf64_t xs, double ∗array, const long long offset0, const long long offset1)

  Vector Scatter-Store Float Double to Scalar Offsets.
- static void vec_vsstfddo (vf64_t xs, double ∗array, vi64_t vra)

  Vector Scatter-Store Float Double to Doubleword Offsets.
- static void vec_vsstfddsx (vf64_t xs, double ∗array, vi64_t vra, const unsigned char scale)

  Vector Scatter-Store Float Double to Doubleword Scaled Index.
- static void vec_vsstfddx (vf64_t xs, double ∗array, vi64_t vra)

  Vector Scatter-Store Float Double to Doubleword Indexes.
- static vf64_t vec_vlxsfdx (const signed long long ra, const double ∗rb)

  Vector Load Scalar Float Double Indexed.
- static void vec_vstxsfdx (vf64_t xs, const signed long long ra, double ∗rb)

  Vector Store Scalar Float Double Indexed.
- static vf64_t vec_xviexpdp (vui64_t sig, vui64_t exp)

  Vector Insert Exponent Double-Precision.
- static vui64_t vec_xvxexpdp (vf64_t vrb)

  Vector Extract Exponent Double-Precision.
- static vui64_t vec_xvxsigdp (vf64_t vrb)

  Vector Extract Significand Double-Precision.

7.6.1 Detailed Description

Header package containing a collection of 128-bit SIMD operations over 64-bit double-precision floating point elements.

Many vector double-precision (64-bit float) operations are implemented with PowerISA-2.06 Vector Scalar Extended (VSX) (POWER7 and later) instructions. Most VSX instructions provide access to 64 combined scalar/vector registers. PowerISA-3.0 (POWER9) provides additional vector double operations: convert with round, convert to/from integer, insert/extract exponent and significand, and test data class. Most of these operations (compiler built-ins, or intrinsics) are defined in <altivec.h> and described in the `compiler documentation`.

Note

> The compiler disables associated <altivec.h> built-ins if the **mcpu** target does not enable the specific instruction. For example if you compile with **-mcpu=power8**, the double-precision vector converts, insert/extract and test data class built-ins are are not defined. This header provides the appropriate substitutions, will generate the minimum code, appropriate for the target, and produce correct results.
>
> Most ppc64le compilers will default to -mcpu=power8 if not specified.
>
> GCC 7.3 defines vector forms of the test data class, extract significand, and extract/insert_exp for float and double. These built-ins are not defined in GCC 6.4. See `compiler documentation`. These are useful operations and can be implemented in a few vector logical instructions for earlier machines.

So it is reasonable for this header to provide vector forms of the double-precision floating point classification functions (isnormal/subnormal/finite/inf/nan/zero, etc.). These functions can be implemented directly using (one or more) PO↩WER9 instructions, or a few vector logical and integer compare instructions for POWER7/8. Each is comfortably small enough to be in-lined and inherently faster than the equivalent POSIX or compiler built-in runtime scalar functions.

Most of these operations are implemented in a few instructions on newer (POWER7/POWER8/POWER9) processors. This header serves to fill in functional gaps for older (POWER7, POWER8) processors and provides an inline assembler implementation for older compilers that do not provide the built-ins.

This header covers operations that are any of the following:

- Implemented in hardware instructions in newer processors, but useful to programmers on slightly older processors (even if the equivalent function requires more instructions).
- Defined in the OpenPOWER ABI but *not* yet defined in <altivec.h> provided by available compilers in common use. Examples include vector double even/odd conversions.
- Providing special vector double tests for special conditions without generating extraneous floating-point exceptions. This is important for implementing vectorized forms of ISO C99 Math functions. Examples include vector double isnan, isinf, etc.
- Commonly used operations, not covered by the ABI or <altivec.h>, and require multiple instructions or are not obvious. For example, converts that change element size and imply converting two vectors into one vector of smaller elements, or one vector into two vectors of larger elements. Another example is the special case of packing/unpacking an IBM long double between a pair of floating-point registers (FPRs) and a single vector register (VR).

7.6.2 Examples

For example: using the the classification functions for implementing the math library function sine and cosine. The POSIX specification requires that special input values are processed without raising extraneous floating point exceptions and return specific floating point values in response. For example, the sin() function.

- If the input *value* is NaN then return a NaN.
- If the input *value* is +-0.0 then return *value*.
- If the input *value* is subnormal then return *value*.
- If the input *value* is +-Inf then return a quiet-NaN.
- Otherwise compute and return sin(value).

The following code example uses functions from this header to address the POSIX requirements for special values input to for a vectorized sinf():

```
vf64_t
test_vec_sinf64 (vf64_t value)
{
  const vf64_t vec_f0 = { 0.0, 0.0 };
  const vui64_t vec_f64_qnan =
    { 0x7ff8000000000000, 0x7ff8000000000000 };
  vf64_t result;
  vb64_t normmask, infmask;
  normmask = vec_isnormalf64 (value);
  if (vec_any_isnormalf64 (value))
    {
      // replace non-normal input values with safe values.
      vf64_t safeval = vec_sel (vec_f0, value, normmask);
      // body of vec_sin(safeval) computation elided for this example.
    }
  else
    result = value;
  // merge non-normal input values back into result
  result = vec_sel (value, result, normmask);
  // Inf input value elements return quiet-nan.
  infmask = vec_isinff64 (value);
  result = vec_sel (result, (vf64_t) vec_f64_qnan, infmask);
  return result;
}
```

The code generated for this fragment runs between 24 (-mcpu=power9) and 40 (-mcpu=power8) instructions. The normal execution path is 14 to 25 instructions respectively.

Another example the cos() function.

- If the input *value* is NaN then return a NaN.
- If the input *value* is +-0.0 then return *1.0*.
- If the input *value* is +-Inf then return a quiet-NaN.
- Otherwise compute and return cos(value).

The following code example uses functions from this header to address the POSIX requirements for special values input to vectorized cosf():

```
vf64_t
test_vec_cosf64 (vf64_t value)
{
  vf64_t result;
  const vf64_t vec_f0 = { 0.0, 0.0 };
  const vf64_t vec_f1 = { 1.0, 1.0 };
  const vui64_t vec_f64_qnan =
    { 0x7ff8000000000000, 0x7ff8000000000000 };
  vb64_t finitemask, infmask, zeromask;
  finitemask = vec_isfinitef64 (value);
  if (vec_any_isfinitef64 (value))
    {
      // replace non-finite input values with safe values.
      vf64_t safeval = vec_sel (vec_f0, value, finitemask);
      // body of vec_sin(safeval) computation elided for this example.
    }
  else
    result = value;
  // merge non-finite input values back into result
  result = vec_sel (value, result, finitemask);
  // Set +-0.0 input elements to exactly 1.0 in result.
  zeromask = vec_iszerof64 (value);
  result = vec_sel (result, vec_f1, zeromask);
  // Set Inf input elements to quiet-nan in result.
  infmask = vec_isinff64 (value);
  result = vec_sel (result, (vf64_t) vec_f64_qnan, infmask);
  return result;
}
```

Neither example raises floating point exceptions or sets **errno**, as appropriate for a vector math library.

7.6.3 Performance data.

High level performance estimates are provided as an aid to function selection when evaluating algorithms. For background on how *Latency* and *Throughput* are derived see: Performance data.

7.6.4 Function Documentation

7.6.4.1 vec_absf64()

```
static vf64_t vec_absf64 (
             vf64_t vf64x )  [inline], [static]
```

Vector double absolute value.

processor	Latency	Throughput
power8	6-7	2/cycle
power9	2	2/cycle

Parameters

vf64x	vector double values containing the magnitudes.

Returns

vector double absolute values of vf64x.

7.6.4.2 vec_all_isfinitef64()

```
static int vec_all_isfinitef64 (
             vf64_t vf64 )  [inline], [static]
```

Return true if all 2x64-bit vector double values are Finite (Not NaN nor Inf).

A IEEE Binary64 finite value has an exponent between 0x000 and 0x7fe (a 0x7ff indicates NaN or Inf). The significand can be any value. The sign bit is ignored.

processor	Latency	Throughput
power8	4-20	2/cycle
power9	6	1/cycle

Note

This function will not raise VXSNAN or VXVC (FE_INVALID) exceptions. A normal __binary64 compare can.

Parameters

vf64	a vector of __binary64 values.

Returns

an int containing 0 or 1.

7.6.4.3 vec_all_isinff64()

```
static int vec_all_isinff64 (
             vf64_t vf64 )  [inline], [static]
```

Return true if all 2x64-bit vector double values are infinity.

A IEEE Binary64 infinity has a exponent of 0x7ff and significand of all zeros. The sign bit is ignored.

Note

This function will not raise VXSNAN or VXVC (FE_INVALID) exceptions. A normal double compare can.

processor	Latency	Throughput
power8	6-20	2/cycle
power9	6	1/cycle

Parameters

vf64	a vector of __binary64 values.

Returns

boolean int, true if all 2 double values are infinity

7.6.4.4 vec_all_isnanf64()

```
static int vec_all_isnanf64 (
            vf64_t vf64 )  [inline], [static]
```

Return true if all 2x64-bit vector double values are NaN.

A IEEE Binary64 NaN value has an exponent between 0x7ff and the significand is nonzero. The sign bit is ignored.

Note

This function will not raise VXSNAN or VXVC (FE_INVALID) exceptions. A normal double compare can.

processor	Latency	Throughput
power8	6-20	2/cycle
power9	6	1/cycle

Parameters

vf64	a vector of __binary64 values.

Returns

a boolean int, true if all 2 vector double values are NaN.

7.6.4.5 vec_all_isnormalf64()

```
static int vec_all_isnormalf64 (
            vf64_t vf64 )  [inline], [static]
```

Return true if all 2x64-bit vector double values are normal (Not NaN, Inf, denormal, or zero).

A IEEE Binary64 normal value has an exponent between 0x001 and 0x7fe (a 0x7ff indicates NaN or Inf). The significand can be any value (expect 0 if the exponent is zero). The sign bit is ignored.

Note

This function will not raise VXSNAN or VXVC (FE_INVALID) exceptions. A normal double compare can.

processor	Latency	Throughput
power8	10-28	1/cycle
power9	6	1/cycle

Parameters

vf64	a vector of __binary64 values.

Returns

a boolean int, true if all 2 vector double values are normal.

7.6.4.6 vec_all_issubnormalf64()

```
static int vec_all_issubnormalf64 (
            vf64_t vf64 )  [inline], [static]
```

Return true if all 2x64-bit vector double values are subnormal (denormal).

A IEEE Binary64 subnormal has an exponent of 0x000 and a nonzero significand. The sign bit is ignored.

Note

This function will not raise VXSNAN or VXVC (FE_INVALID) exceptions. A normal double compare can.

processor	Latency	Throughput
power8	10-30	1/cycle
power9	6	1/cycle

Parameters

vf64	a vector of __binary64 values.

Returns

a boolean int, true if all of 2 vector double values are subnormal.

7.6.4.7 vec_all_iszerof64()

```
static int vec_all_iszerof64 (
            vf64_t vf64 )  [inline], [static]
```

Return true if all 2x64-bit vector double values are +-0.0.

A IEEE Binary64 zero has an exponent of 0x000 and a zero significand. The sign bit is ignored.

Note

This function will not raise VXSNAN or VXVC (FE_INVALID) exceptions. A normal double compare can.

processor	Latency	Throughput
power8	6-20	2/cycle
power9	6	1/cycle

Parameters

vf64	a vector of __binary64 values.

Returns

a boolean int, true if all 2 vector double values are +/- zero.

7.6.4.8 vec_any_isfinitef64()

```
static int vec_any_isfinitef64 (
            vf64_t vf64 )  [inline], [static]
```

Return true if any of 2x64-bit vector double values are Finite (Not NaN nor Inf).

A IEEE Binary64 finite value has an exponent between 0x000 and 0x7fe (a 0x7ff indicates NaN or Inf). The significand can be any value. The sign bit is ignored.

Note

This function will not raise VXSNAN or VXVC (FE_INVALID) exceptions. A normal double compare can.

processor	Latency	Throughput
power8	4-20	2/cycle
power9	6	1/cycle

Parameters

vf64	a vector of __binary64 values.

Returns

an int containing 0 or 1.

7.6.4.9 vec_any_isinff64()

```
static int vec_any_isinff64 (
            vf64_t vf64 )  [inline], [static]
```

Return true if any of 2x64-bit vector double values are infinity.

A IEEE Binary64 infinity has a exponent of 0x7ff and significand of all zeros.

Note

This function will not raise VXSNAN or VXVC (FE_INVALID) exceptions. A normal double compare can.

processor	Latency	Throughput
power8	6-20	2/cycle
power9	6	1/cycle

Parameters

vf64	a vector of __binary32 values.

Returns

boolean int, true if any of 2 double values are infinity

7.6.4.10 vec_any_isnanf64()

```
static int vec_any_isnanf64 (
            vf64_t vf64 )  [inline], [static]
```

Return true if any of 2x64-bit vector double values are NaN.

A IEEE Binary64 NaN value has an exponent between 0x7ff and the significand is nonzero. The sign bit is ignored.

Note

This function will not raise VXSNAN or VXVC (FE_INVALID) exceptions. A normal double compare can.

processor	Latency	Throughput
power8	6-20	2/cycle
power9	6	1/cycle

Parameters

vf64	a vector of __binary64 values.

Returns

a boolean int, true if any of 2 vector double values are NaN.

7.6.4.11 vec_any_isnormalf64()

```
static int vec_any_isnormalf64 (
            vf64_t vf64 )  [inline], [static]
```

Return true if any of 2x64-bit vector double values are normal (Not NaN, Inf, denormal, or zero).

A IEEE Binary64 normal value has an exponent between 0x001 and 0x7fe (a 0x7ff indicates NaN or Inf). The significand can be any value (expect 0 if the exponent is zero). The sign bit is ignored.

Note

This function will not raise VXSNAN or VXVC (FE_INVALID) exceptions. A normal double compare can.

processor	Latency	Throughput
power8	6-20	1/cycle
power9	6	1/cycle

Parameters

vf64	a vector of __binary64 values.

Returns

a boolean int, true if any of 2 vector double values are normal.

7.6.4.12 vec_any_issubnormalf64()

```
static int vec_any_issubnormalf64 (
             vf64_t vf64 )  [inline], [static]
```

Return true if any of 2x64-bit vector double values is subnormal (denormal).

A IEEE Binary64 subnormal has an exponent of 0x000 and a nonzero significand. The sign bit is ignored.

Note

This function will not raise VXSNAN or VXVC (FE_INVALID) exceptions. A normal double compare can.

processor	Latency	Throughput
power8	10-18	1/cycle
power9	6	1/cycle

Parameters

vf64	a vector of __binary64 values.

Returns

true if any of 2 vector double values are subnormal.

7.6.4.13 vec_any_iszerof64()

```
static int vec_any_iszerof64 (
             vf64_t vf64 )  [inline], [static]
```

Return true if any of 2x64-bit vector double values are +-0.0.

A IEEE Binary64 zero has an exponent of 0x000 and a zero significand. The sign bit is ignored.

Note

This function will not raise VXSNAN or VXVC (FE_INVALID) exceptions. A normal double compare can.

processor	Latency	Throughput
power8	6-20	2/cycle
power9	6	1/cycle

Parameters

vf64	a vector of __binary64 values.

Returns

a boolean int, true if any of 2 vector double values are +/- zero.

7.6.4.14 vec_copysignf64()

```
static vf64_t vec_copysignf64 (
            vf64_t vf64x,
            vf64_t vf64y )  [inline], [static]
```

Copy the sign bit from vf64x merged with magnitude from vf64y and return the resulting vector double values.

Note

This operation was patterned after the intrinsic vec_cpsgn (altivec.h) introduced for POWER7 and VSX. It turns out the original (GCC 4.9) compiler implementation reversed the operands and does not match the PowerISA or the Vector Intrinsic Programming Reference manuals. Subsequent compilers and PVECLIB implementations replicated this (operand order) error. This has now been reported as bug against the compilers, which are in the process of applying fixes and distributing updates. This version of PVECLIB is updated to match the Vector Intrinsic Programming Reference. This implementation is independent of the compilers update status.

processor	Latency	Throughput
power8	6-7	2/cycle
power9	2	2/cycle

Parameters

vf64x	vector double values containing the sign bits.
vf64y	vector double values containing the magnitudes.

Returns

vector double values with magnitude from vf64y and the sign of vf64x.

7.6.4.15 vec_isfinitef64()

```
static vb64_t vec_isfinitef64 (
             vf64_t vf64 )  [inline], [static]
```

Return 2x64-bit vector boolean true values for each double element that is Finite (Not NaN nor Inf).

A IEEE Binary64 finite value has an exponent between 0x000 and 0x7fe (a 0x7ff indicates NaN or Inf). The significand can be any value.

Using the vec_cmpeq conditional to generate the predicate mask for NaN / Inf and then invert this for the finite condition. The sign bit is ignored.

Note

This function will not raise VXSNAN or VXVC (FE_INVALID) exceptions. A normal double compare can.

processor	Latency	Throughput
power8	6-15	2/cycle
power9	5	2/cycle

Parameters

vf64	a vector of __binary64 values.

Returns

a vector boolean long, each containing all 0s(false) or 1s(true).

7.6.4.16 vec_isinff64()

```
static vb64_t vec_isinff64 (
             vf64_t vf64 )  [inline], [static]
```

Return 2x64-bit vector boolean true values for each double, if infinity.

A IEEE Binary64 infinity has a exponent of 0x7ff and significand of all zeros.

Note

This function will not raise VXSNAN or VXVC (FE_INVALID) exceptions. A normal double compare can.

processor	Latency	Throughput
power8	4-13	2/cycle
power9	3	2/cycle

Parameters

vf64	a vector of __binary64 values.

Returns

a vector boolean long long, each containing all 0s(false) or 1s(true).

7.6.4.17 vec_isnanf64()

```
static vb64_t vec_isnanf64 (
             vf64_t vf64 ) [inline], [static]
```

Return 2x64-bit vector boolean true values, for each double NaN value.

A IEEE Binary64 NaN value has an exponent between 0x7ff and the significand is nonzero. The sign bit is ignored.

processor	Latency	Throughput
power8	4-13	2/cycle
power9	3	2/cycle

Parameters

vf64	a vector of __binary64 values.

Returns

a vector boolean long long, each containing all 0s(false) or 1s(true).

7.6.4.18 vec_isnormalf64()

```
static vb64_t vec_isnormalf64 (
             vf64_t vf64 ) [inline], [static]
```

Return 2x64-bit vector boolean true values, for each double value, if normal (Not NaN, Inf, denormal, or zero).

A IEEE Binary64 normal value has an exponent between 0x001 and 0x7ffe (a 0x7ff indicates NaN or Inf). The significand can be any value (expect 0 if the exponent is zero).

Note

This function will not raise VXSNAN or VXVC (FE_INVALID) exceptions. A normal double compare can.

processor	Latency	Throughput
power8	6-15	1/cycle
power9	5	1/cycle

Parameters

vf64	a vector of __binary64 values.

Returns

a vector boolean long long, each containing all 0s(false) or 1s(true).

7.6.4.19 vec_issubnormalf64()

```
static vb64_t vec_issubnormalf64 (
              vf64_t vf64 )  [inline], [static]
```

Return 2x64-bit vector boolean true values, for each double value that is subnormal (denormal).

A IEEE Binary64 subnormal has an exponent of 0x000 and a nonzero significand. The sign bit is ignored.

Note

This function will not raise VXSNAN or VXVC (FE_INVALID) exceptions. A normal double compare can.

processor	Latency	Throughput
power8	6-16	1/cycle
power9	3	1/cycle

Parameters

vf64	a vector of __binary64 values.

Returns

a vector boolean long long, each containing all 0s(false) or 1s(true).

7.6.4.20 vec_iszerof64()

```
static vb64_t vec_iszerof64 (
             vf64_t vf64 )  [inline], [static]
```

Return 2x64-bit vector boolean true values, for each double value that is +-0.0.

A IEEE Binary64 zero has an exponent of 0x000 and a zero significand. The sign bit is ignored.

Note

This function will not raise VXSNAN or VXVC (FE_INVALID) exceptions. A normal double compare can.

processor	Latency	Throughput
power8	4-13	2/cycle
power9	3	2/cycle

Parameters

vf64	a vector of __binary32 values.

Returns

a vector boolean int, each containing all 0s(false) or 1s(true).

7.6.4.21 vec_pack_longdouble()

```
static long double vec_pack_longdouble (
             vf64_t lval )  [inline], [static]
```

Copy the pair of doubles from a vector to IBM long double.

Parameters

lval	vector double values containing the IBM long double.

Returns

IBM long double as FPR pair.

7.6.4.22 vec_setb_dp()

```
static vb64_t vec_setb_dp (
            vf64_t vra )  [inline], [static]
```

Vector Set Bool from Sign, Double Precision.

For each double, propagate the sign bit to all 64-bits of that doubleword. The result is vector bool long long reflecting the sign bit of each 64-bit double.

The resulting mask can be used in vector masking and select operations.

Note

This operation will set the sign mask regardless of data class, while the Vector Test Data Class instructions will not distinguish between +/- NaN.

processor	Latency	Throughput
power8	2-4	2/cycle
power9	2-5	2/cycle

Parameters

vra	Vector double.

Returns

vector bool long long reflecting the sign bits of each double value.

7.6.4.23 vec_unpack_longdouble()

```
static vf64_t vec_unpack_longdouble (
            long double lval )  [inline], [static]
```

Copy the pair of doubles from a IBM long double to a vector double.

Parameters

lval	IBM long double as FPR pair.

Returns

vector double values containing the IBM long double.

7.6.4.24 vec_vglfddo()

```
static vf64_t vec_vglfddo (
             double * array,
             vi64_t vra )  [inline], [static]
```

Vector Gather-Load Float Double from Doubleword Offsets.

For each doubleword element [i] of vra, load the float double element at ∗(char∗)array+vra[i]. Merge those float double elements and return the resulting vector.

Note

> As effective address calculation is modulo 64-bits, signed or unsigned doubleword offsets are equivalent.

processor	Latency	Throughput
power8	12	1/cycle
power9	11	1/cycle

Parameters

array	Pointer to array of doubles.
vra	Vector of doubleword (64-bit) byte offsets from &array.

Returns

> vector double containing elements loaded from ∗(char∗)array+vra[0] and ∗(char∗)array+vra[1].

7.6.4.25 vec_vglfddsx()

```
static vf64_t vec_vglfddsx (
             double * array,
             vi64_t vra,
             const unsigned char scale )  [inline], [static]
```

Vector Gather-Load Float Double from Doubleword Scaled Indexes.

For each doubleword element [i] of vra, load the float double element ∗array[vra[i] ∗ (1 << scale)]. Merge those float double elements and return the resulting vector. Indexes are converted to offsets from ∗array by shifting each doubleword left (3+scale) bits.

Note

> As effective address calculation is modulo 64-bits, signed or unsigned doubleword indexes are equivalent.

processor	Latency	Throughput
power8	14-23	1/cycle
power9	13-22	1/cycle

Parameters

array	Pointer to array of doubles.
vra	Vector of doubleword indexes.
scale	8-bit integer. Indexes are multiplied by 2^{scale}.

Returns

Vector double containing array[vra[0]∗(1<<scale)] and array[vra[1]∗(1<<scale)].

7.6.4.26 vec_vglfddx()

```
static vf64_t vec_vglfddx (
            double * array,
            vi64_t vra )  [inline], [static]
```

Vector Gather-Load Float Double from Doubleword indexes.

For each doubleword element [i] of vra, load the double element array[vra[i]]. Merge those float double elements and return the resulting vector. The indexes are converted to offsets from ∗array by shifting each doubleword index left 3-bits (∗8).

Note

As effective address calculation is modulo 64-bits, signed or unsigned doubleword indexes are equivalent.

processor	Latency	Throughput
power8	14-23	1/cycle
power9	13-22	1/cycle

Parameters

array	Pointer to array of doubles.
vra	Vector of doubleword indexes.

Returns

vector double containing {array[vra[0]], array[vra[1]]}.

7.6.4.27 vec_vglfdso()

```
static vf64_t vec_vglfdso (
             double * array,
             const long long offset0,
             const long long offset1 )  [inline], [static]
```

Vector Gather-Load Float Double from scalar Offsets.

For each scalar offset[0|1], load the float double element at ∗(char∗)array+offset[0|1]. Merge those float double elements and return the resulting vector.

processor	Latency	Throughput
power8	12	1/cycle
power9	11	1/cycle

Parameters

array	Pointer to array of doubles.
offset0	Scalar (64-bit) byte offsets from &array.
offset1	Scalar (64-bit) byte offsets from &array.

Returns

vector double containing elements loaded from ∗(char∗)array+offset0 and ∗(char∗)array+offset1.

7.6.4.28 vec_vlxsfdx()

```
static vf64_t vec_vlxsfdx (
             const signed long long ra,
             const double * rb )  [inline], [static]
```

Vector Load Scalar Float Double Indexed.

Load the left most doubleword of vector **xt** as a scalar double from the effective address formed by **rb+ra**. The operand **rb** is a pointer to an array of doubles. The operand **ra** is a doubleword integer byte offset from **rb**. The result **xt** is returned as a vf64_t vector. For best performance **rb** and **ra** should be doubleword aligned (integer multiple of 8).

Note

the right most doubleword of vector **xt** is left *undefined* by this operation.

This operation is an alternate form of Vector Load Element (vec_lde), with the added simplification that data is always left justified in the vector. This simplifies merging elements for gather operations.

Note

This is instruction was introduced in PowerISA 2.06 (POWER7). For POWER8/9 there are additional optimizations by effectively converting small constant index values into displacements. For POWER8 a specific pattern of addi/lsxdx instruction is *fused* into a single load displacement internal operation. For POWER9 we can use the lxsd (DS-form) instruction directly.

processor	Latency	Throughput
power8	5	2/cycle
power9	5	2/cycle

Parameters

ra	const doubleword index (offset/displacement).
rb	const doubleword pointer to an array of doubles.

Returns

The data stored at (ra + rb) is loaded into vector doubleword element 0. Element 1 is undefined.

7.6.4.29 vec_vsstfddo()

```
static void vec_vsstfddo (
            vf64_t xs,
            double * array,
            vi64_t vra )  [inline], [static]
```

Vector Scatter-Store Float Double to Doubleword Offsets.

For each doubleword element [i] of vra, Store the double element xs[i] at ∗(char∗)array+vra[i].

Note

As effective address calculation is modulo 64-bits, signed or unsigned doubleword offsets are equivalent.

processor	Latency	Throughput
power8	12	1/cycle
power9	8	1/cycle

Parameters

xs	Vector double elements to scatter store.
array	Pointer to array of doubles.
vra	Vector of doubleword (64-bit) byte offsets from &array.

7.6.4.30 vec_vsstfddsx()

```
static void vec_vsstfddsx (
             vf64_t xs,
             double * array,
             vi64_t vra,
             const unsigned char scale )  [inline], [static]
```

Vector Scatter-Store Float Double to Doubleword Scaled Index.

For each doubleword element [i] of vra, store the double element xs[i] at array[vra[i] ∗ (1 << scale)]. Indexes are converted to offsets from ∗array by shifting each doubleword of vra left (3+scale) bits.

Note

> As effective address calculation is modulo 64-bits, signed or unsigned doubleword indexes are equivalent.

processor	Latency	Throughput
power8	14-23	1/cycle
power9	10-19	1/cycle

Parameters

xs	Vector double elements to store.
array	Pointer to array of doubles.
vra	Vector of doubleword indexes.
scale	Factor effectually multiplying the indexes by 2^{scale}.

7.6.4.31 vec_vsstfddx()

```
static void vec_vsstfddx (
             vf64_t xs,
             double * array,
             vi64_t vra )  [inline], [static]
```

Vector Scatter-Store Float Double to Doubleword Indexes.

For each doubleword element [i] of vra, store the double element xs[i] at array[vra[i]]. Indexes are converted to offsets from ∗array by shifting each doubleword of vra left 3 bits.

Note

> As effective address calculation is modulo 64-bits, signed or unsigned doubleword indexes are equivalent.

processor	Latency	Throughput
power8	14-23	1/cycle
power9	10-19	1/cycle

Parameters

xs	Vector double elements to store.
array	Pointer to array of doubles.
vra	Vector of doubleword indexes.

7.6.4.32 vec_vsstfdso()

```
static void vec_vsstfdso (
            vf64_t xs,
            double * array,
            const long long offset0,
            const long long offset1 )  [inline], [static]
```

Vector Scatter-Store Float Double to Scalar Offsets.

For each doubleword element [i] of vra, Store the double element xs[i] at ∗(char∗)array+offset[0|1].

Note

As effective address calculation is modulo 64-bits, signed or unsigned doubleword offsets are equivalent.

processor	Latency	Throughput
power8	12	1/cycle
power9	8	1/cycle

Parameters

xs	Vector double elements to scatter store.
array	Pointer to array of doubles.
offset0	Scalar (64-bit) byte offset from &array.
offset1	Scalar (64-bit) byte offset from &array.

7.6.4.33 vec_vstxsfdx()

```
static void vec_vstxsfdx (
            vf64_t xs,
            const signed long long ra,
            double * rb )  [inline], [static]
```

Vector Store Scalar Float Double Indexed.

Stores the left most doubleword of vector **xs** as a scalar double float at the effective address formed by **rb+ra**. The operand **rb** is a pointer to an array of doubles. The operand **ra** is a doubleword integer byte offset from **rb**. For best performance **rb** and **ra** should be doubleword aligned (integer multiple of 8).

This operation is an alternate form of vector store element, with the added simplification that data is always left justified in the vector. This simplifies scatter operations.

Note

> This is instruction was introduced in PowerISA 2.06 (POWER7). For POWER9 there are additional optimizations by effectively converting small constant index values into displacements. For POWER9 we can use the stxsd (DS-form) instruction directly.

processor	Latency	Throughput
power8	0 - 2	2/cycle
power9	0 - 2	4/cycle

Parameters

xs	vector doubleword element 0 to be stored.
ra	const doubleword index (offset/displacement).
rb	const doubleword pointer to an array of doubles.

7.6.4.34 vec_xviexpdp()

```
static vf64_t vec_xviexpdp (
            vui64_t sig,
            vui64_t exp )  [inline], [static]
```

Vector Insert Exponent Double-Precision.

For each doubleword of **sig** and **exp**, merge the sign (bit 0) and significand (bits 12:63) from **sig** with the 11-bit exponent from **exp** (bits 53:63). The exponent is merged into bits 1:11 of the final result. The result is returned as a Vector Double-Precision floating point value.

Note

> This operation is equivalent to the POWER9 xviexpdp instruction and the built-in vec_insert_exp. These require a POWER9-enabled compiler targeting -mcpu=power9 and are not available for older compilers nor POWER8 and earlier. This function provides this operation for all VSX-enabled platforms.

processor	Latency	Throughput
power8	6-15	2/cycle
power9	2	4/cycle

Parameters

sig	Vector unsigned long long containing the Sign Bit and 52-bit significand.
exp	Vector unsigned long long containing the 11-bit exponent.

Returns

a vf64_t value where the exponent bits (1:11) of sig are replaced from bits 53:63 of exp.

7.6.4.35 vec_xvxexpdp()

```
static vui64_t vec_xvxexpdp (
             vf64_t vrb )  [inline], [static]
```

Vector Extract Exponent Double-Precision.

For each doubleword of **vrb**, Extract the double-precision exponent (bits 1:11) and right justify it to (bits 53:63 of) of the result vector doubleword. The result is returned as vector long long integer value.

Note

This operation is equivalent to the POWER9 xvxexpdp instruction and the built-in vec_extract_exp. These require a POWER9-enabled compiler targeting -mcpu=power9 and are not available for older compilers nor POWER8 and earlier. This function provides this operation for all VSX-enabled platforms.

processor	Latency	Throughput
power8	6-15	2/cycle
power9	2	4/cycle

Parameters

vrb	vector double value.

Returns

vector unsigned long long containing 11-bit exponent right justified in each doubleword

7.6.4.36 vec_xvxsigdp()

```
static vui64_t vec_xvxsigdp (
            vf64_t vrb )  [inline], [static]
```

Vector Extract Significand Double-Precision.

For each doubleword of **vrb**, Extract the double-precision significand (bits 12:63) and restore the implied (hidden) bit (bit 11) if the double-precision value is normal (not zero, subnormal, Infinity or NaN). The result is return as vector long long integer value with up to 53 bits of significance.

Note

> This operation is equivalent to the POWER9 xvxsigdp instruction and the built-in vec_extract_sig. These require a POWER9-enabled compiler targeting -mcpu=power9 and are not available for older compilers nor POWER8 and earlier. This function provides this operation for all VSX-enabled platforms.

processor	Latency	Throughput
power8	8-17	1/cycle
power9	3	2/cycle

Parameters

vrb	vector double value.

Returns

> vector unsigned long long containing the significand.

7.7 src/pveclib/vec_int128_ppc.h File Reference

Header package containing a collection of 128-bit computation functions implemented with PowerISA VMX and VSX instructions.

```
#include <pveclib/vec_common_ppc.h>
#include <pveclib/vec_int64_ppc.h>
```

Macros

- #define CONST_VUINT128_QxW(__q0, __q1, __q2, __q3)

  Generate a vector unsigned __int128 constant from words.
- #define CONST_VUINT128_QxD(__q0, __q1)

  Generate a vector unsigned __int128 constant from doublewords.
- #define CONST_VUINT128_Qx19d(__q0, __q1)

Generate a vector unsigned __int128 constant from doublewords.

- #define CONST_VUINT128_Qx18d(__q0, __q1)

  Generate a vector unsigned __int128 constant from doublewords.
- #define CONST_VUINT128_Qx16d(__q0, __q1)

  Generate a vector unsigned __int128 constant from doublewords.

Functions

- static vui128_t vec_absduq (vui128_t vra, vui128_t vrb)

  Vector Absolute Difference Unsigned Quadword.
- static vi128_t vec_abssq (vi128_t vra)

  Vector Absolute Value Signed Quadword.
- static vui128_t vec_avguq (vui128_t vra, vui128_t vrb)

  Vector Average Unsigned Quadword.
- static vui128_t vec_addcuq (vui128_t a, vui128_t b)

  Vector Add & write Carry Unsigned Quadword.
- static vui128_t vec_addecuq (vui128_t a, vui128_t b, vui128_t ci)

  Vector Add Extended & write Carry Unsigned Quadword.
- static vui128_t vec_addeuqm (vui128_t a, vui128_t b, vui128_t ci)

  Vector Add Extended Unsigned Quadword Modulo.
- static vui128_t vec_adduqm (vui128_t a, vui128_t b)

  Vector Add Unsigned Quadword Modulo.
- static vui128_t vec_addcq (vui128_t ∗cout, vui128_t a, vui128_t b)

  Vector Add with carry Unsigned Quadword.
- static vui128_t vec_addeq (vui128_t ∗cout, vui128_t a, vui128_t b, vui128_t ci)

  Vector Add Extend with carry Unsigned Quadword.
- static vui128_t vec_clzq (vui128_t vra)

  Vector Count Leading Zeros Quadword for unsigned __int128 elements.
- static vui128_t vec_ctzq (vui128_t vra)

  Vector Count Trailing Zeros Quadword for unsigned __int128 elements.
- static vb128_t vec_cmpeqsq (vi128_t vra, vi128_t vrb)

  Vector Compare Equal Signed Quadword.
- static vb128_t vec_cmpequq (vui128_t vra, vui128_t vrb)

  Vector Compare Equal Unsigned Quadword.
- static vb128_t vec_cmpgesq (vi128_t vra, vi128_t vrb)

  Vector Compare Greater Than or Equal Signed Quadword.
- static vb128_t vec_cmpgeuq (vui128_t vra, vui128_t vrb)

  Vector Compare Greater Than or Equal Unsigned Quadword.
- static vb128_t vec_cmpgtsq (vi128_t vra, vi128_t vrb)

  Vector Compare Greater Than Signed Quadword.
- static vb128_t vec_cmpgtuq (vui128_t vra, vui128_t vrb)

  Vector Compare Greater Than Unsigned Quadword.
- static vb128_t vec_cmplesq (vi128_t vra, vi128_t vrb)

  Vector Compare Less Than or Equal Signed Quadword.
- static vb128_t vec_cmpleuq (vui128_t vra, vui128_t vrb)

  Vector Compare Less Than or Equal Unsigned Quadword.

- static vb128_t vec_cmpltsq (vi128_t vra, vi128_t vrb)

  Vector Compare Less Than Signed Quadword.
- static vb128_t vec_cmpltuq (vui128_t vra, vui128_t vrb)

  Vector Compare Less Than Unsigned Quadword.
- static vb128_t vec_cmpnesq (vi128_t vra, vi128_t vrb)

  Vector Compare Equal Signed Quadword.
- static vb128_t vec_cmpneuq (vui128_t vra, vui128_t vrb)

  Vector Compare Not Equal Unsigned Quadword.
- static int vec_cmpsq_all_eq (vi128_t vra, vi128_t vrb)

  Vector Compare all Equal Signed Quadword.
- static int vec_cmpsq_all_ge (vi128_t vra, vi128_t vrb)

  Vector Compare any Greater Than or Equal Signed Quadword.
- static int vec_cmpsq_all_gt (vi128_t vra, vi128_t vrb)

  Vector Compare any Greater Than Signed Quadword.
- static int vec_cmpsq_all_le (vi128_t vra, vi128_t vrb)

  Vector Compare any Less Than or Equal Signed Quadword.
- static int vec_cmpsq_all_lt (vi128_t vra, vi128_t vrb)

  Vector Compare any Less Than Signed Quadword.
- static int vec_cmpsq_all_ne (vi128_t vra, vi128_t vrb)

  Vector Compare all Not Equal Signed Quadword.
- static int vec_cmpuq_all_eq (vui128_t vra, vui128_t vrb)

  Vector Compare all Equal Unsigned Quadword.
- static int vec_cmpuq_all_ge (vui128_t vra, vui128_t vrb)

  Vector Compare any Greater Than or Equal Unsigned Quadword.
- static int vec_cmpuq_all_gt (vui128_t vra, vui128_t vrb)

  Vector Compare any Greater Than Unsigned Quadword.
- static int vec_cmpuq_all_le (vui128_t vra, vui128_t vrb)

  Vector Compare any Less Than or Equal Unsigned Quadword.
- static int vec_cmpuq_all_lt (vui128_t vra, vui128_t vrb)

  Vector Compare any Less Than Unsigned Quadword.
- static int vec_cmpuq_all_ne (vui128_t vra, vui128_t vrb)

  Vector Compare all Not Equal Unsigned Quadword.
- static vui128_t vec_cmul10ecuq (vui128_t ∗cout, vui128_t a, vui128_t cin)

  Vector combined Multiply by 10 Extended & write Carry Unsigned Quadword.
- static vui128_t vec_cmul10cuq (vui128_t ∗cout, vui128_t a)

  Vector combined Multiply by 10 & write Carry Unsigned Quadword.
- static vi128_t vec_divsq_10e31 (vi128_t vra)

  Vector Divide by const 10e31 Signed Quadword.
- static vui128_t vec_divudq_10e31 (vui128_t ∗qh, vui128_t vra, vui128_t vrb)

  Vector Divide Unsigned Double Quadword by const 10e31.
- static vui128_t vec_divudq_10e32 (vui128_t ∗qh, vui128_t vra, vui128_t vrb)

  Vector Divide Unsigned Double Quadword by const 10e32.
- static vui128_t vec_divuq_10e31 (vui128_t vra)

  Vector Divide by const 10e31 Unsigned Quadword.
- static vui128_t vec_divuq_10e32 (vui128_t vra)

  Vector Divide by const 10e32 Unsigned Quadword.
- static vi128_t vec_maxsq (vi128_t vra, vi128_t vrb)

Vector Maximum Signed Quadword.
- static vui128_t vec_maxuq (vui128_t vra, vui128_t vrb)

  Vector Maximum Unsigned Quadword.
- static vi128_t vec_minsq (vi128_t vra, vi128_t vrb)

  Vector Minimum Signed Quadword.
- static vui128_t vec_minuq (vui128_t vra, vui128_t vrb)

  Vector Minimum Unsigned Quadword.
- static vi128_t vec_modsq_10e31 (vi128_t vra, vi128_t q)

  Vector Modulo by const 10e31 Signed Quadword.
- static vui128_t vec_modudq_10e31 (vui128_t vra, vui128_t vrb, vui128_t ∗ql)

  Vector Modulo Unsigned Double Quadword by const 10e31.
- static vui128_t vec_modudq_10e32 (vui128_t vra, vui128_t vrb, vui128_t ∗ql)

  Vector Modulo Unsigned Double Quadword by const 10e32.
- static vui128_t vec_moduq_10e31 (vui128_t vra, vui128_t q)

  Vector Modulo by const 10e31 Unsigned Quadword.
- static vui128_t vec_moduq_10e32 (vui128_t vra, vui128_t q)

  Vector Modulo by const 10e32 Unsigned Quadword.
- static vui128_t vec_mul10cuq (vui128_t a)

  Vector Multiply by 10 & write Carry Unsigned Quadword.
- static vui128_t vec_mul10ecuq (vui128_t a, vui128_t cin)

  Vector Multiply by 10 Extended & write Carry Unsigned Quadword.
- static vui128_t vec_mul10euq (vui128_t a, vui128_t cin)

  Vector Multiply by 10 Extended Unsigned Quadword.
- static vui128_t vec_mul10uq (vui128_t a)

  Vector Multiply by 10 Unsigned Quadword.
- static vui128_t vec_cmul100cuq (vui128_t ∗cout, vui128_t a)

  Vector combined Multiply by 100 & write Carry Unsigned Quadword.
- static vui128_t vec_cmul100ecuq (vui128_t ∗cout, vui128_t a, vui128_t cin)

  Vector combined Multiply by 100 Extended & write Carry Unsigned Quadword.
- static vui128_t vec_msumcud (vui64_t a, vui64_t b, vui128_t c)

  Vector Multiply-Sum and Write Carryout Unsigned Doubleword.
- static vui128_t vec_msumudm (vui64_t a, vui64_t b, vui128_t c)

  Vector Multiply-Sum Unsigned Doubleword Modulo.
- static vui128_t vec_muleud (vui64_t a, vui64_t b)

  Vector Multiply Even Unsigned Doublewords.
- static vui64_t vec_mulhud (vui64_t vra, vui64_t vrb)

  Vector Multiply High Unsigned Doubleword.
- static vui128_t vec_muloud (vui64_t a, vui64_t b)

  Vector Multiply Odd Unsigned Doublewords.
- static vui64_t vec_muludm (vui64_t vra, vui64_t vrb)

  Vector Multiply Unsigned Doubleword Modulo.
- static vui128_t vec_mulhuq (vui128_t a, vui128_t b)

  Vector Multiply High Unsigned Quadword.
- static vui128_t vec_mulluq (vui128_t a, vui128_t b)

  Vector Multiply Low Unsigned Quadword.
- static vui128_t vec_muludq (vui128_t ∗mulu, vui128_t a, vui128_t b)

  Vector Multiply Unsigned Double Quadword.

- static vui128_t vec_madduq (vui128_t ∗mulu, vui128_t a, vui128_t b, vui128_t c)

  Vector Multiply-Add Unsigned Quadword.
- static vui128_t vec_madd2uq (vui128_t ∗mulu, vui128_t a, vui128_t b, vui128_t c1, vui128_t c2)

  Vector Multiply-Add2 Unsigned Quadword.
- static vi128_t vec_negsq (vi128_t int128)

  Vector Negate Signed Quadword.
- static vui128_t vec_neguq (vui128_t int128)

  Vector Negate Unsigned Quadword.
- static vui128_t vec_popcntq (vui128_t vra)

  Vector Population Count Quadword for unsigned __int128 elements.
- static vui128_t vec_revbq (vui128_t vra)

  Vector Byte Reverse Quadword.
- static vui128_t vec_rlq (vui128_t vra, vui128_t vrb)

  Vector Rotate Left Quadword.
- static vui128_t vec_rlqi (vui128_t vra, const unsigned int shb)

  Vector Rotate Left Quadword Immediate.
- static vi128_t vec_selsq (vi128_t vra, vi128_t vrb, vb128_t vrc)

  Vector Select Signed Quadword.
- static vui128_t vec_seluq (vui128_t vra, vui128_t vrb, vb128_t vrc)

  Vector Select Unsigned Quadword.
- static vb128_t vec_setb_cyq (vui128_t vcy)

  Vector Set Bool from Quadword Carry.
- static vb128_t vec_setb_ncq (vui128_t vcy)

  Vector Set Bool from Quadword not Carry.
- static vb128_t vec_setb_sq (vi128_t vra)

  Vector Set Bool from Signed Quadword.
- static vui128_t vec_sldq (vui128_t vrw, vui128_t vrx, vui128_t vrb)

  Vector Shift Left Double Quadword.
- static vui128_t vec_sldqi (vui128_t vrw, vui128_t vrx, const unsigned int shb)

  Vector Shift Left Double Quadword Immediate.
- static vui128_t vec_slq (vui128_t vra, vui128_t vrb)

  Vector Shift Left Quadword.
- static vui128_t vec_slqi (vui128_t vra, const unsigned int shb)

  Vector Shift Left Quadword Immediate.
- static vi128_t vec_splat_s128 (const int sim)

  Vector Splat Immediate Signed Quadword. Extend a signed integer constant across the quadword element of the result. This is the quadword equivalent of Vector Splat Immediate Signed (Byte | Halfword |Word).
- static vui128_t vec_splat_u128 (const int sim)

  Vector Splat Immediate Unsigned Quadword. Extend a unsigned integer constant across the quadword element of the result. This is the quadword equivalent of Vector Splat Immediate Unsigned (Byte | Halfword |Word).
- static vi128_t vec_sraq (vi128_t vra, vui128_t vrb)

  Vector Shift Right Algebraic Quadword.
- static vi128_t vec_sraqi (vi128_t vra, const unsigned int shb)

  Vector Shift Right Algebraic Quadword Immediate.
- static vui128_t vec_srq (vui128_t vra, vui128_t vrb)

  Vector Shift Right Quadword.
- static vui128_t vec_srqi (vui128_t vra, const unsigned int shb)

Vector Shift Right Quadword Immediate.
- static vui128_t vec_slq4 (vui128_t vra)
- static vui128_t vec_slq5 (vui128_t vra)
- static vui128_t vec_srq4 (vui128_t vra)
- static vui128_t vec_srq5 (vui128_t vra)
- static vui128_t vec_subcuq (vui128_t vra, vui128_t vrb)

  Vector Subtract and Write Carry Unsigned Quadword.
- static vui128_t vec_subecuq (vui128_t vra, vui128_t vrb, vui128_t vrc)

  Vector Subtract Extended and Write Carry Unsigned Quadword.
- static vui128_t vec_subeuqm (vui128_t vra, vui128_t vrb, vui128_t vrc)

  Vector Subtract Extended Unsigned Quadword Modulo.
- static vui128_t vec_subuqm (vui128_t vra, vui128_t vrb)

  Vector Subtract Unsigned Quadword Modulo.
- static vui128_t vec_vmuleud (vui64_t a, vui64_t b)

  Vector Multiply Even Unsigned Doublewords.
- static vui128_t vec_vmaddeud (vui64_t a, vui64_t b, vui64_t c)

  Vector Multiply-Add Even Unsigned Doublewords.
- static vui128_t vec_vmadd2eud (vui64_t a, vui64_t b, vui64_t c, vui64_t d)

  Vector Multiply-Add2 Even Unsigned Doublewords.
- static vui128_t vec_vmuloud (vui64_t a, vui64_t b)

  Vector Multiply Odd Unsigned Doublewords.
- static vui128_t vec_vmaddoud (vui64_t a, vui64_t b, vui64_t c)

  Vector Multiply-Add Odd Unsigned Doublewords.
- static vui128_t vec_vmadd2oud (vui64_t a, vui64_t b, vui64_t c, vui64_t d)

  Vector Multiply-Add2 Odd Unsigned Doublewords.
- static vui128_t vec_vmsumeud (vui64_t a, vui64_t b, vui128_t c)

  Vector Multiply-Sum Even Unsigned Doublewords.
- static vui128_t vec_vmsumoud (vui64_t a, vui64_t b, vui128_t c)

  Vector Multiply-Sum Odd Unsigned Doublewords.
- static vui128_t vec_vsldbi (vui128_t vra, vui128_t vrb, const unsigned int shb)

  Vector Shift Left Double Quadword by Bit Immediate.
- static vui128_t vec_vsrdbi (vui128_t vra, vui128_t vrb, const unsigned int shb)

  Vector Shift Right Double Quadword by Bit Immediate.

7.7.1 Detailed Description

Header package containing a collection of 128-bit computation functions implemented with PowerISA VMX and VSX instructions.

Some of these operations are implemented in a single instruction on newer (POWER8/POWER9) processors. This header serves to fill in functional gaps for older (POWER7, POWER8) processors and provides a in-line assembler implementation for older compilers that do not provide the build-ins. Other operations do not exist as instructions on any current processor but are useful and should be provided. This header serves to provide these operations as inline functions using existing vector built-ins or other pveclib operations.

The original VMX (AKA Altivec) only defined a few instructions that operated on the 128-bit vector as a whole. This included the vector shift left/right (bit), vector shift left/right by octet (byte), vector shift left double by octet (select a contiguous 16-bytes from 2 concatenated vectors) 256-bit), and generalized vector permute (select any 16-bytes from 2 concatenated vectors). Use of these instructions can be complicated when;

- the shift amount is more than 8 bits,
- the shift amount is not a multiple of 8-bits (octet),
- the shift amount is a constant and needs to be generated/loaded before use.

These instructions can used in combination to provide generalized vector __int128 shift/rotate operations. Pveclib uses these operations to provide vector __int128 shift / rotate left, shift right and shift algebraic right operations. These operations require pre-conditions to avoid multiple instructions or require a combination of (bit and octet shift) instructions to get the quadword result. The compiler <altivec.h> built-ins only supports individual instructions. So using these operations quickly inspires a need for a header (like this) to contain implementations of the common operations.

The VSX facility (introduced with POWER7) did not add any integer doubleword (64-bit) or quadword (128-bit) operations. However it did add a useful doubleword permute immediate and word wise; merge, shift, and splat immediate operations. Otherwise vector __int128 (128-bit elements) operations have to be implemented using VMX word and halfword element integer operations for POWER7.

POWER8 added multiply word operations that produce the full doubleword product and full quadword add / subtract (with carry extend). The add quadword is useful to sum the partial products for a full 128 x 128-bit multiply. The add quadword write carry and extend forms, simplify extending arithmetic to 256-bits and beyond.

While POWER8 provided quadword integer add and subtract operations, it did not provide quadword Signed/Unsigned integer compare operations. It is possible to implement quadword compare operations using existing word / doubleword compares and the the new quadword subtract write-carry operation. The trick it so convert the carry into a vector bool __int128 via the vec_setb_ncq () operation. This header provides easy to use quadword compare operations.

POWER9 (PowerISA 3.0B) adds the **Vector Multiply-Sum unsigned Doubleword Modulo** instruction. Aspects of this instruction mean it needs to be used carefully as part of larger quadword multiply. It performs only two of the four required doubleword multiplies. The final quadword modulo sum will discard any overflow/carry from the potential 130-bit result. With careful pre-conditioning of doubleword inputs the results are can not overflow from 128-bits. Then separate add quadword add/write carry operations can be used to complete the sum of partial products. These techniques are used in the POWER9 specific implementations of vec_muleud, vec_muloud, vec_mulluq, and vec_muludq.

PowerISA 3.0B also defined additional: Binary Coded Decimal (BCD) and Zoned character format conversions. String processing operations. Vector Parity operations. Integer Extend Sign Operations. Integer Absolute Difference Operations. All of these seem to useful additions to pveclib for older (POWER7/8) processors and across element sizes (including quadword elements).

Most of these intrinsic (compiler built-in) operations are defined in <altivec.h> and described in the compiler documentation. However it took several compiler releases for all the new POWER8 64-bit and 128-bit integer vector intrinsics to be added to **altivec.h**. This support started with the GCC 4.9 but was not complete across function/type and bug free until GCC 6.0.

Note

> The compiler disables associated <altivec.h> built-ins if the **mcpu** target does not enable the specific instruction. For example, if you compile with **-mcpu=power7**, vec_vadduqm and vec_vsubudm will not be defined. But vec_adduqm() and vec_subudm() and always be defined in this header, will generate the minimum code, appropriate for the target, and produce correct results.

Most of these operations are implemented in a single instruction on newer (POWER8/POWER9) processors. So this header serves to fill in functional gaps for older (POWER7, POWER8) processors and provides a in-line assembler implementation for older compilers that do not provide the build-ins.

This header covers operations that are either:

- Operations implemented in hardware instructions for later processors and useful to programmers, on slightly older processors, even if the equivalent function requires more instructions. Examples include quadword byte reverse, add and subtract.
- Defined in the OpenPOWER ABI but *not* yet defined in <altivec.n> provided by available compilers in common use. Examples include quadword byte reverse, add and subtract.
- Are commonly used operations, not covered by the ABI or <altivec.h>, and require multiple instructions or are not obvious. Examples include quadword; Signed and Unsigned compare, shift immediate, multiply, multiply by 10 immediate, count leading zeros and population count.

Note

> The Multiply sum/even/odd doubleword operations are currently implemented here (in <vec_int128_ppc.h>) which resolves a dependency on Add Quadword. These functions (vec_msumudm, vec_muleud, vec_muloud) all produce a quadword results and may use the vec_adduqm implementation to sum partial products.

See Returning extended quadword results. for more background on extended quadword computation.

7.7.2 Endian problems with quadword implementations

Technically operations on quadword elements should not require any endian specific transformation. There is only one element so there can be no confusion about element numbering or order. However some of the more complex quadword operations are constructed from operations on smaller elements. And those operations as provided by <altivec.h> are required by the OpenPOWER ABI to be endian sensitive. See Endian problems with doubleword operations for a more detailed discussion.

In any case the arithmetic (high to low) order of bits in a quadword are defined in the PowerISA (See vec_adduqm() and vec_subuqm()). So pveclib implementations will need to either:

- Nullify little endian transforms of <altivec.h> operations. The <altivec.h> built-ins vec_muleuw(), vec_mulouw(), vec_mergel(), and vec_mergeh() are endian sensitive and often require nullification that restores the original operation.
- Use new operations that are specifically defined to be stable across BE/LE implementations. The pveclib operations; vec_vmuleud() vec_vmuloud(), vec_mrgahd(), vec_mrgald(). and vec_permdi() are defined to be endian stable.

7.7.2.1 Quadword Integer Constants

The compilers may not support 128-bit integers for constants and printf (integer to ascii). For example GCC provides ANSI mandated constant and runtime support for integers up to long long which for PowerPC is only 64-bit.

The __int128 type is an extension that provides basic arithmetic operations but does not compile 128-bit constants or support printf formating for integers larger then long long. The following section provides examples and work around's for these restrictions.

The GCC compiler allows integer constants to be assigned/cast to __int128 types. The support also allows __int128 constants to be assigned/cast to vector __int128 types. So the following are allowed:

```
const vui128_t vec128_zeros = {(vui128_t) ((unsigned __int128) 0)};
const vui128_t vec128_10 = {(vui128_t) ((unsigned __int128) 10)};
const vui128_t vec128_10to16 = {(vui128_t) ((unsigned __int128)
                                10000000000000000UL)};
const vui128_t vec128_maxLong = {(vui128_t) ((unsigned __int128)
                                __INT64_MAX__)};
const vui128_t vec128_max_Long = {(vui128_t) ((unsigned __int128)
                                0x7fffffffffffffffL)};
// -1 signed extended to __int128 is 0xFFFF...FFFF
const vui128_t vec128_foxes = {(vui128_t) ((__int128) -1L)};
```

It gets more complicated when the constant exceeds the range of a long long value. For example the magic numbers for the multiplicative inverse described in Printing Vector __int128 values. The decimal integer constant we need for the quadword multiplier is "76624777043294442917917351357515459181" or the equivalent hexadecimal value "0x39a5652fb1137856d30baf9a1e626a6d". GCC does not allow constants this large to be expressed directly.

GCC supports aggregate initializer lists for the elements of vectors. For example:

```
vui32_t xyzw = (vector int) { 1, 2, 3, 4 };
```

So it is possible to compose a quadword constant by initializing a vector of word or doubleword elements then casting the result to a quadword type. For example:

```
const vui128_t invmul = (vui128_t) (vector unsigned long long)
                  { 0x39a5652fb1137856UL, 0xd30baf9a1e626a6dUL };
```

or

```
const vui128_t invmul = (vui128_t) (vector unsigned int)
                  { 0x39a5652f, 0xb1137856, 0xd30baf9a, 0x1e626a6d };
```

There is one small problem with this as element order is endian dependent, while a vector quadword integer is always big endian. So we would need to adjust the element order for endian. For example:

```
   const vui128_t invmul = (vui128_t) (vector unsigned long long)
#if __BYTE_ORDER__ == __ORDER_LITTLE_ENDIAN__
                     { 0xd30baf9a1e626a6dUL, 0x39a5652fb1137856UL };
#else
                     { 0x39a5652fb1137856UL, 0xd30baf9a1e626a6dUL };
#endif
```

or

```
   const vui128_t invmul = (vui128_t) (vector unsigned int)
#if __BYTE_ORDER__ == __ORDER_LITTLE_ENDIAN__
                     { 0x1e626a6d, 0xd30baf9a, 0xb1137856, 0x39a5652f };
#else
                     { 0x39a5652f, 0xb1137856, 0xd30baf9a, 0x1e626a6d };
#endif
```

Remembering to add the endian correction for constants used quadword operations is an issue and manually reversing the element order can be error prone. There should be an easier way.

7.7.2.2 Support for Quadword Integer Constants

The vec_common_ppc.h header provides some helper macros for when quadword operations need big endian element order on little endian platforms. These macros accept 2, 4, 8, or 16 element constants to form an aggregate initializer for a vector of the corresponding element type. The elements are always arranged left to right, high to low order. These macros are endian sensitive and either effectively pass-through for big endian or reverse the element order for little endian.

For example:

```
const vui128_t mul_invs_ten16 = (vui128_t) CONST_VINT128_DW(
   0x39a5652fb1137856UL, 0xd30baf9a1e626a6dUL);
```

or

```
const vui128_t mul_invs_ten16 = (vui128_t) CONST_VINT128_W(
                  0x39a5652f, 0xb1137856, 0xd30baf9a, 0x1e626a6d);
```

These macros internally cast to a vector unsigned integer type for the aggregate initializer. This type corresponds to the size and number of elements to fit in a 128-bit vector. This tells the compiler how many elements to expect and the allowed value range for the initializer. A final explicit cast is required to the vector type needed (usually a signed or unsigned __int128). (See: CONST_VINT128_DW(), CONST_VINT128_W(), CONST_VINT128_H(), CONST_VINT128_B()). Other macros require the programmer to provide a cast to match the element count and size. (See: CONST_VINT64_DW(), CONST_VINT32_W(), CONST_VINT16_H(), CONST_VINT8_B())

The methods above are effectively forming multi-digit constants where each digit is itself a large (word or doubleword) binary coded integer value. Because the digits are radix 2∗∗N it is normal to convert large decimal constants to hexadecimal. This makes it easier to split the large constants into word or doubleword elements for the initializer.

Most compilers support compile time computation on constants. This is an optimization where only the final computed constant result is used in the generated code. Compile time constant computation supports the usual arithmetic operations on the usual types. Some compilers (including GCC) support constant computation on extended types including __int128.

For example:

```
const vui128_t ten32_minus1 = (vui128_t)
      (((unsigned __int128) 9999999999999999UL) * 10000000000000000UL)
     + ((unsigned __int128) 9999999999999999UL);
```

produces the quadword integer value for the decimal constant 99999999999999999999999999999999.

Note

> we must cast any int or long long constants to [unsigned] __int128 so the compiler will use 128-bits arithmetic to compute the final constant.

With this technique we can split large decimal constants into 16, 18, or 19 digit blocks and then compute effective 32, 36, or 38 digit constant. (see CONST_VUINT128_Qx16d(), CONST_VUINT128_Qx18d(), and CONST_VUINT128_Qx19d()). For example:

```
  const vui128_t ten32_minus1 = CONST_VUINT128_Qx16d
        ( 9999999999999999UL, 9999999999999999UL );
// The quadword multiplicative inverse to divide by 10**16
// is 76624777043294442917917351357515459181.
// Which is 38 digits, so we split into 2 consts of 19 digits each.
  const vui128_t mul_invs_ten16 = CONST_VUINT128_Qx19d(
     7662477704329444291UL, 7917351357515459181UL);
```

7.7.2.3 Loading small Quadword constants

Programming with quadword integers will need quadword constants for masking and arithmetic operations. In the sections above we provide means to define large and complex constants. But often there is need for small integer constants for use in boolean logic, masking/select operations, and simple arithmetic.

The technique above can used for small integer constants as well. For example:

```
const vui128_t qw_one = CONST_VINT128_DW(0, 1);
const vui128_t qw_ten = CONST_VINT128_DW(0, 10);
const vui128_t qw_digit_mask = CONST_VINT128_DW(0, 0xf);
```

In most cases this compiler will allocate these constant values to the read-only data (.rodata) section. When these constants are referenced in programming operations the compiler generates the appropriate vector loads. For example the GCC V11 generates the following for the **-mcpu=power8** target:

```
addis   r9,r2,.rodata.cst16+0x30@toc@ha
addi    r9,r9,.rodata.cst16+0x30@toc@l
lvx     v2,0,r9
```

And the following for the **-mcpu=power9** target:

```
addis   r9,r2,.rodata.cst16+0x30@toc@ha
addi    r9,r9,.rodata.cst16+0x30@toc@l
lxv     v2,0(r9)
```

This is expected for POWER8 as PowerISA 2.07B does not have any displacement form (D-Form) vector (VSX) loads/stores instructions. The compiler allocates constants to the .rodata sections and the linker collects .rodata from object files into a combined executable .rodata section. This is placed near the *Table of Contents (**TOC**)* section. The ABI dedicates R2 as the base address **.TOC.** for the TOC and adjacent sections.

The *Add Immediate Shifted (addis) Add Immediate (addi)* sequence above computes a signed 32-bit **.TOC.** relative offset to a specific .rodata quadword. Two instructions are required as; *addis* provides the *high adjusted* 16-bits shifted left 16-bits, while *addi* provides the *low* 16-bits. The sum of R2 and these immediate values is the 64-bit effective address of a .rodata constant value. A signed 32-bit offset is large enough to support most program and library executables.

The load itself has a 5-cycle latency assuming a L1 cache hit. The three instruction sequence is sequentially dependent and requires 9-cycles latency (minimum) to execute. A L1 cache miss will increase the latency by 7-28 cycles, assuming the data resides in the L2/L3 caches.

However the compiler is not following the recommendations of

PowerISA 2.07B, *Book II, Chapter 2.1 Performance-Optimized Instruction Sequences*. This chapter recommends a specific pattern for the addi/lvx sequence. For example:

```
addis   rA,r2,.rodata.cst16+0x30@toc@ha
addi    rx,0,.rodata.cst16+0x30@toc@l
lvx     v2,rA,rx
```

In this case rx can be any GPR (including r0) while RA must be a valid base (r1 <-> r31) register.

The POWER8 implementation allows for *Instruction Fusion* combining information from two *adjacent*t instructions into one (internal) instruction so that it executes faster than the non-fused case. Effectively the addi/lvx combination above becomes a D-Form load vector instruction.

There are additional restrictions on the definition of *adjacent*:

- The instruction must be in the same dispatch group.
  - In single-threaded mode, up to six non-branch and up to two branch instructions (6/2 groups).
  - In multi-threaded mode, up to three non-branch and up to one branch instructions (3/1 groups).
- Without any intervening branch instructions.
- Instructions may span an I-cache line, but with both fetched and residing in the i-buffer.

This can reduce the latency from 9 to 7-cycles. This would be true even without *Instruction Funsion* as the addis/addi instructions are now independent and can execute in parallel.

The sequence generated for POWER9 is even more disappointing. The lxv is a D-Form (DQ) instruction and the displacement operand could be used to replace the addi instruction. For example: **-mcpu=power9** target:

```
addis   r9,r2,.rodata.cst16+0x30@toc@ha
lxv     v2,.rodata.cst16+0x30@toc@l(r9)
```

This provides the equivalent 32-bit TOC relative displacement with one less instructions and reduced latency of 7-cycles.

7.7.2.3.1 Alternatives to loading from .rodata

This is all a little cumbersome and it seems like there should be a better/faster way. Any instruction sequence that loads quadword integer constant in:

- three instruction or less,
- latency of 6 cycles or less,
- and avoids cache misses

is a good deal.

The base (Altivec) vector ISA included Vector Splat Immediate Signed Byte/Halfword/Word instructions. These are fast (2-cycle latency) and convenient for small integer constants in the range -16 to 15. So far the ISA has not added doubleword or quadword forms for these.

POWER9 added a VSX Vector Splat Immediate Byte (xxspltib) instruction. This expands the immediate range to -128 to 127 but does not include larger element sizes. POWER9 does provide Vector Extend Sign Byte To Word/Doubleword (vextsb2w/vextsb2d) instructions. For example the two instruction sequence:

```
xxspltib vs34,127
vextsb2d v2,v2
```

can generate a doubleword splat immediate for integers in the range -128 to 127 with a cycle latency of 5-cycles. So far there is no extend sign byte/halfword/word to quadword. POWER10 does add Vector Extend Sign Doubleword To Quadword (vextsd2q).

Note

> POWER10 does add the interesting *VSX Vector Splat Immediate Double-Precision* instruction. This is a 64-bit instruction with a 32-bit single precision immediate operand. Interesting but not helpful for quadword integer.

7.7.2.3.2 Some special quadword constants

The GCC compiler does recognize some vector constants as special case. For example:

```
vi128_t
__test_splatisq_n1_V0 (void)
{
  const vui32_t q_ones = {-1, -1, -1, -1};
  return (vi128_t) q_ones;
}
vi128_t
__test_splatisq_0_V0 (void)
{
  const vui32_t q_zero = {0, 0, 0, 0};
  return (vi128_t) q_zero;
}
```

will generate:

```
0000000000000080 <__test_splatisq_n1_V0>:
     vspltisw v2,-1
     blr
00000000000000a0 <__test_splatisq_0_V0>:
     vspltisw v2,0
     blr
```

Another interesting example is the quadword sign mask. For example:

```
vui32_t
__test_splatisq_signmask_V0 (void)
{
  const vui32_t signmask = CONST_VINT128_W(0x80000000, 0, 0, 0);
  return signmask;
}
```

will generate:

```
00000000000000c0 <__test_splatisq_signmask_V0>:
     vspltisw v0,-1
     vspltisw v2,0
     vslw    v0,v0,v0
     vsldoi  v2,v0,v2,12
     blr
```

The first 2 instructions generate vector constants of *all zeros* and *all ones* (same as above). The third instruction uses vector shift left word (vslw) to convert the word elements from 0xffffffff to 0x80000000.

The cleaver bit is shifting elements of the *all ones* (0xffffffff or -1) vector, left by 31-bits (0b11111), which is the value of low order 5-bits of the *all ones* element. Fortunately the **vsl[bhw]** instructions ignores all but the lower order bits needed for the element shift count.

Note

> This applies for element sizes byte, halfword and word. It also applies to doubleword elements on POWER8/9 using **vsld** but the compiler does not the recognize this case. And with POWER10 this technique applies to quadwords using **vslq**.

To convert a word sign mask to a quadword sign mask we need the *all zeros* vector and one additional instruction. The Vector Shift Left Double by Octet Immediate (**vsldoi**) rotates the low-order signmask word element to the high order word with 3 words of '0' concatenated on the right.

The equivalent C language with <altivec.h> intrinsics implementation is:

```
static inline vui32_t
vec_mask128_f128sign (void)
{
  const vui32_t q_zero = {0, 0, 0, 0};
  const vui32_t q_ones = {-1, -1, -1, -1};
  vui32_t signmask;
  signmask = vec_sl (q_ones, q_ones);
  return vec_sld (signmask, q_zero, 12);
}
```

This sequence is a little bigger (4 instructions) then we would like but should execute in 6-cycles. The first two instructions are independent and should execute in parallel. Also (as we will see) the all zero/ones constants are common building blocks. So the compiler should treat these as common sub expressions with across all operations using those constants.

7.7.2.3.3 Defining our own vec_splat_s128 So the compiler can do clever things with vector constants. But so far these are the only examples I have found. Other cases that you might expect to be a special case are not. For example:

```
vi128_t
__test_splatisq_15_V1 (void)
{
  const vui128_t qw_15 = {15};
  return (vi128_t) qw_15;
}
```

and

```
vi128_t
__test_splatisq_15_V0 (void)
{
  const vui32_t qw_15 = CONST_VINT128_W(0, 0, 0, 15);
  return (vi128_t) qw_15;
}
```

generate the 3 instruction (9-cycle) load from .rodata sequence. also constants using the vector long long or __int128 types may fail to compile on older versions of the compiler.

Note

> PVECLIB has found it best to consistently use vector unsigned int (**vui32_t**) internally for these operations. First older compiles may fail to compile specific combinations of vector long long or __int128 types and <altivec.h> intrinsics. Second the compiler may consider the vector long long constants as not quadword aligned and generate lxvd2x/xxswapd instead of lvx.

We can generate small constants in the range 1-15 with using the following pattern:

```
vi128_t
__test_splatisq_15_V2 (void)
{
  //  const vui32_t qw_15 = CONST_VINT128_W(0, 0, 0, 15);
  const vui32_t q_zero = CONST_VINT128_W (0, 0, 0, 0);
  vui32_t qw_15 = (vui32_t) vec_splat_s32(15);
  return (vi128_t) vec_sld (q_zero, qw_15, 4);
}
```

Which generates:

```
00000000000000e0 <__test_splatisq_15_V2>:
     vspltisw v0,0
     vspltisw v2,15
     vsldoi  v2,v0,v2,4
     blr
```

Here we use the vec_splat_s32() intrinsic to generate the vspltisw instruction for the value 15.

This sequence is only 3 instructions, which should execute in 4-cycles. The first two instructions are independent and should execute in parallel. Also the q_zero constant is commonly used and the compiler should treat it as a common sub expressions.

For small (-16 to -1) negative constants we need to make one small change. We use the q_ones constant to propagate the sign across the quadword.

```
vi128_t
__test_splatisq_n16_V2 (void)
{
  //  const vui32_t qw_16 = CONST_VINT128_W(-1, -1, -1, -16);
  const vui32_t q_ones = {-1, -1, -1, -1};
  vui32_t qw_16 = (vui32_t) vec_splat_s32(-16);
  return (vi128_t) vec_sld (q_ones, qw_16, 4);
}
```

The generated sequence is also 3 instructions and should execute in 4-cycles.

Putting this all together we can create a static inline function to generate small quadword constants (in the range -16 to 15). For example:

```
static inline vi128_t
vec_splat_s128_PWR8 (const int sim)
{
  vi128_t result;
  if (__builtin_constant_p (sim) && ((sim >= -16) && (sim < 16)))
    {
      vui32_t vwi = (vui32_t) vec_splat_s32(sim);
      if (__builtin_constant_p (sim) && ((sim == 0) || (sim == -1)))
        {
          // Special case for -1 and 0. Skip vec_sld().
          result = (vi128_t) vwi;
        }
      else
        {
          if (__builtin_constant_p (sim) && (sim > 0))
            {
              const vui32_t q_zero = {0, 0, 0, 0};
              result = (vi128_t) vec_sld (q_zero, vwi, 4);
            }
          else
            {
              const vui32_t q_ones = {-1, -1, -1, -1};
              result = (vi128_t) vec_sld (q_ones, vwi, 4);
            }
        }
    }
  else
    result = vec_splats ((signed __int128) sim);
  return (result);
}
```

This version uses only <altivec.h> intrinsics supported by POWER8 and earlier. For constants in the range (-16 to 15) the range is divided into three groups:

- Special values -1 and 0 that can be gnerated in a single instruction.
- Values 1 to 15 that require the q_zero constant to sign extend.
- Values -16 to -2 that require the q_ones constant to sign extend.

Values outside this range use the vec_splats() intrinsic which will generate the appropriate quadword constant in .rodata and the load sequence to retrieve that value.

For POWER9 and later we can use the VSX Vector Splat Immediate Byte (xxspltib) instruction and support the extended constant range of -128 to 127.

```
static inline vi128_t
vec_splat_s128_PWR9 (const int sim)
{
  vi128_t result;
  if (__builtin_constant_p (sim) && ((sim >= -128) && (sim < 128)))
    {
      // Expect the compiler to generate a single xxspltib for this.
      vi8_t vbi = vec_splats ((signed char) sim);
      if (__builtin_constant_p (sim) && ((sim == 0) || (sim == -1)))
        {
          // Special case for -1 and 0. Skip vec_sld().
          result = (vi128_t) vbi;
        }
      else
        {
          if (__builtin_constant_p (sim) && (sim > 0))
            {
              const vui32_t q_zero = {0, 0, 0, 0};
              result = (vi128_t) vec_sld ((vi8_t) q_zero, vbi, 1);
            }
          else
            {
              const vui32_t q_ones = {-1, -1, -1, -1};
              result = (vi128_t) vec_sld ((vi8_t) q_ones, vbi, 1);
            }
        }
    }
  else
    result = vec_splats ((signed __int128) sim);
  return (result);
}
```

Here we use the vec_splats() intrinsic to generate the xxspltib instruction. The rest follows the pattern we used for POWER8 but shift left is adjusted for the byte (vs word) element splat to be 1 octet.

7.7.3 Some facts about fixed precision integers

The transition from grade school math to computer programming requires the realization that computers handle numbers in fixed sized chunks. For the PowerISA these chunks are byte, halfword, word, doubleword, and quadword. While computer languages like "C" have integer types like char, short, int, long int, and __int128.

Happily these chunks are large enough to hold the equivalent of several decimal digits and handle most of the grotty details of multiply, divide, add, and subtract. But sometimes the chunk (used) is not large enough to hold all the digits you need. Sums may overflow and multiplies may be truncated (modulo the chunk size).

Sometimes we can simply switch to the next larger size (int to long, word to doubleword) and avoid the problem (overflow of sums or truncation of multiply). But sometimes the largest chunk the compiler or hardware supports is still not large enough for the numbers we are dealing with. This requires *multiple precision arithmetic* with works a lot like grade school arithmetic but with larger digits represented by the most convenient computer sized chunk.

Most programmers would prefer to use an existing *multiple precision arithmetic* library and move on. Existing libraries are implemented with scalar instructions and loops over storage arrays. But here we need to provide vector quadword multiply and extended quadword add/subtract operations. Any transfers between the libraries multi-precision storage arrays and vector registers are likely to exceed the timing for a direct vector implementation.

Note

> The PowerISA 2.07 provides direct vector quadword integer add/subtract with carry/extend. PowerISA 3.0 provides unsigned doubleword multiply with quadword product. This exceeds the capability of the PowerISA 64-bit (doubleword) Fixed Point unit which requires multiple instructions to generate quadword results.

We also want to provide the basis for general *multiple quadword precision arithmetic* operations (see vec_int512_ppc.h). And for security implementations requiring large multiply products we are motivated to leverage the PowerISA large vector register set to avoid exposing these results (and partial products) to memory/cache side channel attacks.

7.7.3.1 Some useful arithmetic facts (you may of forgotten)

First multiplying a M-digits by N-digits number requires up to (M+N)-digits to store the result. This is true independent of the size of your digit, including decimal, hexadecimal, and computer words/doublewords/quadwords. This explains why a 32-bit (word) by 32-bit integer multiply product is either:

- Truncated (modulo) to 32-bits, potentially loosing the high order precision.
- Expanded to the next larger (double) size (in this case 64-bit doubleword).

The hardware has to one or the other.

Let's looks at some examples of multiplying two maximal 4-digit numbers:

```
Decimal:        9999 x 9999 = 99980001
Hexadecimal:    FFFF x FFFF = FFFE0001
```

And to drive home the point, let's look at the case of multiplying two maximal (32-bit word) 4-digit numbers:

```
quadword:       FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
              x FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
              = FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFE
                00000000 00000000 00000000 00000001
```

This is also a (128-bit quadword) digit multiply with a (256-bit) 2 quadword digit result.

Adding asymmetric example; 4-digit by 1 digit multiply:

```
Decimal:        9999 x 9 = 89991
Hexadecimal:    FFFF x F = EFFF1
quadword:       FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
              x FFFFFFFF
              = FFFFFFFE FFFFFFFF FFFFFFFF FFFFFFFF 00000001
```

This pattern repeats across the all digit bases/size and values of M, N.

Note that the product is not the maximum value for the product width. It seem the product leave *room* to add another digit or two without overflowing the double wide product. Lets try some 4 digit examples by adding a maximal 4 digit value to the product.

```
Decimal:       9999 x 9999 = 99980001
                           +     9999
                           = 99990000
Hexadecimal:   FFFF x FFFF = FFFE0001
                           +     FFFF
                           = FFFF0000
```

Looks like there is still room in the double wide product to add another maximal 4 digit value.

```
Decimal:       9999 x 9999 = 99980001
                           +     9999
                           +     9999
                           = 99999999
Hexadecimal:   FFFF x FFFF = FFFE0001
                           +     FFFF
                           +     FFFF
                           = FFFFFFFF
```

But any more then that would cause a overflow.

Now we should look addends to asymmetric multiply. For example 4-digit by 1 digit multiply:

```
Decimal:       9999 x 9 = 89991
                        +  9999
                        +     9
                        = 99999
Hexadecimal:   FFFF x F = EFFF1
                        +  FFFF
                        +     F
                        = FFFFF
quadword:      FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
             x FFFFFFFF
             = FFFFFFFE FFFFFFFF FFFFFFFF FFFFFFFF 00000001
             +          FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
             +                                     FFFFFFFF
             = FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
```

Note that when M not equal N then the addends are restrict to size M and/or size N. Two addends of the larger multiplier size can overflow. This pattern repeats across the all digit bases/sizes and values of M, N. For the binary fixed pointer multiply-add or bit sizes M/N we can write the equation:

$$(2^{(M+N)} - 1) = ((2^{M} - 1) * (2^{N} - 1)) + (2^{M} - 1) + (2^{N} - 1)$$

Or in terms of fixed sized "words" of W-bits and M by N words.

$$(2^{(W*(M+N))} - 1) = ((2^{(W*M)} - 1) * (2^{(W*N)} - 1)) + (2^{(W*M)} - 1) + (2^{(W*N)} - 1)$$

7.7.3.2 Why does this matter?

Because with modern hardware the actual multiply operations are faster and have less impact while the summation across the partial products becomes the major bottleneck. For recent POWER processors fixed-point are 5-7 cycles latency and dual issue (2/cycle). These multiplies are only dependent on the inputs (multiplicands). This allows the compiler and (super-scalar processor) to schedule the multiply operations early to prepare for summation. In many cases the 3rd and 4th multiplies are complete before the summation of the first two multiplies completes.

The add operations involved in partial product summation are dependent on the current column multiply and the high order word of summation of the previous stage. While add operations are nominally faster (2-3 cycles) than multiplies, they can generate carries that have to be propagated.

The Fixed-Point Unit has a dedicated *carry-bit (CA)* which becomes the critical resource. This dependency on the carry (in addition to the column multiply and previous summation) limits the compiler's (and hardware's) ability to parallelize stages of the summation. The Vector unit (PowerISA 2.07+) has quadword (vs Fixed point doubleword) binary add/subtract with carry/extend. The Vector Unit requires separate *write Carry* instructions to detect and return the carry to VRs. The *write Carry* instructions are paired with *Unsigned Quadword Modulo* instructions that generates the (modulo) 128-bit result.

Note

> In PowerISA 3.0B has a new add extended (addex) instruction that can use the *overflow-bit (OF)* as a second carry (independent of CA). However the OF must be explicitly cleared (using subfo) before use as a carry flag.
>
> The Vector Unit has the effective use of up to 32 carry bits. The down-side is it requires an extra instruction and whole 128-bit VR ro generate and hold each carry bit.

So knowing how to avoid overflows and carries in the summation of partial products can be useful. To illustrate we can examine the POWER8 implementation of vec_muludq(). POWER8 (PowerISA 2.07) does support add quadword but the largest vector fixed-point multiply is 32-bit Vector Multiply Even/Odd Unsigned Words (vec_muleuw() and (vec_mulouw()). The implementation generates four quadword by word (160-bit) partial products that are summed in four stages to generate the final 256-bit product.

Code for the first stage looks like this:

```
// Splat the lowest order word of b to tsw for word multiply
tsw = vec_splat ((vui32_t) b, VEC_WE_3);
// Multiply quadword a by lowest order word of b
t_even = (vui32_t)vec_vmuleuw((vui32_t)a, tsw);
t_odd = (vui32_t)vec_vmulouw((vui32_t)a, tsw);
// Rotate the low 32-bits (right) into tmq. This is actually
// implemented as 96-bit (12-byte) shift left.
tmq = vec_sld (t_odd, z, 12);
// shift the low 128 bits of partial product right 32-bits
t_odd = vec_sld (z, t_odd, 12);
// add the high 128 bits of even / odd partial products
t = (vui32_t) vec_adduqm ((vui128_t) t_even, (vui128_t) t_odd);
```

Note in this case we can assume that the sum of aligned even/odd quadwords will not generate a carry. For example with maximum values for multiplicands a,b:

```
quadword a:    FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
quadword b   x FFFFFFFF[3]
t_even       = FFFFFFFE 00000001 FFFFFFFE 00000001
t_odd » 32  + 00000000 FFFFFFFE 00000001 FFFFFFFE
t            = FFFFFFFE FFFFFFFF FFFFFFFF FFFFFFFF
tmq          = 00000001
```

The high order 128-bits of the sum did not overflow.

The next tree stages are more complex.

```
// Splat the next word of b to tsw for word multiply
tsw = vec_splat ((vui32_t) b, VEC_WE_2);
// Multiply quadword a by next word of b
t_even = (vui32_t)vec_vmuleuw((vui32_t)a, tsw);
t_odd = (vui32_t)vec_vmulouw((vui32_t)a, tsw);
// Add with carry the odd multiply with previous partial product
tc = (vui32_t) vec_addcuq ((vui128_t) t_odd, (vui128_t) t);
t_odd = (vui32_t) vec_adduqm ((vui128_t) t_odd, (vui128_t) t);
// Rotate the low 32-bits (right) into tmq.
tmq = vec_sld (t_odd, tmq, 12);
// shift the low 128 bits (with carry) right 32-bits
t_odd = vec_sld (tc, t_odd, 12);
// add the high 128 bits of even / odd partial products
t = (vui32_t) vec_adduqm ((vui128_t) t_even, (vui128_t) t_odd);
```

Here we need a 3-way sum of the previous partial product, and the odd, even products from this stage. In this case the high 128-bits of previous partial product needs to align with the lower 128-bits of this stages 160-bit product for the first quadword add. This can produce a overflow, so we need to capture the carry and concatenate it the odd sum before shifting right 32-bits. Again we can assume that the sum of aligned even/odd quadwords will not generate a carry. For example stage 2 with maximum values for multiplicands a,b:

```
quadword a:    FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
quadword b   x FFFFFFFF[2]
t_odd          FFFFFFFE 00000001 FFFFFFFE 00000001
t            + FFFFFFFE FFFFFFFF FFFFFFFF FFFFFFFF
t_odd        = FFFFFFFD 00000001 FFFFFFFE 00000000
tc           = 00000000 00000000 00000000 00000001
tc|t_odd»32 = 00000001 FFFFFFFD 00000001 FFFFFFFE
t_odd|tmq    = 00000000 00000001
t_even       = FFFFFFFE 00000001 FFFFFFFE 00000001
```

```
tc|t_odd»32 + 00000001 FFFFFFFD 00000001 FFFFFFFE
t             = FFFFFFFF FFFFFFFE FFFFFFFF FFFFFFFF
```

For POWER8 this 3-way sum and the required write-carry adds significant latency to stages 2, 3, and 4 of this multiply.

In POWER8 the vector quadword add/subtract instructions are cracked into 2 dependent simple fixed-point (XS) IOPs. So the effective instruction latency is (2+2=4) cycles. Also cracked instructions must be *first in group*, so back-to-back vaddcuq/vadduqm sequences will be dispatched separately. There no possibility of executing the pair concurrently, so the latency for the pair is 5-6 cycles.

So there is value in finding an alternative summation that avoids/reduces the number write-carry operations. From above (Some useful arithmetic facts (you may of forgotten)) we know it is possible to add one or two unsigned words to each of the doubleword products generated by vmuleuw/vmulouw.

We need to align the words of the quadword addend (zero extended on the left to doublewords) with the corresponding doublewords of the products. We can use Vector Merge Even/Odd Word operations to split and pad the addend into to align with the products. Then we use Vector Add Doubleword for the even/odd product-sums. Finally we use shift and add quadword to produce the 160-bit stage 2 sum.

```
quadword a:    FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
quadword b   x FFFFFFFF[2]
quadword t:    FFFFFFFE FFFFFFFF FFFFFFFF FFFFFFFF
t_even       = FFFFFFFE 00000001 FFFFFFFE 00000001
mrgew(z,t)   + 00000000 FFFFFFFE 00000000 FFFFFFFF
             = FFFFFFFE FFFFFFFF FFFFFFFF 00000000
t_odd        = FFFFFFFE 00000001 FFFFFFFE 00000001
mrgow(z,t)   + 00000000 FFFFFFFF 00000000 FFFFFFFF
             = FFFFFFFF 00000000 FFFFFFFF 00000000
t_odd»32    = 00000000 FFFFFFFF 00000000 FFFFFFFF
t_odd|tmq»32= 00000000 00000001
t_even       = FFFFFFFE FFFFFFFF FFFFFFFF 00000000
t_odd»32    + 00000000 FFFFFFFF 00000000 FFFFFFFF
t            = FFFFFFFF FFFFFFFE FFFFFFFF FFFFFFFF
t_odd|tmq    = 00000000 00000001
```

This sequence replaces two instructions (vaddcuq/vadduqm) with four instructions (vmrgew/vmrgow/vaddudm/vaddudm), all of which;

- have 2 cycle latency
- are dual issue
- without dispatch restrictions

We expect a latency of 4 cycles over the whole sequence. And splitting the first add into even/odd add blocks allows the compiler (and out-of-order hardware) more flexibility for instruction scheduling.

7.7.3.2.1 Vector Multiply-Add

Multiply-add seems to be a useful operation that does not exist in the current Power ISA. But it is simple enough to create an in-line PVECLIB operation that we can use here. For example:

```
static inline vui64_t
vec_vmaddeuw (vui32_t a, vui32_t b, vui32_t c)
{
  const vui32_t zero = { 0, 0, 0, 0 };
  vui64_t res;
  vui32_t c_euw = vec_mrgahw ((vui64_t) zero, (vui64_t) c);
  res = vec_vmuleuw (a, b);
  return vec_addudm (res, (vui64_t) c_euw);
}
```

Which generates the following instruction sequence:

```
<__vec_vmaddeuw_PWR8>:
     d70:       vmuleuw v2,v2,v3
     d74:       vspltisw v0,0
     d78:       vmrgew  v4,v0,v4
     d7c:       vaddudm v2,v2,v4
```

The vspltisw loads (immediate) the zero vector and the compiler should *common* this across operations and schedule this instruction once, early in the function. The vmrgew has a latency of 2 cycles and should execute concurrently with vmuleuw. Similarly for vec_vmaddouw().

These operations (vec_vmaddeuw() and vec_vmaddouw()) are included in vec_int64_ppc.h as they require vec_addudm() and produce doubleword results. With this addition we can improve and simplify the code for stages 2-4 of the _ARCH_PWR8 implementation of vec_muludq(). For example:

```
// Splat the next word of b to tsw for word multiply
tsw = vec_splat ((vui32_t) b, VEC_WE_2);
// Multiply quadword a by next word of b and add previous partial
// product using multiply-add even/odd
t_even = (vui32_t)vec_vmaddeuw((vui32_t)a, tsw, t);
t_odd = (vui32_t)vec_vmaddouw((vui32_t)a, tsw, t);
// Rotate the low 32-bits (right) into tmq.
tmq = vec_sld (t_odd, tmq, 12);
// shift the low 128 bits (with carry) right 32-bits
t_odd = vec_sld (z, t_odd, 12);
// add the high 128 bits of even / odd partial products
t = (vui32_t) vec_adduqm ((vui128_t) t_even, (vui128_t) t_odd);
```

7.7.3.2.2 And Vector Multiply-Add2

From the description above (Some useful arithmetic facts (you may of forgotten)) we know we can add two unsigned words to the doubleword product without overflow. This is another useful operation that does not exist in the current PowerISA. But it is simple enough to create an in-line PVECLIB operation. For example:

```
static inline vui64_t
vec_vmadd2euw (vui32_t a, vui32_t b, vui32_t c, vui32_t d)
{
  const vui32_t zero = { 0, 0, 0, 0 };
  vui64_t res, sum;
  vui32_t c_euw = vec_mrgahw ((vui64_t) zero, (vui64_t) c);
  vui32_t d_euw = vec_mrgahw ((vui64_t) zero, (vui64_t) d);
  res = vec_vmuleuw (a, b);
  sum = vec_addudm ( (vui64_t) c_euw, (vui64_t) d_euw);
  return vec_addudm (res, sum);
}
```

Which generates to following instruction sequence:

```
<__vec_vmadd2euw_PWR8>:
    db0:       vmuleuw v2,v2,v3
    db4:       vspltisw v0,0
    db8:       vmrgew  v4,v0,v4
    dbc:       vmrgew  v5,v0,v5
    dc0:       vaddudm v5,v4,v5
    dc4:       vaddudm v2,v2,v5
```

The vspltisw loads (immediate) the zero vector and the compiler should *common* this across operations and schedule this instruction once, early in the function. The vmrgew/vmrgew/vaddudm sequence has a latency of 4-6 cycles and should execute concurrently with vmuleuw. Similarly for vec_vmadd2ouw().

7.7.3.2.3 Why not Vector Multiply-Sum

The PowerISA has a number of Multiply-Sum instructions that look a lot like the Multiply-Add described above? Well not exactly:

- The behavior of Multiply-Sum allows overflow without any architected way to detect/capture and propagate the carry.
  - Each of the two (even/odd) halves of each "word" element of VRA and VRB: Multiply the even halves of each "word" element. Then multiply the odd halves of each "word" element. This generates two unsigned integer "word" products for each "word" element.
  - The sum of these two integer "word" products is added to the corresponding integer "word" element in VRC.

  - This 3-way sum of can overflow without notification.
- Multiply-Sum instructions can be used to emulate Multiply Even/Odd and Multiply-Add Even/Odd by constraining the inputs.
  - Using Multiply-Sum to add prior partial-sums creates a serial dependency that limits instruction scheduling and slows execution.
- The PowerISA does not have Multiply-Sum Word instructions.
- The PowerISA 3.0 has a Multiply-Sum Unsigned Doubleword instruction but it does not exist in POWER8.
- The base Altivec has Multiply-Sum Halfword/Byte instructions. But using POWER8's Multiply Even/Odd Unsigned Word is better for implementing quadword multiply on POWER8.

First we should look at the arithmetic of Multiply-Sum using maximal unsigned integer values.

```
VRA:           FFFF x FFFF
VRB:           FFFF x FFFF
VRC:           FFFF   FFFF
Even half:     FFFF x FFFF ->    FFFE0001
odd half:      FFFF x FFFF -> +  FFFE0001
Word addend                -> +  FFFFFFFF
                           =   2 FFFC0001
```

Note the sum overflows the word twice and high order bits of the sum will be lost.

For POWER9 we can simulate Vector Multiply Even/Odd Unsigned Doubleword by setting the Odd/Even doubleword of VRB to zero and the whole quadword addend VRC to zero. For example the even doubleword multiply.

```
static inline vui128_t
vec_vmuleud (vui64_t a, vui64_t b)
{
  const vui64_t zero = { 0, 0 };
  vui64_t b_eud = vec_mrgahd ((vui128_t) b, (vui128_t) zero);
  return vec_msumudm(a, b_eud, zero);
}
```

And similarly for the odd doubleword multiply.

```
static inline vui128_t
vec_vmuloud (vui64_t a, vui64_t b)
{
  const vui64_t zero = { 0, 0 };
  vui64_t b_oud = vec_mrgald ((vui128_t) zero, (vui128_t) b);
  return vec_msumudm(a, b_oud, (vui128_t) zero);
}
```

And review the arithmetic for vec_vmuleud() using maximal quadword values for a and b.

```
quadword a:    FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
quadword b:    FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
quadword z:    00000000 00000000 00000000 00000000
mrged(b,z)   = FFFFFFFF FFFFFFFF 00000000 00000000
Even prod:     FFFFFFFF FFFFFFFE 00000000 00000001
odd prod     + 00000000 00000000 00000000 00000000
Word addend  + 00000000 00000000 00000000 00000000
msumudm      = FFFFFFFF FFFFFFFE 00000000 00000001
```

And for vec_vmuldud().

```
quadword a:    FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
quadword b:    FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
quadword z:    00000000 00000000 00000000 00000000
mrgod(z,b)   = 00000000 00000000 FFFFFFFF FFFFFFFF
Even prod:     00000000 00000000 00000000 00000000
odd prod     + FFFFFFFF FFFFFFFE 00000000 00000001
Word addend  + 00000000 00000000 00000000 00000000
msumudm      = FFFFFFFF FFFFFFFE 00000000 00000001
```

We can also simulate Vector Multiply-Add Even/Odd Unsigned Doubleword by setting the odd/even doubleword of VRB to zero and the whole quadword addend to the even/odd double word of VRC. For example the even doubleword multiply-add.

```
static inline vui128_t
vec_vmaddeud (vui64_t a, vui64_t b, vui64_t c)
{
  const vui64_t zero = { 0, 0 };
  vui64_t b_eud = vec_mrgahd ((vui128_t) b, (vui128_t) zero);
  vui64_t c_eud = vec_mrgahd ((vui128_t) zero, (vui128_t) c);
  return vec_msumudm(a, b_eud, (vui128_t) c_eud);
}
```

And similarly for the odd doubleword multiply-add.

```
static inline vui128_t
vec_vmaddoud (vui64_t a, vui64_t b, vui64_t c)
{
  const vui64_t zero = { 0, 0 };
  vui64_t b_oud = vec_mrgald ((vui128_t) zero, (vui128_t) b);
  vui64_t c_oud = vec_mrgald ((vui128_t) zero, (vui128_t) c);
  return vec_msumudm(a, b_oud, (vui128_t) c_oud);
}
```

And review the arithmetic for vec_vmaddeud() using maximal quadword values for a and b. The even/odd doublewords of c have slightly different values for illustrative purposes.

```
quadword a:     FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
quadword b:     FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
quadword c:     FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFE
mrged(b,z)    = FFFFFFFF FFFFFFFF 00000000 00000000
mrged(z,c)    = 00000000 00000000 FFFFFFFF FFFFFFFF
Even prod:      FFFFFFFF FFFFFFFE 00000000 00000001
odd prod      + 00000000 00000000 00000000 00000000
Word addend   + 00000000 00000000 FFFFFFFF FFFFFFFF
msumudm       = FFFFFFFF FFFFFFFF 00000000 00000000
```

And for vec_vmaddoud().

```
quadword a:     FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
quadword b:     FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
quadword c:     FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFE
mrgod(z,b)    = 00000000 00000000 FFFFFFFF FFFFFFFF
mrgod(z,c)    = 00000000 00000000 FFFFFFFF FFFFFFFE
Even prod:      00000000 00000000 00000000 00000000
odd prod      + FFFFFFFF FFFFFFFE 00000000 00000001
Word addend   + 00000000 00000000 FFFFFFFF FFFFFFFE
msumudm       = FFFFFFFF FFFFFFFE FFFFFFFF FFFFFFFF
```

This multiply-add even/odd doulbeword form only adds one additional (xxmrghd AKA xxpermdi) instruction over that required for the base multiply even/odd doubleword operation.

```
<__vmuleud_PWR9>:
     120:       xxspltib v0,0
     124:       xxmrghd v3,v3,v0
     128:       vmsumudm v2,v2,v3,v0
<__vmaddeud_PWR9>:
     1a0:       xxspltib v0,0
     1a4:       xxmrghd v3,v3,v0
     1a8:       xxmrghd v4,v0,v4
     1ac:       vmsumudm v2,v2,v3,v4
```

The xxspltib loads (immediate) the zero vector and the compiler should *common* this across operations and schedule this instruction once, early in the function.

For POWER9 instruction instruction timing is different and there are some unique trade-offs. The implementations above are small and appropriate for single instances of multiply doubleword or implementations of multiply quadword. However using the vmsumudm (operand VRC) addend creates a serial dependency within the multiply quadword implementation. When multiply quadword and multiply-add quadword are used in the implementation of wider multiplies (see vec_int512_ppc.h) these serial dependencies actually slow down the implementation.

- A full 128 x 128-bit multiply only requires two stages of even/odd doubleword multiplies. This allows some simplification.
  - Alignment shifts can be replaced with permute doubleword immediate (xxmrgld/xxmrghd/xxpermdi) operations.

  - Careful rearrangement of the operations and operands allow the compiler to optimize (as common subexpressions) some of the doubleword masking operations.
- The multiply even/odd doubleword operations require explicit masking of the even/odd multiplicands.
  - Doubleword masking can be done with xxmrgld/xxmrghd/xxpermdi instructions which are dual issue with a 3 cycle latency.
  - The multiplies (vmsumudm) are serially dependent on these masking instructions.
  - In the POWER8 implementation (using vmuleuw/vmulouw) the multiplicand masking is implicit to the instruction.
- The vmsumudm with the VRC addend can be used to combine the multiply-add of the partial production from the previous stage.
  - This also requires explicit doubleword masking to avoid overflowing the quadword sum.
  - This can make the masking operation and the multiply itself, serially dependent on the partial product sum from the previous stage.
- The add (modulo/write-carry/extend) quadword instructions are dual issue with a 3 cycle latency. So the cost of quadword sums and generating/propagating carries is of less concern (than on POWER8).
  - It can be better to use explicit add quadword and avoid the serial dependency on the vmsumudm (VRC) addend.
  - This allows the compiler (and out-of-order hardware) more flexibility for instruction scheduling.

So lets look at some examples using the vmsumudm (VRC) addend and the alternative using VRC (settting VRA to zero) and explicit add quadword. First a 128x128-bit unsigned multiply using vmsumudm and exploiting the VRC addend where appropriate.

```
vui128_t
__test_muludq_y_PWR9 (vui128_t *mulu, vui128_t a, vui128_t b)
{
  vui32_t t, tmq;
  // compute the 256 bit product of two 128 bit values a, b.
  // The high 128 bits are accumulated in t and the low 128-bits
  // in tmq. The high 128-bits of the product are returned to the
  // address of the 1st parm. The low 128-bits are the return
  // value.
  const vui64_t zero = { 0, 0 };
  vui64_t a_swap = vec_swapd ((vui64_t) a);
  vui128_t tmh, tab, tba, tb0, tc1, tc2;
  // multiply the low 64-bits of a and b.  For PWR9 this is just
  // vmsumudm with conditioned inputs.
  tmq = (vui32_t) vec_vmuloud ((vui64_t)a, (vui64_t)b);
  // compute the 2 middle partial projects.  Use vmaddeud to add the
  // high 64-bits of the low product to one of the middle products.
  // This can not overflow.
  tab = vec_vmuloud (a_swap, (vui64_t) b);
  tba = vec_vmaddeud (a_swap, (vui64_t) b, (vui64_t) tmq);
  // sum the two middle products (plus the high 64-bits of the low
  // product.  This will generate a carry that we need to capture.
  t   = (vui32_t) vec_adduqm (tab, tba);
  tc1 = vec_addcuq (tab, tba);
  // result = t[l] || tmq[l].
  tmq = (vui32_t) vec_mrgald ((vui128_t) t, (vui128_t) tmq);
  // we can use multiply sum here because the high product plus the
  // high sum of middle partial products can't overflow.
  t   = (vui32_t) vec_permdi ((vui64_t) tc1, (vui64_t) t, 2);
  // This is equivalent to vec_vmadd2eud(a, b, tab, tba)
  // were (tab_even + tba_even) was pre-computed including the carry,
  // so no masking is required.
  t   = (vui32_t) vec_vmsumeud ((vui64_t) a, (vui64_t) b, (vui128_t) t);
  *mulu = (vui128_t) t;
  return ((vui128_t) tmq);
}
<__test_muludq_y_PWR9>:
    370:       xxspltib v1,0
    374:       xxswapd v12,v2
    378:       xxlor   v13,v2,v2
    37c:       xxmrgld v0,v1,v3
    380:       xxmrghd v3,v3,v1
    384:       vmsumudm v2,v2,v0,v1
    388:       vmsumudm v0,v12,v0,v1
    38c:       xxmrghd v1,v1,v2
    390:       vmsumudm v1,v12,v3,v1
    394:       vadduqm v12,v1,v0
    398:       vaddcuq v0,v0,v1
    39c:       xxmrgld v2,v12,v2
    3a0:       xxpermdi v0,v0,v12,2
    3a4:       vmsumudm v13,v13,v3,v0
    3a8:       stxv    v13,0(r3)
    3ac:       blr
```

Note

> that first vmsumudm instruction is only dependent on the parameters a, masked b_odd, and const zero. The second vmsumudm instruction is only dependent on the parameters a_swap, masked b_odd, and const zero. The swap/mask operations requires 3-4 cycles and 7 cycles to complete first two vmsumudm's. The third vmsumudm instruction is dependent on the parameters a_swap, masked b_even, and masked tmq_even. The masked tmq←_even is dependent on the xxmrghd of the results of the first vmsumudm. This adds another 10 cycles. The forth and final vmsumudm instruction is dependent on the parameters a, masked b_even, and the shifted sum (with carry) of (tab + tba). This is in turn dependent on the results from the second and third vmsumudm instructions. This adds another (6+7= 13) cycles for a total of 34 cycles. When this operation is expanded in-line the stxv and xxspltib will be optimized and can be ignored for this analysis.

Next a 128x128-bit unsigned multiply using vmsumudm but only passing const zero to the VRC addend.

```
vui128_t
__test_muludq_x_PWR9 (vui128_t *mulu, vui128_t a, vui128_t b)
{
  // compute the 256 bit product of two 128 bit values a, b.
  // The high 128 bits are accumulated in t and the low 128-bits
  // in tmq. The high 128-bits of the product are returned to the
  // address of the 1st parm. The low 128-bits are the return
  // value.
  const vui64_t zero = { 0, 0 };
  vui64_t a_swap = vec_swapd ((vui64_t) a);
  vui128_t thq, tlq, tx;
  vui128_t t0l, tc1;
  vui128_t thh, thl, tlh, tll;
  // multiply the low 64-bits of a and b.  For PWR9 this is just
  // vmsumudm with conditioned inputs.
  tll = vec_vmuloud ((vui64_t)a, (vui64_t)b);
  thh = vec_vmuleud ((vui64_t)a, (vui64_t)b);
  thl = vec_vmuloud (a_swap, (vui64_t)b);
  tlh = vec_vmuleud (a_swap, (vui64_t)b);
  // sum the two middle products (plus the high 64-bits of the low
  // product.  This will generate a carry that we need to capture.
  t0l   = (vui128_t) vec_mrgahd ( (vui128_t) zero, tll);
  tc1 = vec_addcuq (thl, tlh);
  tx   = vec_adduqm (thl, tlh);
  tx   = vec_adduqm (tx, t0l);
  // result = t[l] || tll[l].
  tlq = (vui128_t) vec_mrgald ((vui128_t) tx, (vui128_t) tll);
  // Sum the high product plus the high sum (with carry) of middle
  // partial products.  This can't overflow.
  thq = (vui128_t) vec_permdi ((vui64_t) tc1, (vui64_t) tx, 2);
  thq = vec_adduqm ( thh, thq);
  *mulu = (vui128_t) thq;
  return ((vui128_t) tlq);
}
<__test_muludq_x_PWR9>:
    320:       xxspltib v0,0
    324:       xxswapd v12,v2
    328:       xxmrgld v13,v0,v3
    32c:       xxmrghd v3,v3,v0
    330:       vmsumudm v1,v12,v13,v0
    334:       vmsumudm v13,v2,v13,v0
    338:       vmsumudm v12,v12,v3,v0
    33c:       xxmrghd v10,v0,v13
    340:       vadduqm v11,v12,v1
    344:       vmsumudm v3,v2,v3,v0
    348:       vaddcuq v1,v1,v12
    34c:       vadduqm v2,v11,v10
    350:       xxpermdi v1,v1,v2,2
    354:       xxmrgld v2,v2,v13
    358:       vadduqm v3,v3,v1
    35c:       stxv    v3,0(r3)
    360:       blr
```

Note

> that the vmsumudm instructions only depend on the parameters a/a_swap, masked b_odd/b_even, and const zero. After the parameters are conditioned (swapped/masked) the independent vmsumudm's can be scheduled early. The swap/mask operations requires 3-4 cycles and 8 cycles to complete four independent vmsumudm's. The partial product alignment and sums require another 12 cycles, for a total of 24 cycles. When this operation is expanded in-line the stxv and xxspltib will be optimized and can be ignored for this analysis.

The second example (using explicit add quadword);

- only adds 1 instruction over the first example,
- and executes 10 cycles faster.

7.7.3.2.4 Vector Multiply-Add Quadword We can use multiply-add operation for wider word sizes (quadword and multiple precision quadword). The simplest quadword implementation would create a vec_madduq() operation based on vec_muludq() and add a quadword parameter "c" for the addend. Then modify the first stage of the platform specific multiplies to replace vector multiply even/odd with vector multiply-add even/odd, passing the addend as the the third parameter.

This works well for the POWER8 implementation because the additional vector add doublewords can be scheduled independently of the vector multiply even/odd words. But for POWER9 we need to avoid the serial dependences explained above in Why not Vector Multiply-Sum.

For the POWER9 implementation we use an explicit add quadword (and write-Carry) to sum the addend parameter to the first stage Multiply odd doubleword. For example:

```
 vui128_t
__test_madduq_y_PWR9 (vui128_t *mulu, vui128_t a, vui128_t b, vui128_t c)
{
  // compute the 256 bit sum of product of two 128 bit values a, b
  // plus the quadword addend c.
  vui64_t a_swap = vec_swapd ((vui64_t) a);
  vui128_t thq, tlq, tx;
  vui128_t t0l, tc1, tcl;
  vui128_t thh, thl, tlh, tll;
  // multiply the four combinations of a_odd/a_even by b_odd/b_even.
  tll = vec_vmuloud ((vui64_t)a, (vui64_t)b);
  thh = vec_vmuleud ((vui64_t)a, (vui64_t)b);
  thl = vec_vmuloud (a_swap, (vui64_t)b);
  tlh = vec_vmuleud (a_swap, (vui64_t)b);
  // Add c to lower 128-bits of the partial product.
  tcl = vec_addcuq (tll, c);
  tll = vec_adduqm (tll, c);
  t0l = (vui128_t) vec_permdi ((vui64_t) tcl, (vui64_t) tll, 2);
  // sum the two middle products (plus the high 65-bits of the low
  // product-sum).
  tc1 = vec_addcuq (thl, tlh);
  tx  = vec_adduqm (thl, tlh);
  tx  = vec_adduqm (tx, t0l);
  // result = tx[l]_odd || tll[l]_odd.
  tlq = (vui128_t) vec_mrgald ((vui128_t) tx, (vui128_t) tll);
  // Sum the high product plus the high sum (with carry) of middle
  // partial products.  This can't overflow.
  thq = (vui128_t) vec_permdi ((vui64_t) tc1, (vui64_t) tx, 2);
  thq = vec_adduqm ( thh, thq);
  *mulu = (vui128_t) thq;
  return ((vui128_t) tlq);
}
```

The generated code is the same size as the serially depended version

This is just another example where the shortest instruction sequence or using the most powerful instructions, may not be the fastest implementation. The key point is that avoiding serial dependencies in the code and allowing the compiler to schedule high latency instructions early, allows better performance. This effect is amplified when quadword multiplies (vec_muludq(), vec_madduq(), and vec_madd2uq()) are used to compose wider multiply operations (see vec_int512_ppc.h).

7.7.4 Vector Quadword Examples

The PowerISA Vector facilities provide logical and integer arithmetic quadword (128-bit) operations. Some operations as direct PowerISA instructions and other operations composed of short instruction sequences. The Power Vector Library provides a higher level and comprehensive API of quadword integer integer arithmetic and support for extended arithmetic to multiple quadwords.

7.7.4.1 Printing Vector __int128 values

The GCC compiler supports the (vector) __int128 type but the runtime does not support **printf()** formating for __int128 types. However if we can use divide/modulo operations to split vector __int128 values into modulo 10^16 long int (doubleword) chunks, we can use printf() to convert and concatenate the decimal values into a complete number.

For example, from the __int128 value (39 decimal digits):

- Detect the sign and set a char to "+' or '-'
- Then from the absolute value, divide/modulo by 10000000000000000. Producing:
  - The highest 7 digits (t_high)
  - The middle 16 digits (t_mid)
  - The lowest 16 digits (t_low)

We can use signed compare to detect the sign and set a char value to print a ' ' or '+' prefix. If the value is negative we want the absolute value before we do the divide/modulo steps. For example:

```
if (vec_cmpsq_all_ge (value, zero128))
  {
    sign = ' ';
    val128 = (vui128_t) value;
  }
else
  {
    sign = '-';
    val128 = vec_subuqm ((vui128_t) zero128, (vui128_t) value);
  }
```

Here we use the **pveclib** operation vec_cmpsq_all_ge() because the ABI and compilers do not define compare built-ins operations for the vector __int128 type. For the negative case we use the **pveclib** operation vec_subuqm() instead of vec_abs. Again the ABI and compilers do not define vec_abs built-ins for the vector __int128 type. Using **pveclib** operations have the additional benefit of supporting older compilers and platform specific implementations for POWER7 and POWER8.

Now we have the absolute value in val128 we can factor it into (3) chunks of 16 digits each. Normally scalar codes would use integer divide/modulo by 10000000000000000. And we are reminded that the PowerISA vector unit does not support integer divide operations and definitely not for quadword integers.

Instead we can use the multiplicative inverse which is a scaled fixed point fraction calculated from the original divisor. This works nicely if the fixed radix point is just before the 128-bit fraction and we have a multiply high (vec_mulhuq()) operation. Multiplying a 128-bit unsigned integer by a 128-bit unsigned fraction generates a 256-bit product with 128-bits above (integer) and below (fraction) the radix point. The high 128-bits of the product is the integer quotient and we can discard the low order 128-bits.

It turns out that generating the multiplicative inverse can be tricky. To produce correct results over the full range requires, possible pre-scaling and post-shifting, and sometimes a corrective addition is necessary. Fortunately the mathematics are well understood and are commonly used in optimizing compilers. Even better, Henry Warren's book has a whole chapter on this topic.

See also

> "Hacker's Delight, 2nd Edition," Henry S. Warren, Jr, Addison Wesley, 2013. Chapter 10, Integer Division by Constants.

In the chapter above;

> Figure 10-2 Computing the magic number for unsigned division.

provides a sample C function for generating the magic number (actually a struct containing; the magic multiplicative inverse, "add" indicator, and the shift amount.). For quadword and the divisor 10000000000000000,this is { 76624777043294442917917351357515459181, 0 , 51 }:

- the multiplier is 76624777043294442917917351357515459181.
- no corrective add is required.
- the final shift is 51-bits right.

```
const vui128_t mul_ten16 = (vui128_t) CONST_VINT128_DW(
    0UL, 10000000000000000UL);
// Magic numbers for multiplicative inverse to divide by 10**16
// are 76624777043294442917917351357515459181, no corrective add,
// and shift right 51 bits.
const vui128_t mul_invs_ten16 = (vui128_t) CONST_VINT128_DW(
    0x39a5652fb1137856UL, 0xd30baf9a1e626a6dUL);
const int shift_ten16 = 51;
...
// first divide/modulo the 39 digits __int128 by 10**16.
// This separates the high/middle 23 digits (tmpq) and low 16 digits.
tmpq = vec_mulhuq (val128, mul_invs_ten16);
tmpq = vec_srqi (tmpq, shift_ten16);
// Compute remainder of val128 / 10**16
// t_low = val128 - (tmpq * 10**16)
// Here we know tmpq and mul_ten16 are less then 64-bits
// so can use vec_vmuloud instead of vec_mulluq
tmp = vec_vmuloud ((vui64_t) tmpq, (vui64_t) mul_ten16);
t_low = (vui64_t) vec_subuqm (val128, tmp);
// Next divide/modulo the high/middle digits by 10**16.
// This separates the high 7 and middle 16 digits.
val128 = tmpq;
tmpq = vec_mulhuq (tmpq, mul_invs_ten16);
t_high = (vui64_t) vec_srqi (tmpq, shift_ten16);
tmp = vec_vmuloud (t_high, (vui64_t) mul_ten16);
t_mid = (vui64_t) vec_subuqm (val128, tmp);
```

All the operations used above are defined and implemented by **pveclib**. Most of these operations is not defined as single instructions in the PowerISA or as built-ins the ABI or require alternative implementations for older processors.

Now we have three vector unsigned __int128 values (t_low, t_mid, t_high) in the range 0-9999999999999999. Fixed point values in that range fit into the low order doubleword of each quadword. We can access these doublewords with array notation ([VEC_DW_L]) and the compiler will transfer them to fixed point (long int) GPRs. Then use normal char and long int printf() formating. For example:

```
printf ("%c%07lld%016lld%016lld", sign,
        t_high[VEC_DW_L], t_mid[VEC_DW_L], t_low[VEC_DW_L]);
```

Here is the complete vector __int128 printf example:

```
 void
example_print_vint128 (vi128_t value)
{
  const vi128_t max_neg = (vi128_t) CONST_VINT128_DW(
      0x8000000000000000L, 0UL);
  const vi128_t zero128 = (vi128_t) CONST_VINT128_DW(
      0x0L, 0UL);
  const vui128_t mul_ten16 = (vui128_t) CONST_VINT128_DW(
      0UL, 10000000000000000UL);
  // Magic numbers for multiplicative inverse to divide by 10**16
  // are 76624777043294442917917351357515459181, no corrective add,
  // and shift right 51 bits.
  const vui128_t mul_invs_ten16 = (vui128_t) CONST_VINT128_DW(
      0x39a5652fb1137856UL, 0xd30baf9a1e626a6dUL);
  const int shift_ten16 = 51;
  vui128_t tmpq, tmp;
  vui64_t t_low, t_mid, t_high;
  vui128_t val128;
  char sign;
  if (vec_cmpsq_all_ge (value, zero128))
    {
      sign = ' ';
      val128 = (vui128_t) value;
    }
  else
    {
      sign = '-';
      val128 = vec_subuqm ((vui128_t) zero128, (vui128_t) value);
    }
  // Convert the absolute (unsigned) value to Decimal and
  // prefix the sign.
  // first divide/modulo the 39 digits __int128 by 10**16.
  // This separates the high/middle 23 digits (tmpq) and low 16 digits.
  tmpq = vec_mulhuq (val128, mul_invs_ten16);
  tmpq = vec_srqi (tmpq, shift_ten16);
  // Compute remainder of val128 / 10**16
  // t_low = val128 - (tmpq * 10**16)
  // Here we know tmpq and mul_ten16 are less then 64-bits
  // so can use vec_vmuloud instead of vec_mulluq
  tmp = vec_vmuloud ((vui64_t) tmpq, (vui64_t) mul_ten16);
  t_low = (vui64_t) vec_subuqm (val128, tmp);
  // Next divide/modulo the high/middle digits by 10**16.
  // This separates the high 7 and middle 16 digits.
  val128 = tmpq;
  tmpq = vec_mulhuq (tmpq, mul_invs_ten16);
  t_high = (vui64_t) vec_srqi (tmpq, shift_ten16);
  tmp = vec_vmuloud (t_high, (vui64_t) mul_ten16);
  t_mid = (vui64_t) vec_subuqm (val128, tmp);
  printf ("%c%07lld%016lld%016lld", sign, t_high[VEC_DW_L],
          t_mid[VEC_DW_L], t_low[VEC_DW_L]);
}
```

7.7.4.2 Converting Vector __int128 values to BCD

POWER8 and POWER9 added a number of Binary Code Decimal (BCD) and Zoned Decimal operations that should be helpful for radix conversion and even faster large integer formatting for print.

See also

vec_bcd_ppc.h

The issue remains that __int128 values can represent up to 39 decimal digits while Signed BCD supports only 31 digits. POWER9 provides a **Decimal Convert From Signed Quadword** instruction with the following restriction:

Note

> If the signed value of vrb is less then -(10∗∗31-1) or greater than 10∗∗31-1 the result is too large for the BCD format and the result is undefined.

It would be useful to check for this and if required, factor the __int128 value into to the high order 8 digits and the low order 31 digits. This allows for the safe and correct use of the vec_bcdcfsq() and with some decimal shifts/truncates vec_bcdctz(). This also enables conversion to multiple precision Vector BCD to represent 39 digits and more for radix conversions.

We first address the factoring by providing **Vector Divide by const 10e31 Unsigned Quadword** and **Vector Modulo by const 10e31 Unsigned Quadword** operation. This requires the multiplicative inverse using the vec_mulhuq() operation.

```
static inline vui128_t
vec_divuq_10e31 (vui128_t vra)
  // ten32  = +100000000000000000000000000000000UQ
  const vui128_t ten31 = (vui128_t)
          { (__int128) 1000000000000000UL * (__int128) 10000000000000000UL };
  // Magic numbers for multiplicative inverse to divide by 10**31
  // are 4804950418589725908363185682083061167, corrective add,
  // and shift right 107 bits.
  const vui128_t mul_invs_ten31 = (vui128_t) CONST_VINT128_DW(
      0x039d66589687f9e9UL, 0x01d59f290ee19dafUL);
  const int shift_ten31 = 103;
  vui128_t result, t, q;
  if (vec_cmpuq_all_ge (vra, ten31))
    {
      q = vec_mulhuq (vra, mul_invs_ten31);
      // Need corrective add but want to avoid carry & double quad shift
      // The following avoids the carry and less instructions
      t = vec_subuqm (vra, q);
      t = vec_srqi (t, 1);
      t = vec_adduqm (t, q);
      result = vec_srqi (t, (shift_ten31 - 1));
    }
  else
    result = (vui128_t) { (__int128) 0 };
  return result;
}
```

As the vec_mulhuq() operation is relatively expensive and we expect most __int128 values to 31-digits or less, using a compare to bypass the multiplication and return the 0 quotient, seems a prudent optimization.

So far we only have the quotient (the high order 8 digits) and still need to extract the remainder (the low order 31 digits). This is simply the quotient from above multiplied by 10e31 and subtracted from the original input. To avoid the multiple return value issue we define a modulo operation to take the original value and the quotient from vec_divuq_10e31().

```
static inline vui128_t
vec_moduq_10e31 (vui128_t vra, vui128_t q)
{
  // ten32  = +100000000000000000000000000000000UQ
  const vui128_t ten31 = (vui128_t)
          { (__int128) 1000000000000000UL * (__int128) 10000000000000000UL };
  vui128_t result, t;
  if (vec_cmpuq_all_ge (vra, ten31))
    {
      t = vec_mulluq (q, ten31);
      result = vec_subuqm (vra, t);
    }
  else
    result = vra;
  return result;
}
```

Again as the vec_mulluq() operation is relatively expensive and we expect most __int128 values to 31-digits or less, using a compare to bypass the multiplication and return the input value as the remainder, seems a prudent optimization.

We expect these operations to be used together as in this example.

```
q = vec_divuq_10e31 (a);
r = vec_moduq_10e31 (a, q);
```

We also expect the compiler to common the various constant loads across the two operations as the code is in-lined. This header also provides variants for factoring by 10e32 (to use with the Zone conversion) and signed variants of the 10e31 operation for direct conversion to extend precision signed BCD.

See also

vec_divuq_10e32(), vec_moduq_10e32(), vec_divsq_10e31, vec_modsq_10e31.

7.7.4.3 Extending integer operations beyond Quadword

Some algorithms require even high integer precision than __int128 provides. this includes:

- POSIX compliant conversion between __float128 and _Decimal128 types
- POSIX compliant conversion from double and __float128 to decimal for print.
- Cryptographic operations for Public-key cryptography and Elliptic Curves

The POWER8 provides instructions for extending add and subtract to 128-bit integer and beyond with carry/extend operations (see vec_addcuq(), vec_addecuq(), vec_addeuqm(), vec_adduqm(), (see vec_subcuq(), vec_subecuq(), vec_subeuqm(), vec_subuqm()). POWER9 adds instructions to improve decimal / binary conversion to/from 128-bit integer and beyond with carry/extend operations. And while the PowerISA does not yet provide full 128 x 128 bit integer multiply instructions, it has provided wider integer multiply instructions, beginning in POWER8 (see vec_mulesw(), vec_mulosw(), vec_muleuw(), vec_mulouw()) and again in POWER9 (see vec_msumudm()).

This all allows the **pveclib** to improve (reduce the latency of) the implementation of multiply quadword operations. This includes operations that generate the full 256-bit multiply product (see vec_muludq(), vec_mulhuq(). vec_mulluq()). And this in combination with add/subtract with carry extend quadword allows the coding of even wider (multiple quadword) multiply operations.

7.7.4.3.1 Extended Quadword multiply

The following example performs a 256x256 bit unsigned integer multiply generating a 512-bit product:

```
void
test_mul4uq (vui128_t *__restrict__ mulu, vui128_t m1h, vui128_t m1l,
             vui128_t m2h, vui128_t m2l)
{
  vui128_t mc, mp, mq, mqhl;
  vui128_t mphh, mphl, mplh, mpll;
  mpll = vec_muludq (&mplh, m1l, m2l);
  mp = vec_muludq (&mphl, m1h, m2l);
  mplh = vec_addcq (&mc, mplh, mp);
  mphl = vec_adduqm (mphl, mc);
  mp = vec_muludq (&mqhl, m2h, m1l);
  mplh = vec_addcq (&mq, mplh, mp);
  mphl = vec_addeq (&mc, mphl, mqhl, mq);
  mp = vec_muludq (&mphh, m2h, m1h);
  mphl = vec_addcq (&mq, mphl, mp);
  mphh = vec_addeuqm (mphh, mq, mc);
  mulu[0] = mpll;
  mulu[1] = mplh;
  mulu[2] = mphl;
  mulu[3] = mphh;
}
```

This example generates some additional questions:

- Why use vec_muludq() instead of pairing vec_mulhuq() and vec_mulluq()?
- Why use vec_addcq() instead of pairing vec_addcuq() and vec_adduqm()?
- Why return the 512-bit product via a pointer instead of returning a struct or array of 4 x vui128_t (*homogeneous aggregates*)?

The detailed rationale for this is documented in section Returning extended quadword results. In this specific case (quadword integer operations that generate two vector values) **pveclib** provides both alternatives:

- separate operations each returning a single (high or low order) vector.
- combined operations providing:
  - the lower order vector as the function return value.
  - the high order (carry or high product) vector via a pointer reference parameter.

Either method should provide the same results. For example:

```
mplh = vec_addcq (&mc, mplh, mp);
```

is equivalent to

```
mc   = vec_addcuq (mplh, mp);
mplh = vec_adduqm (mplh, mp);
```

and

```
mpll = vec_muludq (&mplh, m1l, m2l);
```

is equivalent to

```
mpll = vec_mulluq (m1l, m2l);
mplh = vec_mulhud (m1l, m2l);
```

So is there any advantage to separate versus combined operations?

Functionally it is useful to have separate operations for the cases where only one quadword part is needed. For example if you know that a add/subtract operation can not overflow, why generate the carry? Alternatively the quadword greater/less-than compares are based solely on the carry from the subtract quadword, why generate lower 128-bit (modulo) difference? For multiplication the modulo (multiply low) operation is the expected semantic or is known to be sufficient. Alternatively the multiplicative inverse only uses the high order (multiply high) quadword of the product.

From the performance (instruction latency and throughput) perspective, if the algorithm requires the extended result or full product, the combined operation is usually the better choice. Otherwise use the specific single return operation needed. At best, the separate operations may generate the same instruction sequence as the combined operation, But this depends on the target platform and specific optimizations implemented by the compiler.

Note

> For inlined operations the pointer reference in the combined form, is usually optimized to a simple register assignment, by the compiler.
>
> For platform targets where the separate operations each generate a single instruction, we expect the compiler to generate the same instructions as the combined operation. But this is only likely for add/sub quadword on the POWER8 and multiply by 10 quadword on POWER9.

7.7.4.3.2 Quadword Long Division

In the section Converting Vector __int128 values to BCD above we used multiplicative inverse to factor a binary quadword value in two (high quotient and low remainder) parts. Here we divide by a large power of 10 (10^{31} or 10^{32}) of a size where the quotient and remainder allow direct conversion to BCD (see vec_bcdcfsq(), vec_bcdcfuq()). After conversion, the BCD parts can be concatenated to form the larger (39 digit) decimal radix value equivalent of the 128-bit binary value.

We can extend this technique to larger (multiple quadword) binary values but this requires long division. This is the version of the long division you learned in grade school, where a multi-digit value is divided in stages by a single digit. But the digits we are using are really big (10^{31}-1 or 10^{32}-1).

The first step is relatively easy. Start by dividing the left-most *digit* of the dividend by the divisor, generating the integer quotient and remainder. We already have operations to implement that.

```
// initial step for the top digits
dn = d[0];
qh = vec_divuq_10e31 (dn);
rh = vec_moduq_10e31 (dn, qh);
q[0] = qh;
```

The array *d* contains the quadwords of the extended precision integer dividend. The array *q* will contain the quadwords of the extended precision integer quotient. Here we have generated the first *quadword q[0]* digit of the quotient. The remainder *rh* will be used in the next step of the long division.

The process repeats except after the first step we have an intermediate dividend formed from:

- The remainder from the previous step
- Concatenated with the next *digit* of the extended precision quadword dividend.

So for each additional step we need to divide two quadwords (256-bits) by the quadword divisor. Actually this dividend should be less than a full 256-bits because we know the remainder is less than the divisor. So the intermediate dividend is less than ((divisor - 1) $*$ 2^{128}). So we know the quotient can not exceed (2^{128}-1) or one quadword.

Now we need an operation that will divide this double quadword value and provide quotient and remainder that are correct (or close enough). Remember your grade school long division where you would:

- estimate the quotient
- multiply the quotient by the divisor
- subtract this product from the current 2 digit dividend
- check that the remainder is less than the divisor.
  - if the remainder is greater than the divisor; the estimated quotient is too small
  - if the remainder is negative (the product was greater than the dividend); the estimated quotient is too large.
- correct the quotient and remainder if needed before doing the next step.

So we don't need to be perfect, but close enough. As long as we can detect any problems and (if needed) correct the results, we can implement long division to any size.

We already have an operation for dividing a quadword by 10^{31} using the magic numbers for multiplicative inverse. This can easily be extended to multiply double quadword high. For example:

```
// Multiply high [vra||vrb] * mul_invs_ten31
q = vec_mulhuq (vrb, mul_invs_ten31);
q1 = vec_muludq (&t, vra, mul_invs_ten31);
c = vec_addcuq (q1, q);
```

```
q = vec_adduqm (q1, q);
q1 = vec_adduqm (t, c);
// corrective add [q2||q1||q] = [q1||q] + [vra||vrb]
c = vec_addcuq (vrb, q);
q = vec_adduqm (vrb, q);
// q2 is the carry-out from the corrective add
q2 = vec_addecuq (q1, vra, c);
q1 = vec_addeuqm (q1, vra, c);
// shift 384-bits (including the carry) right 107 bits
// Using shift left double quadword shift by (128-107)-bits
r2 = vec_sldqi (q2, q1, (128 - shift_ten31));
result = vec_sldqi (q1, q, (128 - shift_ten31));
```

Here we generate a 256-bit multiply high using the vec_mulhuq() for the low dividend (vrb) and vec_muludq() for high dividend (vra). Then sum the partial products ([t||q1] + [0||q]) to get initial 256-bit product [q1||q]. Then apply the corrective add ([q1||q] + [vra||vrb]). This may generate a carry which needs to be included in the final shift.

Technically we only expect a 128-bit quotient after the shift, but we have 3 quadwords (2 quadwords and a carry) going into the shift right. Also our (estimated) quotient may be *off by 1* and generate a 129-bit result. This is due to using a the magic numbers for 128-bit multiplicative inverse and not regenerating magic numbers for 256-bits. We can't do anything about that now and so return a 256-bit double quadword quotient.

Note

> This is where only needing to be "close enough", works in our favor. We will check and correct the quotient in the modulo operation.

The 256-bits we want are spanning multiple quadwords so we replace a simple quadword shift right with two **Shift Left Double Quadword Immediate** operations and complement the shift count (128 - shift_ten31). This gives a 256-bit quotient which we expect to have zero in the high quadword.

As this operation will be used in a loop for long division operations and the extended multiplies are fairly expensive, we should check for an short-circuit special conditions. The most important special condition is when the dividend is less that the divisor and the quotient is zero. This also helps when the long division dividend may have leading quadword zeros that need to be skipped over. For the full implementation looks like:

```
static inline vui128_t
vec_divudq_10e31 (vui128_t *qh, vui128_t vra, vui128_t vrb)
{
  const vui128_t ten31 = (vui128_t)
          { (__int128) 1000000000000000UL * (__int128) 10000000000000000UL };
  const vui128_t zero = (vui128_t) { (__int128) 0UL };
  // Magic numbers for multiplicative inverse to divide by 10**31
  // are 4804950418589725908363185682083061167, corrective add,
  // and shift right 103 bits.
  const vui128_t mul_invs_ten31 = (vui128_t) CONST_VINT128_DW(
      0x039d66589687f9e9UL, 0x01d59f290ee19dafUL);
  const int shift_ten31 = 103;
  vui128_t result, r2, t, q, q1, q2, c;
  if (vec_cmpuq_all_ne (vra, zero) || vec_cmpuq_all_ge (vrb, ten31))
    {
      // Multiply high [vra||vrb] * mul_invs_ten31
      q = vec_mulhuq (vrb, mul_invs_ten31);
      q1 = vec_muludq (&t, vra, mul_invs_ten31);
      c = vec_addcuq (q1, q);
      q = vec_adduqm (q1, q);
      q1 = vec_adduqm (t, c);
      // corrective add [q2||q1||q] = [q1||q] + [vra||vrb]
      c = vec_addcuq (vrb, q);
      q = vec_adduqm (vrb, q);
      // q2 is the carry-out from the corrective add
      q2 = vec_addecuq (q1, vra, c);
      q1 = vec_addeuqm (q1, vra, c);
      // shift 384-bits (including the carry) right 103 bits
      // Using shift left double quadword shift by (128-103)-bits
      r2 = vec_sldqi (q2, q1, (128 - shift_ten31));
      result = vec_sldqi (q1, q, (128 - shift_ten31));
    }
  else
```

```
    {
      // Dividend is less than divisor then return zero quotient
      r2 = zero;
      result = zero;
    }
  // return 256-bit quotient
  *qh = r2;
  return result;
}
```

To complete the long division operation we need to perform double quadword modulo operations. Here the dividend is two quadwords and the low quadword of the quotient from the divide double quadword operation above. We use multiply double quadword to compute the remainder ([vra||vrb] - (q $*$ 10^{31}). Generating the 256-bit product and difference ensure we can detect the case where the quotient is off-by-1 on the high side.

```
t = vec_muludq (&th, *ql, ten31);
c = vec_subcuq (vrb, t);
t = vec_subuqm (vrb, t);
th = vec_subeuqm (vra, th, c);
// The remainder should be less than the divisor
if (vec_cmpuq_all_ne (th, zero) && vec_cmpuq_all_ge (t, ten31))
  {
    // Otherwise the estimated quotient is off by 1
     *ql = vec_adduqm (*ql, minus_one);
    // And the remainder is negative, so add the divisor
    t = vec_adduqm (t, ten31);
  }
result = t;
```

In this case we need to correct both remainder and the (estimated) quotient. This is a bit tricky as the quotient is normally passed by value, but for this operation we need to pass by reference, which allows the corrected quotient to be passed on to the next step.

Again as this operation will be used in a loop for long division operations and the extended multiplies are fairly expensive, we should check for and short-circuit special conditions. The most important special condition is when the dividend is less that the divisor and the remainder is simply the dividend.

```
static inline vui128_t
vec_modudq_10e31 (vui128_t vra, vui128_t vrb, vui128_t *ql)
{
  // ten31  = +10000000000000000000000000000000UQ
  const vui128_t ten31 = (vui128_t)
          { (__int128) 1000000000000000UL * (__int128) 10000000000000000UL };
  const vui128_t zero = (vui128_t) { (__int128) 0UL };
  const vui128_t minus_one = (vui128_t) { (__int128) -1L };
  vui128_t result, t, th, c;
  if (vec_cmpuq_all_ne (vra, zero) || vec_cmpuq_all_ge (vrb, ten31))
    {
      t = vec_muludq (&th, *ql, ten31);
      c = vec_subcuq (vrb, t);
      t = vec_subuqm (vrb, t);
      th = vec_subeuqm (vra, th, c);
      // The remainder should be less than the divisor
      if (vec_cmpuq_all_ne (th, zero) && vec_cmpuq_all_ge (t, ten31))
        {
          // If not the estimated quotient is off by 1
           *ql = vec_adduqm (*ql, minus_one);
          // And the remainder is negative, so add the divisor
          t = vec_adduqm (t, ten31);
        }
      result = t;
    }
  else
    result = vrb;
  return result;
}
```

Now we have all the operations needed to complete the implementation of long division by the decimal constant (10^{31}).

```
vui128_t
example_longdiv_10e31 (vui128_t *q, vui128_t *d, long int _N)
{
  vui128_t dn, qh, ql, rh;
  long int i;
  // initial step for the top digits
  dn = d[0];
  qh = vec_divuq_10e31 (dn);
  rh = vec_moduq_10e31 (dn, qh);
  q[0] = qh;
  // now we know the remainder is less than the divisor.
  for (i=1; i<_N; i++)
    {
      dn = d[i];
      ql = vec_divudq_10e31 (&qh, rh, dn);
      rh = vec_modudq_10e31 (rh, dn, &ql);
      q[i] = ql;
    }
  // return the final remainder
  return rh;
}
```

The result of each call to example_longdiv_10e31() is the output array *q* of quadwords containing the extended quotient, and the remainder as the return value. The input array *d* and output array *q* should not overlap in storage. The remainder is in the range 0-9999999999999999999999999999999 and is suitable for conversion to BCD or decimal characters. (see vec_bcdcfsq()). Repeated calls passing the quotient from the previous call as the dividend, reduces the quotient by 31 digits and returns another 31 digits in the remainder for conversion. This continues until the quotient is less than 10^{31} which provides the highest order digits of the decimal result.

Note

> Similarly for long division in support of unsigned 32-digit BCD conversion using operations; vec_divuq_10e32(), vec_moduq_10e32(), vec_divudq_10e32(), and vec_modudq_10e32(). Long division for other constant divisors or multiple quadword divisors is an exercise for the student.

Todo The implementation above gives correct results for all the cases tested for divide by constants 10^{31} and 10^{32}). This is not a mathematical proof of correctness, just an observation. Anyone who finds a counter example or offers a mathematical proof should submit a bug report.

7.7.5 Performance data.

High level performance estimates are provided as an aid to function selection when evaluating algorithms. For background on how *Latency* and *Throughput* are derived see: Performance data.

7.7.6 Macro Definition Documentation

7.7.6.1 CONST_VUINT128_Qx16d

```
#define CONST_VUINT128_Qx16d(
                __q0,
                __q1 )
```

Value:

```
    ( (vui128_t) \
    (((unsigned __int128) __q0) * 10000000000000000UL) \
    + ((unsigned __int128) __q1) )
```

Generate a vector unsigned __int128 constant from doublewords.

Combine 2 x 16 decimal digit long long constants into a single 32 decimal digit __int128 constant. The 2 parameters are long integer constant values in high to low order. This order is consistent for big and little endian and the result loaded into vector registers is correct for quadword integer operations.

For example

```
const vui128_t ten32 = CONST_VUINT128_Qx16d (10000000000000000UL, 0UL);
```

7.7.6.2 CONST_VUINT128_Qx18d

```
#define CONST_VUINT128_Qx18d(
               __q0,
               __q1 )
```

Value:

```
    ( (vui128_t) \
    (((unsigned __int128) __q0) * 1000000000000000000UL) \
    + ((unsigned __int128) __q1) )
```

Generate a vector unsigned __int128 constant from doublewords.

Combine 2 x 18 decimal digit long long constants into a single 36 decimal digit __int128 constant. The 2 parameters are long integer constant values in high to low order. This order is consistent for big and little endian and the result loaded into vector registers is correct for quadword integer operations.

For example

```
vui128_t ten36-1 = CONST_VUINT128_Qx18d (999999999999999999UL, 999999999999999999UL);
```

7.7.6.3 CONST_VUINT128_Qx19d

```
#define CONST_VUINT128_Qx19d(
               __q0,
               __q1 )
```

Value:

```
    ( (vui128_t) \
    (((unsigned __int128) __q0) * 10000000000000000000UL) \
    + ((unsigned __int128) __q1) )
```

Generate a vector unsigned __int128 constant from doublewords.

Combine 2 x 19 decimal digit long long constants into a single 38 decimal digit __int128 constant. The 2 parameters are long integer constant values in high to low order. This order is consistent for big and little endian and the result loaded into vector registers is correct for quadword integer operations.

For example

```
const vui128_t mul_invs_ten16 = CONST_VUINT128_Qx19d(
     7662477704329444291UL, 7917351357515459181UL);
```

7.7.6.4 CONST_VUINT128_QxD

```
#define CONST_VUINT128_QxD(
               __q0,
               __q1 )
```

Value:

```
    ( (vui128_t) \
    (((unsigned __int128) __q0) « 64) \
    + ((unsigned __int128) __q1) )
```

Generate a vector unsigned __int128 constant from doublewords.

Combine 2 x 64-bit long long constants into a single __int128 constant. The 2 parameters are long integer constant values in high to low order. This order is consistent for big and little endian and the result loaded into vector registers is correct for quadword integer operations.

For example

```
vui128_t ten32 = CONST_VUINT128_QxD (0x000004ee2d6d415bUL, 0x85acef8100000000UL);
```

7.7.6.5 CONST_VUINT128_QxW

```
#define CONST_VUINT128_QxW(
                __q0,
                __q1,
                __q2,
                __q3 )
```

Value:

```
      ( (vui128_t) \
      (((unsigned __int128) __q0) « 96) \
    + (((unsigned __int128) __q1) « 64) \
    + (((unsigned __int128) __q2) « 32) \
    +  ((unsigned __int128) __q3) )
```

Generate a vector unsigned __int128 constant from words.

Combine 4 x 32-bit int constants into a single __int128 constant. The 4 parameters are integer constant values in high to low order. This order is consistent for big and little endian and the result loaded into vector registers is correct for quadword integer operations.

The effect is to compute an unsigned __int128 constant from 4 x 32-bit unsigned int constants.

```
int128 = (__q0 « 96) + (__q1 « 64) + (__q2 « 32) + q3
```

For example

```
// const for 100000000000000000000000000000000 (AKA 10**32)
vui128_t ten32 = CONST_VUINT128_QxW (0x000004ee, 0x2d6d415b,
                                     0x85acef81, 0x00000000);
```

7.7.7 Function Documentation

7.7.7.1 vec_absduq()

```
static vui128_t vec_absduq (
              vui128_t vra,
              vui128_t vrb )  [inline], [static]
```

Vector Absolute Difference Unsigned Quadword.

Compute the absolute difference of the quadwords. For each unsigned quadword, subtract VRB from VRA and return the absolute value of the difference.

processor	Latency	Throughput
power8	14	1/cycle
power9	11	1/cycle

Parameters

vra	vector of unsigned __int128

Parameters

vrb	vector of unsigned __int128

Returns

vector of the absolute difference.

7.7.7.2 vec_abssq()

```
static vi128_t vec_abssq (
            vi128_t vra )  [inline], [static]
```

Vector Absolute Value Signed Quadword.

Compute the absolute value of a signed quadwords.

processor	Latency	Throughput
power8	6-8	1/cycle
power9	9-12	1/cycle

Parameters

vra	vector of signed __int128

Returns

vector of the absolute value of vra.

7.7.7.3 vec_addcq()

```
static vui128_t vec_addcq (
            vui128_t * cout,
            vui128_t a,
            vui128_t b )  [inline], [static]
```

Vector Add with carry Unsigned Quadword.

Add two vector __int128 values and return sum and the carry out.

processor	Latency	Throughput
power8	8	1/2 cycles
power9	6	2/cycle

Parameters

∗cout	carry out from the sum of a and b.
a	128-bit vector treated a __int128.
b	128-bit vector treated a __int128.

Returns

__int128 (lower 128-bits) sum of a and b.

7.7.7.4 vec_addcuq()

```
static vui128_t vec_addcuq (
            vui128_t a,
            vui128_t b )  [inline], [static]
```

Vector Add & write Carry Unsigned Quadword.

Add two vector __int128 values and return the carry out.

processor	Latency	Throughput
power8	4	2/2 cycles
power9	3	2/cycle

Parameters

a	128-bit vector treated a __int128.
b	128-bit vector treated a __int128.

Returns

__int128 carry of the sum of a and b.

7.7.7.5 vec_addecuq()

```
static vui128_t vec_addecuq (
            vui128_t a,
            vui128_t b,
            vui128_t ci )  [inline], [static]
```

Vector Add Extended & write Carry Unsigned Quadword.

Add two vector __int128 values plus a carry-in (0|1) and return the carry out bit.

processor	Latency	Throughput
power8	4	2/2 cycles
power9	3	2/cycle

Parameters

a	128-bit vector treated a __int128.
b	128-bit vector treated a __int128.
ci	Carry-in from vector bit[127].

Returns

carry-out in bit[127] of the sum of a + b + c.

7.7.7.6 vec_addeq()

```
static vui128_t vec_addeq (
            vui128_t * cout,
            vui128_t a,
            vui128_t b,
            vui128_t ci )  [inline], [static]
```

Vector Add Extend with carry Unsigned Quadword.

Add two vector __int128 values plus a carry-in (0|1) and return sum and the carry out.

processor	Latency	Throughput
power8	8	1/2 cycles
power9	6	2/cycle

Parameters

∗*cout*	carry out from the sum of a and b.
a	128-bit vector treated a __int128.
b	128-bit vector treated a __int128.
ci	Carry-in from vector bit[127].

Returns

__int128 (lower 128-bits) sum of a + b + c.

7.7.7.7 vec_addeuqm()

```
static vui128_t vec_addeuqm (
             vui128_t a,
             vui128_t b,
             vui128_t ci )  [inline], [static]
```

Vector Add Extended Unsigned Quadword Modulo.

Add two vector __int128 values plus a carry (0|1) and return the modulo 128-bit result.

processor	Latency	Throughput
power8	4	2/2 cycles
power9	3	2/cycle

Parameters

a	128-bit vector treated a __int128.
b	128-bit vector treated a __int128.
ci	Carry-in from vector bit[127].

Returns

__int128 sum of a + b + c, modulo 128-bits.

7.7.7.8 vec_adduqm()

```
static vui128_t vec_adduqm (
             vui128_t a,
             vui128_t b )  [inline], [static]
```

Vector Add Unsigned Quadword Modulo.

Add two vector __int128 values and return result modulo 128-bits.

processor	Latency	Throughput
power8	4	2/2 cycles
power9	3	2/cycle

Parameters

a	128-bit vector treated as a __int128.
b	128-bit vector treated as a __int128.

Returns

__int128 sum of a and b.

7.7.7.9 vec_avguq()

```
static vui128_t vec_avguq (
            vui128_t vra,
            vui128_t vrb )  [inline], [static]
```

Vector Average Unsigned Quadword.

Compute the average of two unsigned quadwords as (VRA + VRB + 1) / 2.

processor	Latency	Throughput
power8	14	1/cycle
power9	11	1/cycle

Parameters

vra	vector unsigned quadwords
vrb	vector unsigned quadwords

Returns

vector of the absolute differences.

7.7.7.10 vec_clzq()

```
static vui128_t vec_clzq (
            vui128_t vra )  [inline], [static]
```

Vector Count Leading Zeros Quadword for unsigned __int128 elements.

Count leading zeros for a vector __int128 and return the count in a vector suitable for use with vector shift (left|right) and vector shift (left|right) by octet instructions.

processor	Latency	Throughput
power8	8-10	1/cycle
power9	10-12	1/cycle

Parameters

vra	a 128-bit vector treated as unsigned __int128.

Returns

a 128-bit vector with bits 121:127 containing the count of leading zeros.

7.7.7.11 vec_cmpeqsq()

```
static vb128_t vec_cmpeqsq (
             vi128_t vra,
             vi128_t vrb )  [inline], [static]
```

Vector Compare Equal Signed Quadword.

Compare signed __int128 (128-bit) integers and return all '1's, if vra == vrb, otherwise all '0's. We use vec_cmpequq as it works for both signed and unsigned compares.

processor	Latency	Throughput
power8	6	2/cycle
power9	7	2/cycle

Parameters

vra	128-bit vector treated as an signed __int128.
vrb	128-bit vector treated as an signed __int128.

Returns

128-bit vector boolean reflecting vector signed __int128 compare equal.

7.7.7.12 vec_cmpequq()

```
static vb128_t vec_cmpequq (
             vui128_t vra,
             vui128_t vrb )  [inline], [static]
```

Vector Compare Equal Unsigned Quadword.

Compare unsigned __int128 (128-bit) integers and return all '1's, if vra == vrb, otherwise all '0's.

For POWER8 (PowerISA 2.07B) or later, use the Vector Compare Equal Unsigned DoubleWord (**vcmpequd**) instruction. To get the correct quadword result, the doubleword element equal truth values are swapped, then *anded* with the original compare results. Otherwise use vector word compare and additional boolean logic to insure all word elements are equal.

processor	Latency	Throughput
power8	6	2/cycle
power9	7	2/cycle

Parameters

vra	128-bit vector treated as an unsigned __int128s.
vrb	128-bit vector treated as an unsigned __int128.

Returns

128-bit vector boolean reflecting vector unsigned __int128 compare equal.

7.7.7.13 vec_cmpgesq()

```
static vb128_t vec_cmpgesq (
              vi128_t vra,
              vi128_t vrb )  [inline], [static]
```

Vector Compare Greater Than or Equal Signed Quadword.

Compare signed __int128 (128-bit) integers and return all '1's, if vra >= vrb, otherwise all '0's.

Flip the operand sign bits and use vec_cmpgeuq for signed compare.

processor	Latency	Throughput
power8	10-16	1/ 2cycles
power9	8-14	1/cycle

Parameters

vra	128-bit vector treated as an signed __int128.
vrb	128-bit vector treated as an signed __int128.

Returns

128-bit vector boolean reflecting vector signed __int128 compare greater than.

7.7.7.14 vec_cmpgeuq()

```
static vb128_t vec_cmpgeuq (
            vui128_t vra,
            vui128_t vrb )  [inline], [static]
```

Vector Compare Greater Than or Equal Unsigned Quadword.

Compare unsigned __int128 (128-bit) integers and return all '1's, if vra >= vrb, otherwise all '0's.

For POWER8 (PowerISA 2.07B) or later, use the Vector Subtract & write Carry QuadWord (**vsubcuq**) instruction. This generates a carry for greater than or equal and NOT carry for less than. Then use vec_setb_cyq ro convert the carry into a vector bool. Here we use the pveclib implementations (vec_subcuq() and vec_setb_cyq()), instead of <altivec.h> intrinsics, to address older compilers and POWER7.

processor	Latency	Throughput
power8	8	2/ 2cycles
power9	6	2/cycle

Parameters

vra	128-bit vector treated as an unsigned __int128.
vrb	128-bit vector treated as an unsigned __int128.

Returns

128-bit vector boolean reflecting vector unsigned __int128 compare greater than.

7.7.7.15 vec_cmpgtsq()

```
static vb128_t vec_cmpgtsq (
            vi128_t vra,
            vi128_t vrb )  [inline], [static]
```

Vector Compare Greater Than Signed Quadword.

Compare signed __int128 (128-bit) integers and return all '1's, if vra > vrb, otherwise all '0's.

Flip the operand sign bits and use vec_cmpgtuq for signed compare.

processor	Latency	Throughput
power8	10-16	1/ 2cycles
power9	8-14	1/cycle

Parameters

vra	128-bit vector treated as an signed __int128.
vrb	128-bit vector treated as an signed __int128.

Returns

128-bit vector boolean reflecting vector signed __int128 compare greater than.

7.7.7.16 vec_cmpgtuq()

```
static vb128_t vec_cmpgtuq (
            vui128_t vra,
            vui128_t vrb )  [inline], [static]
```

Vector Compare Greater Than Unsigned Quadword.

Compare unsigned __int128 (128-bit) integers and return all '1's, if vra > vrb, otherwise all '0's.

For POWER8 (PowerISA 2.07B) or later, use the Vector Subtract & write Carry QuadWord (**vsubcuq**) instruction with the parameters reversed. This generates a carry for less than or equal and NOT carry for greater than. Then use vec_setb_ncq ro convert the carry into a vector bool. Here we use the pveclib implementations (vec_subcuq() and vec_setb_ncq()), instead of <altivec.h> intrinsics, to address older compilers and POWER7.

processor	Latency	Throughput
power8	8	2/ 2cycles
power9	6	2/cycle

Parameters

vra	128-bit vector treated as an unsigned __int128.
vrb	128-bit vector treated as an unsigned __int128.

Returns

128-bit vector boolean reflecting vector unsigned __int128 compare greater than.

7.7.7.17 vec_cmplesq()

```
static vb128_t vec_cmplesq (
            vi128_t vra,
            vi128_t vrb )  [inline], [static]
```

Vector Compare Less Than or Equal Signed Quadword.

Compare signed __int128 (128-bit) integers and return all '1's, if vra <= vrb, otherwise all '0's.

Flip the operand sign bits and use vec_cmpleuq for signed compare.

processor	Latency	Throughput
power8	10-16	1/ 2cycles
power9	8-14	1/cycle

Parameters

vra	128-bit vector treated as an signed __int128.
vrb	128-bit vector treated as an signed __int128.

Returns

128-bit vector boolean reflecting vector signed __int128 compare less than or equal.

7.7.7.18 vec_cmpleuq()

```
static vb128_t vec_cmpleuq (
            vui128_t vra,
            vui128_t vrb )  [inline], [static]
```

Vector Compare Less Than or Equal Unsigned Quadword.

Compare unsigned __int128 (128-bit) integers and return all '1's, if vra <= vrb, otherwise all '0's.

For POWER8 (PowerISA 2.07B) or later, use the Vector Subtract & write Carry QuadWord (**vsubcuq**) instruction. This generates a carry for greater than or equal and NOT carry for less than. Then use vec_setb_ncq ro convert the carry into a vector bool. Here we use the pveclib implementations (vec_subcuq() and vec_setb_cyq()), instead of <altivec.h> intrinsics, to address older compilers and POWER7.

processor	Latency	Throughput
power8	8	2/ 2cycles
power9	6	2/cycle

Parameters

vra	128-bit vector treated as an unsigned __int128.
vrb	128-bit vector treated as an unsigned __int128.

Returns

128-bit vector boolean reflecting vector unsigned __int128 compare less than or equal.

7.7.7.19 vec_cmpltsq()

```
static vb128_t vec_cmpltsq (
             vi128_t vra,
             vi128_t vrb )  [inline], [static]
```

Vector Compare Less Than Signed Quadword.

Compare signed __int128 (128-bit) integers and return all '1's, if vra < vrb, otherwise all '0's.

Flip the operand sign bits and use vec_cmpltuq for signed compare.

processor	Latency	Throughput
power8	10-16	1/ 2cycles
power9	8-14	1/cycle

Parameters

vra	128-bit vector treated as an signed __int128.
vrb	128-bit vector treated as an signed __int128.

Returns

128-bit vector boolean reflecting vector unsigned __int128 compare less than.

7.7.7.20 vec_cmpltuq()

```
static vb128_t vec_cmpltuq (
             vui128_t vra,
             vui128_t vrb )  [inline], [static]
```

Vector Compare Less Than Unsigned Quadword.

Compare unsigned __int128 (128-bit) integers and return all '1's, if vra < vrb, otherwise all '0's.

For POWER8 (PowerISA 2.07B) or later, use the Vector Subtract & write Carry QuadWord (**vsubcuq**) instruction. This generates a carry for greater than or equal and NOT carry for less than. Then use vec_setb_ncq ro convert the carry into a vector bool. Here we use the pveclib implementations (vec_subcuq() and vec_setb_ncq()), instead of <altivec.h> intrinsics, to address older compilers and POWER7.

processor	Latency	Throughput
power8	8	2/ 2cycles
power9	6	2/cycle

Parameters

vra	128-bit vector treated as an unsigned __int128.
vrb	128-bit vector treated as an unsigned __int128.

Returns

128-bit vector boolean reflecting vector unsigned __int128 compare less than.

7.7.7.21 vec_cmpnesq()

```
static vb128_t vec_cmpnesq (
            vi128_t vra,
            vi128_t vrb )  [inline], [static]
```

Vector Compare Equal Signed Quadword.

Compare signed __int128 (128-bit) integers and return all '1's, if vra != vrb, otherwise all '0's. We use vec_cmpequq as it works for both signed and unsigned compares.

processor	Latency	Throughput
power8	6	2/cycle
power9	7	2/cycle

Parameters

vra	128-bit vector treated as an signed __int128.
vrb	128-bit vector treated as an signed __int128.

Returns

128-bit vector boolean reflecting vector signed __int128 compare not equal.

7.7.7.22 vec_cmpneuq()

```
static vb128_t vec_cmpneuq (
            vui128_t vra,
            vui128_t vrb )  [inline], [static]
```

Vector Compare Not Equal Unsigned Quadword.

Compare unsigned __int128 (128-bit) integers and return all '1's, if vra != vrb, otherwise all '0's.

For POWER8 (PowerISA 2.07B) or later, use the Vector Compare Equal Unsigned DoubleWord (**vcmpequd**) instruction. To get the correct quadword result, the doubleword element equal truth values are swapped, then *not anded* with the original compare results. Otherwise use vector word compare and additional boolean logic to insure all word elements are equal.

processor	Latency	Throughput
power8	6	2/cycle
power9	7	2/cycle

Parameters

vra	128-bit vector treated as an unsigned __int128.
vrb	128-bit vector treated as an unsigned __int128.

Returns

128-bit vector boolean reflecting vector unsigned __int128 compare equal.

7.7.7.23 vec_cmpsq_all_eq()

```
static int vec_cmpsq_all_eq (
            vi128_t vra,
            vi128_t vrb )  [inline], [static]
```

Vector Compare all Equal Signed Quadword.

Compare vector signed __int128 values and return true if vra and vrb are equal.

processor	Latency	Throughput
power8	4-9	2/cycle
power9	3	2/cycle

Parameters

vra	128-bit vector treated as an vector signed __int128 (qword) element.
vrb	128-bit vector treated as an vector signed __int128 (qword) element.

Returns

boolean int for all 128-bits, true if equal, false otherwise.

7.7.7.24 vec_cmpsq_all_ge()

```
static int vec_cmpsq_all_ge (
            vi128_t vra,
            vi128_t vrb )  [inline], [static]
```

Vector Compare any Greater Than or Equal Signed Quadword.

Compare vector unsigned __int128 values and return true if vra >= vrb.

processor	Latency	Throughput
power8	10-15	1/ 2cycles
power9	8	1/cycle

Parameters

vra	128-bit vector treated as an vector signed __int128 (qword) element.
vrb	128-bit vector treated as an vector signed __int128 (qword) element.

Returns

boolean int for all 128-bits, true if Greater Than or Equal, false otherwise.

7.7.7.25 vec_cmpsq_all_gt()

```
static int vec_cmpsq_all_gt (
            vi128_t vra,
            vi128_t vrb )  [inline], [static]
```

Vector Compare any Greater Than Signed Quadword.

Compare vector signed __int128 values and return true if vra > vrb.

processor	Latency	Throughput
power8	10-15	1/ 2cycles
power9	8	1/cycle

Parameters

vra	128-bit vector treated as an vector signed __int128 (qword) element.
vrb	128-bit vector treated as an vector signed __int128 (qword) element.

Returns

boolean int for all 128-bits, true if Greater Than, false otherwise.

7.7.7.26 vec_cmpsq_all_le()

```
static int vec_cmpsq_all_le (
            vi128_t vra,
            vi128_t vrb )  [inline], [static]
```

Vector Compare any Less Than or Equal Signed Quadword.

Compare vector signed __int128 values and return true if vra <= vrb.

processor	Latency	Throughput
power8	10-15	1/ 2cycles
power9	8	1/cycle

Parameters

vra	128-bit vector treated as an vector signed __int128 (qword) element.
vrb	128-bit vector treated as an vector signed __int128 (qword) element.

Returns

boolean int for all 128-bits, true if Less Than or Equal, false otherwise.

7.7.7.27 vec_cmpsq_all_lt()

```
static int vec_cmpsq_all_lt (
            vi128_t vra,
            vi128_t vrb )  [inline], [static]
```

Vector Compare any Less Than Signed Quadword.

Compare vector signed __int128 values and return true if vra < vrb.

processor	Latency	Throughput
power8	10-15	1/ 2cycles
power9	8	1/cycle

Parameters

vra	128-bit vector treated as an vector signed __int128 (qword) element.
vrb	128-bit vector treated as an vector signed __int128 (qword) element.

Returns

boolean int for all 128-bits, true if Less Than, false otherwise.

7.7.7.28 vec_cmpsq_all_ne()

```
static int vec_cmpsq_all_ne (
            vi128_t vra,
            vi128_t vrb )  [inline], [static]
```

Vector Compare all Not Equal Signed Quadword.

Compare vector signed __int128 values and return true if vra and vrb are not equal.

processor	Latency	Throughput
power8	4-9	2/cycle
power9	3	2/cycle

Parameters

vra	128-bit vector treated as an vector signed __int128 (qword) element.
vrb	128-bit vector treated as an vector signed __int128 (qword) element.

Returns

boolean __int128 for all 128-bits, true if equal, false otherwise.

7.7.7.29 vec_cmpuq_all_eq()

```
static int vec_cmpuq_all_eq (
            vui128_t vra,
            vui128_t vrb )  [inline], [static]
```

Vector Compare all Equal Unsigned Quadword.

Compare vector unsigned __int128 values and return true if vra and vrb are equal.

processor	Latency	Throughput
power8	4-9	2/cycle
power9	3	2/cycle

Parameters

vra	128-bit vector treated as an vector unsigned __int128 (qword) element.
vrb	128-bit vector treated as an vector unsigned __int128 (qword) element.

Returns

boolean int for all 128-bits, true if equal, false otherwise.

7.7.7.30 vec_cmpuq_all_ge()

```
static int vec_cmpuq_all_ge (
            vui128_t vra,
            vui128_t vrb )  [inline], [static]
```

Vector Compare any Greater Than or Equal Unsigned Quadword.

Compare vector unsigned __int128 values and return true if vra >= vrb.

processor	Latency	Throughput
power8	8-13	2/ 2cycles
power9	6	2/cycle

Parameters

vra	128-bit vector treated as an vector unsigned __int128 (qword) element.
vrb	128-bit vector treated as an vector unsigned __int128 (qword) element.

Returns

boolean int for all 128-bits, true if Greater Than or Equal, false otherwise.

7.7.7.31 vec_cmpuq_all_gt()

```
static int vec_cmpuq_all_gt (
             vui128_t vra,
             vui128_t vrb )  [inline], [static]
```

Vector Compare any Greater Than Unsigned Quadword.

Compare vector unsigned __int128 values and return true if vra > vrb.

processor	Latency	Throughput
power8	8-13	2/ 2cycles
power9	6	2/cycle

Parameters

vra	128-bit vector treated as an vector unsigned __int128 (qword) element.
vrb	128-bit vector treated as an vector unsigned __int128 (qword) element.

Returns

boolean int for all 128-bits, true if Greater Than, false otherwise.

7.7.7.32 vec_cmpuq_all_le()

```
static int vec_cmpuq_all_le (
             vui128_t vra,
             vui128_t vrb )  [inline], [static]
```

Vector Compare any Less Than or Equal Unsigned Quadword.

Compare vector unsigned __int128 values and return true if vra <= vrb.

processor	Latency	Throughput
power8	8-13	2/ 2cycles
power9	6	2/cycle

Parameters

vra	128-bit vector treated as an vector unsigned __int128 (qword) element.
vrb	128-bit vector treated as an vector unsigned __int128 (qword) element.

Returns

boolean int for all 128-bits, true if Less Than or Equal, false otherwise.

7.7.7.33 vec_cmpuq_all_lt()

```
static int vec_cmpuq_all_lt (
             vui128_t vra,
             vui128_t vrb )  [inline], [static]
```

Vector Compare any Less Than Unsigned Quadword.

Compare vector unsigned __int128 values and return true if vra < vrb.

processor	Latency	Throughput
power8	8-13	2/ 2cycles
power9	6	2/cycle

Parameters

vra	128-bit vector treated as an vector unsigned __int128 (qword) element.
vrb	128-bit vector treated as an vector unsigned __int128 (qword) element.

Returns

boolean int for all 128-bits, true if Less Than, false otherwise.

7.7.7.34 vec_cmpuq_all_ne()

```
static int vec_cmpuq_all_ne (
             vui128_t vra,
             vui128_t vrb )  [inline], [static]
```

Vector Compare all Not Equal Unsigned Quadword.

Compare vector unsigned __int128 values and return true if vra and vrb are not equal.

processor	Latency	Throughput
power8	4-9	2/cycle
power9	3	2/cycle

Parameters

vra	128-bit vector treated as an vector unsigned __int128 (qword) element.
vrb	128-bit vector treated as an vector unsigned __int128 (qword) element.

Returns

boolean __int128 for all 128-bits, true if equal, false otherwise.

7.7.7.35 vec_cmul100cuq()

```
static vui128_t vec_cmul100cuq (
            vui128_t * cout,
            vui128_t a )  [inline], [static]
```

Vector combined Multiply by 100 & write Carry Unsigned Quadword.

compute the product of a 128 bit values a ∗ 100. Only the low order 128 bits of the product are returned.

processor	Latency	Throughput
power8	13-15	1/cycle
power9	6	1/cycle

Parameters

∗*cout*	pointer to upper 128-bits of the product.
a	128-bit vector treated as unsigned __int128.

Returns

vector __int128 (lower 128-bits of the 256-bit product) a ∗ 100.

7.7.7.36 vec_cmul100ecuq()

```
static vui128_t vec_cmul100ecuq (
            vui128_t * cout,
            vui128_t a,
            vui128_t cin ) [inline], [static]
```

Vector combined Multiply by 100 Extended & write Carry Unsigned Quadword.

Compute the product of a 128 bit value a ∗ 100 + digit(cin). The function return its low order 128 bits of the extended product. The first parameter (∗cout) it the address of the vector to receive the generated carry out in the range 0-99.

processor	Latency	Throughput
power8	15-17	1/cycle
power9	9	1/cycle

Parameters

∗cout	pointer to upper 128-bits of the product.
a	128-bit vector treated as unsigned __int128.
cin	values 0-99 in bits 120:127 of a vector.

Returns

vector __int128 (lower 128-bits of the 256-bit product) a ∗ 100.

7.7.7.37 vec_cmul10cuq()

```
static vui128_t vec_cmul10cuq (
            vui128_t * cout,
            vui128_t a ) [inline], [static]
```

Vector combined Multiply by 10 & write Carry Unsigned Quadword.

compute the product of a 128 bit values a ∗ 10. Only the low order 128 bits of the product are returned.

processor	Latency	Throughput
power8	13-15	1/cycle
power9	3	1/ 2cycles

Parameters

∗cout	pointer to upper 128-bits of the product.
a	128-bit vector treated as a unsigned __int128.

Returns

vector __int128 (lower 128-bits of the 256-bit product) a ∗ 10.

7.7.7.38 vec_cmul10ecuq()

```
static vui128_t vec_cmul10ecuq (
            vui128_t ∗ cout,
            vui128_t a,
            vui128_t cin )  [inline], [static]
```

Vector combined Multiply by 10 Extended & write Carry Unsigned Quadword.

Compute the product of a 128 bit value a ∗ 10 + digit(cin). Only the low order 128 bits of the extended product are returned.

processor	Latency	Throughput
power8	13-15	1/cycle
power9	3	1/ 2cycles

Parameters

∗*cout*	pointer to upper 128-bits of the product.
a	128-bit vector treated as a unsigned __int128.
cin	values 0-9 in bits 124:127 of a vector.

Returns

vector __int128 (upper 128-bits of the 256-bit product) a ∗ 10.

7.7.7.39 vec_ctzq()

```
static vui128_t vec_ctzq (
            vui128_t vra )  [inline], [static]
```

Vector Count Trailing Zeros Quadword for unsigned __int128 elements.

Count trailing zeros for a vector __int128 and return the count in a vector suitable for use with vector shift (left|right) and vector shift (left|right) by octet instructions.

processor	Latency	Throughput
power8	15-17	1/cycle
power9	13-16	1/cycle

Parameters

vra	a 128-bit vector treated as unsigned __int128.

Returns

a 128-bit vector with bits 121:127 containing the count of trailing zeros.

7.7.7.40 vec_divsq_10e31()

```
static vi128_t vec_divsq_10e31 (
            vi128_t vra )  [inline], [static]
```

Vector Divide by const 10e31 Signed Quadword.

Compute the quotient of a 128 bit values vra / 10e31.

Note

vec_divsq_10e31() and vec_modsq_10e31() can be used to prepare for **Decimal Convert From Signed Quadword** (See vec_bcdcfsq()), This guarantees that the conversion to Vector BCD does not overflow and the 39-digit extended result is obtained.

processor	Latency	Throughput
power8	18-60	1/cycle
power9	20-45	1/cycle

Parameters

vra	the dividend as a vector treated as a unsigned __int128.

Returns

the quotient as vector unsigned __int128.

7.7.7.41 vec_divudq_10e31()

```
static vui128_t vec_divudq_10e31 (
            vui128_t * qh,
            vui128_t vra,
            vui128_t vrb )  [inline], [static]
```

Vector Divide Unsigned Double Quadword by const 10e31.

Compute the quotient of 256 bit value vra||vrb / 10e31.

Note

vec_divudq_10e31() and vec_modudq_10e31() can be used to perform long division of a multi-quaqword binary value by the constant 10e31. The final remainder can be passed to **Decimal Convert From Signed Quadword** (See vec_bcdcfsq()). Long division is repeated on the resulting multi-quadword quotient to extract 31-digits for each step. This continues until the multi-quadword quotient is less than 10e31 which provides the highest order 31-digits of the of the multiple precision binary to BCD conversion.

processor	Latency	Throughput
power8	12-192	1/cycle
power9	9-127	1/cycle

Parameters

∗qh	the high quotient as a vector unsigned __int128.
vra	the high dividend as a vector unsigned __int128.
vrb	the low dividend as a vector unsigned __int128.

Returns

the low quotient as vector unsigned __int128.

7.7.7.42 vec_divudq_10e32()

```
static vui128_t vec_divudq_10e32 (
              vui128_t ∗ qh,
              vui128_t vra,
              vui128_t vrb )  [inline], [static]
```

Vector Divide Unsigned Double Quadword by const 10e32.

Compute the quotient of 256 bit value vra||vrb / 10e32.

Note

vec_divudq_10e32() and vec_modudq_10e32() can be used to perform long division of a multi-quaqword binary value by the constant 10e32. The final remainder can be passed to **Decimal Convert From Unsigned Quadword** (See vec_bcdcfuq()). Long division it repeated on the resulting multi-quadword quotient to extract 32-digits for each step. This continues until the multi-quadword quotient result is less than 10e32 which provides the highest order 32-digits of the of the multiple precision binary to BCD conversion.

processor	Latency	Throughput
power8	12-192	1/cycle
power9	9-127	1/cycle

Parameters

∗*qh*	the high quotient as a vector unsigned __int128.
vra	the high dividend as a vector unsigned __int128.
vrb	the low dividend as a vector unsigned __int128.

Returns

the low quotient as vector unsigned __int128.

7.7.7.43 vec_divuq_10e31()

```
static vui128_t vec_divuq_10e31 (
            vui128_t vra )  [inline], [static]
```

Vector Divide by const 10e31 Unsigned Quadword.

Compute the quotient of a 128 bit values vra / 10e31.

Note

vec_divuq_10e31() and vec_moduq_10e31() can be used to prepare for **Decimal Convert From Signed Quadword** (See vec_bcdcfsq()), This guarantees that the conversion to Vector BCD does not overflow and the 39-digit extended result is obtained.

processor	Latency	Throughput
power8	8-48	1/cycle
power9	9-31	1/cycle

Parameters

vra	the dividend as a vector treated as a unsigned __int128.

Returns

the quotient as vector unsigned __int128.

7.7.7.44 vec_divuq_10e32()

```
static vui128_t vec_divuq_10e32 (
             vui128_t vra )  [inline], [static]
```

Vector Divide by const 10e32 Unsigned Quadword.

Compute the quotient of a 128 bit values vra / 10e32.

Note

vec_divuq_10e32() and vec_moduq_10e32() can be used to prepare for **Decimal Convert From Unsigned Quadword** (See vec_bcdcfuq()), This guarantees that the conversion to Vector BCD does not overflow and the 39-digit extended result is obtained.

processor	Latency	Throughput
power8	8-48	1/cycle
power9	9-31	1/cycle

Parameters

vra	the dividend as a vector treated as a unsigned __int128.

Returns

the quotient as vector unsigned __int128.

7.7.7.45 vec_madd2uq()

```
static vui128_t vec_madd2uq (
             vui128_t * mulu,
             vui128_t a,
             vui128_t b,
             vui128_t c1,
             vui128_t c2 )  [inline], [static]
```

Vector Multiply-Add2 Unsigned Quadword.

Compute the sum of the 256 bit product of two 128 bit values a, b plus the sum of 128 bit values c1 and c2. The low order 128 bits of the sum are returned, while the high order 128-bits are "stored" via the mulu pointer.

Note

The advantage of this form (versus Multiply-Sum) is that the final 256 bit sum can not overflow.

processor	Latency	Throughput
power8	60-66	1/cycle
power9	30-36	1/cycle

Parameters

∗*mulu*	pointer to vector unsigned __int128 to receive the upper 128-bits of the 256 bit sum ((a ∗ b) + c1 + c2).
a	128-bit vector treated as unsigned __int128.
b	128-bit vector treated as unsigned __int128.
c1	128-bit vector treated as unsigned __int128.
c2	128-bit vector treated as unsigned __int128.

Returns

vector unsigned __int128 (lower 128-bits) of ((a ∗ b) + c1 + c2).

7.7.7.46 vec_madduq()

```
static vui128_t vec_madduq (
            vui128_t * mulu,
            vui128_t a,
            vui128_t b,
            vui128_t c )  [inline], [static]
```

Vector Multiply-Add Unsigned Quadword.

Compute the sum of the 256 bit product of two 128 bit values a, b plus the 128 bit value c. The low order 128 bits of the sum are returned, while the high order 128-bits are "stored" via the mulu pointer.

Note

The advantage of this form (versus Multiply-Sum) is that the final 256 bit sum can not overflow.

processor	Latency	Throughput
power8	56-62	1/cycle
power9	27-33	1/cycle

Parameters

∗*mulu*	pointer to vector unsigned __int128 to receive the upper 128-bits of the 256 bit sum ((a ∗ b) + c).
a	128-bit vector treated as unsigned __int128.
b	128-bit vector treated as unsigned __int128.
c	128-bit vector treated as unsigned __int128.

Returns

vector unsigned __int128 (lower 128-bits) of ((a ∗ b) + c).

7.7.7.47 vec_maxsq()

```
static vi128_t vec_maxsq (
             vi128_t vra,
             vi128_t vrb )  [inline], [static]
```

Vector Maximum Signed Quadword.

Compare Quadwords vra and vrb as signed integers and return the larger value in the result.

processor	Latency	Throughput
power8	12-18	2/cycle
power9	10-18	2/cycle

Parameters

vra	128-bit vector __int128.
vrb	128-bit vector __int128.

Returns

vector __int128 maximum of a and b.

7.7.7.48 vec_maxuq()

```
static vui128_t vec_maxuq (
             vui128_t vra,
             vui128_t vrb )  [inline], [static]
```

Vector Maximum Unsigned Quadword.

Compare Quadwords vra and vrb as unsigned integers and return the larger value in the result.

processor	Latency	Throughput
power8	10	2/cycle
power9	8	2/cycle

Parameters

vra	128-bit vector unsigned __int128.
vrb	128-bit vector unsigned __int128.

Returns

vector unsigned __int128 maximum of a and b.

7.7.7.49 vec_minsq()

```
static vi128_t vec_minsq (
             vi128_t vra,
             vi128_t vrb )  [inline], [static]
```

Vector Minimum Signed Quadword.

Compare Quadwords vra and vrb as signed integers and return the smaller value in the result.

processor	Latency	Throughput
power8	12-18	2/cycle
power9	10-18	2/cycle

Parameters

vra	128-bit vector __int128.
vrb	128-bit vector __int128.

Returns

vector __int128 minimum of a and b.

7.7.7.50 vec_minuq()

```
static vui128_t vec_minuq (
             vui128_t vra,
             vui128_t vrb )  [inline], [static]
```

Vector Minimum Unsigned Quadword.

Compare Quadwords vra and vrb as unsigned integers and return the smaller value in the result.

processor	Latency	Throughput
power8	10	2/cycle
power9	8	2/cycle

Parameters

vra	128-bit vector unsigned __int128 int.
vrb	128-bit vector unsigned __int128 int.

Returns

vector unsigned __int128 minimum of a and b.

7.7.7.51 vec_modsq_10e31()

```
static vi128_t vec_modsq_10e31 (
            vi128_t vra,
            vi128_t q )  [inline], [static]
```

Vector Modulo by const 10e31 Signed Quadword.

Compute the remainder of a 128 bit values vra % 10e31.

processor	Latency	Throughput
power8	8-52	1/cycle
power9	9-23	2/cycle

Parameters

vra	the dividend as a vector treated as a signed __int128.
q	128-bit signed __int128 containing the quotient from vec_divuq_10e31().

Returns

the remainder as vector signed __int128.

7.7.7.52 vec_modudq_10e31()

```
static vui128_t vec_modudq_10e31 (
            vui128_t vra,
            vui128_t vrb,
            vui128_t * ql )  [inline], [static]
```

Vector Modulo Unsigned Double Quadword by const 10e31.

Compute the remainder (vra||vrb) - (ql ∗ 10e31).

Note

As we are using 128-bit multiplicative inverse for 128-bit integer in a 256-bit divide, so the quotient may not be exact (one bit off). So we check here if the remainder is too high (greater than 10e31) and correct both the remainder and quotient if needed.

processor	Latency	Throughput
power8	12-124	1/cycle
power9	12-75	1/cycle

Parameters

vra	the high dividend as a vector unsigned __int128.
vrb	the low dividend as a vector unsigned __int128.
∗*ql*	128-bit unsigned __int128 containing the quotient from vec_divudq_10e31().

Returns

the remainder as vector unsigned __int128.

7.7.7.53 vec_modudq_10e32()

```
static vui128_t vec_modudq_10e32 (
            vui128_t vra,
            vui128_t vrb,
            vui128_t * ql )  [inline], [static]
```

Vector Modulo Unsigned Double Quadword by const 10e32.

Compute the remainder (vra||vrb) - (ql ∗ 10e32).

Note

As we are using 128-bit multiplicative inverse for 128-bit integer in a 256-bit divide, so the quotient may not be exact (one bit off). So we check here if the remainder is too high (greater than 10e32) and correct both the remainder and quotient if needed.

processor	Latency	Throughput
power8	12-124	1/cycle
power9	12-75	1/cycle

Parameters

vra	the high dividend as a vector unsigned __int128.
vrb	the low dividend as a vector unsigned __int128.
∗*ql*	128-bit unsigned __int128 containing the quotient from vec_divudq_10e31().

Returns

the remainder as vector unsigned __int128.

7.7.7.54 vec_moduq_10e31()

```
static vui128_t vec_moduq_10e31 (
            vui128_t vra,
            vui128_t q )  [inline], [static]
```

Vector Modulo by const 10e31 Unsigned Quadword.

Compute the remainder of a 128 bit values vra % 10e31.

processor	Latency	Throughput
power8	8-52	1/cycle
power9	9-23	2/cycle

Parameters

vra	the dividend as a vector treated as a unsigned __int128.
q	128-bit unsigned __int128 containing the quotient from vec_divuq_10e31().

Returns

the remainder as vector unsigned __int128.

7.7.7.55 vec_moduq_10e32()

```
static vui128_t vec_moduq_10e32 (
            vui128_t vra,
            vui128_t q )  [inline], [static]
```

Vector Modulo by const 10e32 Unsigned Quadword.

Compute the remainder of a 128 bit values vra % 10e32.

processor	Latency	Throughput
power8	8-52	1/cycle
power9	9-23	2/cycle

Parameters

vra	the dividend as a vector treated as a unsigned __int128.
q	128-bit unsigned __int128 containing the quotient from vec_divuq_10e32().

Returns

the remainder as vector unsigned __int128.

7.7.7.56 vec_msumcud()

```
static vui128_t vec_msumcud (
            vui64_t a,
            vui64_t b,
            vui128_t c )  [inline], [static]
```

Vector Multiply-Sum and Write Carryout Unsigned Doubleword.

Compute the even and odd 128-bit products of doubleword 64-bit element values from a, b. Then compute the carry-out of the low order 128-bits of the sum of $(a_{even} * b_{even}) + (a_{odd} * b_{odd}) + c$. Only the high order 2 bits of the 130-bit Multiply-Sum are returned and the low order 128-bits of the sum are ignored/lost. Results are in the range 0-2.

processor	Latency	Throughput
power8	30-32	1/cycle
power9	5-7	2/cycle

Parameters

a	128-bit __vector unsigned long long.
b	128-bit __vector unsigned long long.
c	128-bit __vector unsigned __int128.

Returns

The Carryout of the __vector unsigned Multiply-Sum.

7.7.7.57 vec_msumudm()

```
static vui128_t vec_msumudm (
            vui64_t a,
            vui64_t b,
            vui128_t c ) [inline], [static]
```

Vector Multiply-Sum Unsigned Doubleword Modulo.

compute the even and odd 128-bit products of doubleword 64-bit element values from a, b. Then compute the 128-bit sum $(a_{even} * b_{even}) + (a_{odd} * b_{odd}) + c$. Only the low order 128 bits of the Multiply-Sum are returned and any overflow/carry-out is ignored/lost.

processor	Latency	Throughput
power8	30-32	1/cycle
power9	5-7	2/cycle

Parameters

a	128-bit __vector unsigned long int.
b	128-bit __vector unsigned long int.
c	128-bit __vector unsigned __int128.

Returns

__vector unsigned Modulo Sum of the 128-bit even / odd products of operands a and b plus the unsigned __int128 operand c.

7.7.7.58 vec_mul10cuq()

```
static vui128_t vec_mul10cuq (
            vui128_t a ) [inline], [static]
```

Vector Multiply by 10 & write Carry Unsigned Quadword.

compute the product of a 128 bit value a $*$ 10. Only the high order 128 bits of the product are returned. This will be binary coded decimal value 0-9 in bits 124-127, Bits 0-123 will be '0'.

processor	Latency	Throughput
power8	13-15	1/cycle
power9	3	1/cycle

Parameters

a	128-bit vector treated as a unsigned __int128.

Returns

__int128 (upper 128-bits of the 256-bit product) a $*$ 10 $>>$ 128.

7.7.7.59 vec_mul10ecuq()

```
static vui128_t vec_mul10ecuq (
            vui128_t a,
            vui128_t cin )  [inline], [static]
```

Vector Multiply by 10 Extended & write Carry Unsigned Quadword.

Compute the product of a 128 bit value a $*$ 10 + digit(cin). Only the low order 128 bits of the extended product are returned.

processor	Latency	Throughput
power8	15-17	1/cycle
power9	3	1/cycle

Parameters

a	128-bit vector treated as unsigned __int128.
cin	values 0-9 in bits 124:127 of a vector.

Returns

__int128 (upper 128-bits of the 256-bit product) a $*$ 10 $>>$ 128.

7.7.7.60 vec_mul10euq()

```
static vui128_t vec_mul10euq (
            vui128_t a,
            vui128_t cin )  [inline], [static]
```

Vector Multiply by 10 Extended Unsigned Quadword.

compute the product of a 128 bit value a $*$ 10 + digit(cin). Only the low order 128 bits of the extended product are returned.

processor	Latency	Throughput
power8	13-15	1/cycle
power9	3	1/cycle

Parameters

a	128-bit vector treated as unsigned __int128.
cin	values 0-9 in bits 124:127 of a vector.

Returns

__int128 (lower 128-bits) a ∗ 10.

7.7.7.61 vec_mul10uq()

```
static vui128_t vec_mul10uq (
             vui128_t a )  [inline], [static]
```

Vector Multiply by 10 Unsigned Quadword.

compute the product of a 128 bit value a ∗ 10. Only the low order 128 bits of the product are returned.

processor	Latency	Throughput
power8	13-15	1/cycle
power9	3	1/cycle

Parameters

a	128-bit vector treated as unsigned __int128.

Returns

__int128 (lower 128-bits) a ∗ 10.

7.7.7.62 vec_muleud()

```
static vui128_t vec_muleud (
             vui64_t a,
             vui64_t b )  [inline], [static]
```

Vector Multiply Even Unsigned Doublewords.

Multiple the even 64-bit doublewords of two vector unsigned long values and return the unsigned __int128 product of the even doublewords.

Note

> The element numbering changes between big and little-endian. So the compiler and this implementation adjusts the generated code to reflect this.

processor	Latency	Throughput
power8	21-23	1/cycle
power9	8-13	2/cycle

Parameters

a	128-bit vector unsigned long int.
b	128-bit vector unsigned long int.

Returns

> vector unsigned __int128 product of the even double words of a and b.

7.7.7.63 vec_mulhud()

```
static vui64_t vec_mulhud (
            vui64_t vra,
            vui64_t vrb )  [inline], [static]
```

Vector Multiply High Unsigned Doubleword.

Multiple the corresponding doubleword elements of two vector unsigned long values and return the high order 64-bits, from each 128-bit product.

processor	Latency	Throughput
power8	28-32	1/cycle
power9	11-16	1/cycle

Note

> This operation can be used to effectively perform a divide by multiplying by the scaled multiplicative inverse (reciprocal).

Warren, Henry S. Jr and *Hacker's Delight*, 2nd Edition, Addison Wesley, 2013. Chapter 10, Integer Division by Constants.

Parameters

vra	128-bit vector unsigned long int.
vrb	128-bit vector unsigned long int.

Returns

> vector unsigned long int of the high order 64-bits of the unsigned 128-bit product of the doubleword elements from vra and vrb.

7.7.7.64 vec_mulhuq()

```
static vui128_t vec_mulhuq (
            vui128_t a,
            vui128_t b )  [inline], [static]
```

Vector Multiply High Unsigned Quadword.

compute the 256 bit product of two 128 bit values a, b. The high order 128 bits of the product are returned.

processor	Latency	Throughput
power8	56-64	1/cycle
power9	33-39	1/cycle

Parameters

a	128-bit vector treated as unsigned __int128.
b	128-bit vector treated as unsigned __int128.

Returns

> vector unsigned __int128 (upper 128-bits) of a $*$ b.

7.7.7.65 vec_mulluq()

```
static vui128_t vec_mulluq (
            vui128_t a,
            vui128_t b )  [inline], [static]
```

Vector Multiply Low Unsigned Quadword.

compute the 256 bit product of two 128 bit values a, b. Only the low order 128 bits of the product are returned.

processor	Latency	Throughput
power8	42-48	1/cycle
power9	16-20	2/cycle

Parameters

a	128-bit vector treated as unsigned __int128.
b	128-bit vector treated as unsigned __int128.

Returns

vector unsigned __int128 (lower 128-bits) a ∗ b.

7.7.7.66 vec_muloud()

```
static vui128_t vec_muloud (
            vui64_t a,
            vui64_t b )  [inline], [static]
```

Vector Multiply Odd Unsigned Doublewords.

Multiple the odd 64-bit doublewords of two vector unsigned long values and return the unsigned __int128 product of the odd doublewords.

Note

The element numbering changes between big and little-endian. So the compiler and this implementation adjusts the generated code to reflect this.

processor	Latency	Throughput
power8	21-23	1/cycle
power9	8-13	2/cycle

Parameters

a	128-bit vector unsigned long int.
b	128-bit vector unsigned long int.

Returns

vector unsigned __int128 product of the odd double words of a and b.

7.7.7.67 vec_muludm()

```
static vui64_t vec_muludm (
             vui64_t vra,
             vui64_t vrb )  [inline], [static]
```

Vector Multiply Unsigned Doubleword Modulo.

Multiple the corresponding doubleword elements of two vector unsigned long values and return the low order 64-bits of the 128-bit product for each element.

Note

vec_muludm can be used for unsigned or signed integers. It is the vector equivalent of Multiply Low Doubleword.

processor	Latency	Throughput
power8	19-28	1/cycle
power9	11-16	1/cycle

Parameters

vra	128-bit vector unsigned long long.
vrb	128-bit vector unsigned long long.

Returns

vector unsigned long long of the low order 64-bits of the unsigned 128-bit product of the doubleword elements from vra and vrb.

7.7.7.68 vec_muludq()

```
static vui128_t vec_muludq (
             vui128_t * mulu,
             vui128_t a,
             vui128_t b )  [inline], [static]
```

Vector Multiply Unsigned Double Quadword.

compute the 256 bit product of two 128 bit values a, b. The low order 128 bits of the product are returned, while the high order 128-bits are "stored" via the mulu pointer.

processor	Latency	Throughput
power8	52-56	1/cycle
power9	24-30	1/cycle

Parameters

**mulu*	pointer to vector unsigned __int128 to receive the upper 128-bits of the product.
a	128-bit vector treated as unsigned __int128.
b	128-bit vector treated as unsigned __int128.

Returns

vector unsigned __int128 (lower 128-bits) of a ∗ b.

7.7.7.69 vec_negsq()

```
static vi128_t vec_negsq (
            vi128_t int128 )  [inline], [static]
```

Vector Negate Signed Quadword.

Negate (0 - int128) the quadword.

processor	Latency	Throughput
power8	6-8	2/cycle
power9	9-12	2/cycle

Parameters

int128	a 128-bit vector treated as signed __int128.

Returns

The negative of int128.

7.7.7.70 vec_neguq()

```
static vui128_t vec_neguq (
            vui128_t int128 )  [inline], [static]
```

Vector Negate Unsigned Quadword.

Negate (0 - int128) the quadword.

processor	Latency	Throughput
power8	6-8	2/cycle
power9	9-12	2/cycle

Parameters

int128	a 128-bit vector treated as unsigned __int128.

Returns

The negative of int128.

7.7.7.71 vec_popcntq()

```
static vui128_t vec_popcntq (
             vui128_t vra )  [inline], [static]
```

Vector Population Count Quadword for unsigned __int128 elements.

Count the number of '1' bits within a vector unsigned __int128 and return the count (0-128) in a vector unsigned __int128.

processor	Latency	Throughput
power8	9-11	2/cycle
power9	9-12	2/cycle

Parameters

vra	a 128-bit vector treated as unsigned __int128.

Returns

a 128-bit vector with bits 121:127 containing the population count.

7.7.7.72 vec_revbq()

```
static vui128_t vec_revbq (
             vui128_t vra )  [inline], [static]
```

Vector Byte Reverse Quadword.

Return the bytes / octets of a 128-bit vector in reverse order.

processor	Latency	Throughput
power8	2-13	2 cycle
power9	3	2/cycle

Parameters

vra	a 128-bit vector treated as unsigned __int128.

Returns

a 128-bit vector with the bytes in reserve order.

7.7.7.73 vec_rlq()

```
static vui128_t vec_rlq (
            vui128_t vra,
            vui128_t vrb )  [inline], [static]
```

Vector Rotate Left Quadword.

Vector Rotate Left Quadword 0-127 bits. The shift amount is from bits 121-127 of vrb.

processor	Latency	Throughput
power8	10	1 cycle
power9	14	1/cycle

Parameters

vra	a 128-bit vector treated as unsigned __int128.
vrb	Shift amount in bits 121:127.

Returns

Left shifted vector.

7.7.7.74 vec_rlqi()

```
static vui128_t vec_rlqi (
            vui128_t vra,
            const unsigned int shb )  [inline], [static]
```

Vector Rotate Left Quadword Immediate.

Vector Rotate Left Quadword 0-127 bits. The shift amount is from bits 121-127 of vrb.

processor	Latency	Throughput
power8	10	1 cycle
power9	14	1/cycle

Parameters

vra	a 128-bit vector treated as unsigned __int128.
shb	Shift amount in the range 0-127.

Returns

Left shifted vector.

7.7.7.75 vec_selsq()

```
static vi128_t vec_selsq (
            vi128_t vra,
            vi128_t vrb,
            vb128_t vrc )  [inline], [static]
```

Vector Select Signed Quadword.

Return the value, (vra & ∼vrc) | (vrb & vrc).

processor	Latency	Throughput
power8	2	2/cycle
power9	3	2/cycle

Parameters

vra	a 128-bit vector treated as unsigned __int128.
vrb	a 128-bit vector treated as unsigned __int128.
vrc	a 128-bit vector treated as bool __int128.

Returns

The selected bit from vra and vrb

7.7.7.76 vec_seluq()

```
static vui128_t vec_seluq (
             vui128_t vra,
             vui128_t vrb,
             vb128_t vrc )  [inline], [static]
```

Vector Select Unsigned Quadword.

Return the value, (vra & ∼vrc) | (vrb & vrc).

processor	Latency	Throughput
power8	2	2/cycle
power9	3	2/cycle

Parameters

vra	a 128-bit vector treated as unsigned __int128.
vrb	a 128-bit vector treated as unsigned __int128.
vrc	a 128-bit vector treated as bool __int128.

Returns

The selected bit from vra and vrb

7.7.7.77 vec_setb_cyq()

```
static vb128_t vec_setb_cyq (
             vui128_t vcy )  [inline], [static]
```

Vector Set Bool from Quadword Carry.

If the vector quadword carry bit (vcy.bit[127]) is '1' then return a vector bool __int128 that is all '1's. Otherwise return all '0's.

processor	Latency	Throughput
power8	4 - 6	2/cycle
power9	3 - 5	2/cycle

Vector quadword carries are normally the result of a *write-Carry* operation. For example; vec_addcuq(), vec_addecuq(), vec_subcuq(), vec_subecuq(), vec_addcq(), vec_addeq().

Parameters

vcy	a 128-bit vector generated from a *write-Carry* operation.

Returns

a 128-bit vector bool of all '1's if the carry bit is '1'. Otherwise all '0's.

7.7.7.78 vec_setb_ncq()

```
static vb128_t vec_setb_ncq (
            vui128_t vcy ) [inline], [static]
```

Vector Set Bool from Quadword not Carry.

If the vector quadword carry bit (vcy.bit[127]) is '1' then return a vector bool __int128 that is all '0's. Otherwise return all '1's.

processor	Latency	Throughput
power8	4 - 6	2/cycle
power9	3 - 5	2/cycle

Vector quadword carries are normally the result of a *write-Carry* operation. For example; vec_addcuq(), vec_addecuq(), vec_subcuq(), vec_subecuq(), vec_addcq(), vec_addeq().

Parameters

vcy	a 128-bit vector generated from a *write-Carry* operation.

Returns

a 128-bit vector bool of all '1's if the carry bit is '0'. Otherwise all '0's.

7.7.7.79 vec_setb_sq()

```
static vb128_t vec_setb_sq (
            vi128_t vra ) [inline], [static]
```

Vector Set Bool from Signed Quadword.

If the quadword's sign bit is '1' then return a vector bool __int128 that is all '1's. Otherwise return all '0's.

processor	Latency	Throughput
power8	4 - 6	2/cycle
power9	5 - 8	2/cycle

Parameters

vra	a 128-bit vector treated as signed __int128.

Returns

a 128-bit vector bool of all '1's if the sign bit is '1'. Otherwise all '0's.

7.7.7.80 vec_sldq()

```
static vui128_t vec_sldq (
             vui128_t vrw,
             vui128_t vrx,
             vui128_t vrb )  [inline], [static]
```

Vector Shift Left Double Quadword.

Vector Shift Left double Quadword 0-127 bits. Return a vector __int128 that is the left most 128-bits after shifting left 0-127-bits of the 256-bit double vector (vrw||vrx). The shift amount is from bits 121:127 of vrb.

processor	Latency	Throughput
power8	10	1 cycle
power9	14	1/cycle

Parameters

vrw	upper 128-bits of the 256-bit double vector.
vrx	lower 128-bits of the 256-bit double vector.
vrb	Shift amount in bits 121:127.

Returns

high 128-bits of left shifted double vector.

7.7.7.81 vec_sldqi()

```
static vui128_t vec_sldqi (
             vui128_t vrw,
             vui128_t vrx,
             const unsigned int shb )  [inline], [static]
```

Vector Shift Left Double Quadword Immediate.

Vector Shift Left double Quadword 0-127 bits. Return a vector __int128 that is the left most 128-bits after shifting left 0-127-bits of the 256-bit double vector (vrw||vrx). The shift amount is from bits 121:127 of vrb.

processor	Latency	Throughput
power8	10	1 cycle
power9	14	1/cycle

Parameters

vrw	upper 128-bits of the 256-bit double vector.
vrx	lower 128-bits of the 256-bit double vector.
shb	Shift amount in the range 0-127.

Returns

high 128-bits of left shifted double vector.

7.7.7.82 vec_slq()

```
static vui128_t vec_slq (
             vui128_t vra,
             vui128_t vrb )  [inline], [static]
```

Vector Shift Left Quadword.

Vector Shift Left Quadword 0-127 bits. The shift amount is from bits 121-127 of vrb.

processor	Latency	Throughput
power8	4	1/cycle
power9	6	1/cycle

Parameters

vra	a 128-bit vector treated as unsigned __int128.
vrb	Shift amount in bits 121:127.

Returns

Left shifted vector.

7.7.7.83 vec_slq4()

```
static vui128_t vec_slq4 (
            vui128_t vra )  [inline], [static]
```

Deprecated Vector Shift Left 4-bits Quadword. Replaced by vec_slqi with shb param = 4.

Vector Shift Left Quadword 0-127 bits. The shift amount is from bits 121-127 of vrb.

Parameters

vra	a 128-bit vector treated a __int128.

Returns

Left shifted vector.

7.7.7.84 vec_slq5()

```
static vui128_t vec_slq5 (
            vui128_t vra )  [inline], [static]
```

Deprecated Vector Shift Left 5-bits Quadword. Replaced by vec_slqi with shb param = 5.

Vector Shift Left Quadword 0-127 bits. The shift amount is from bits 121-127 of vrb.

```
 @param vra a 128-bit vector treated a __int128.
 @return Left shifted vector.
```

7.7.7.85 vec_slqi()

```
static vui128_t vec_slqi (
            vui128_t vra,
            const unsigned int shb )  [inline], [static]
```

Vector Shift Left Quadword Immediate.

Shift left Quadword 0-127 bits. The shift amount is a const unsigned int in the range 0-127. A shift count of 0 returns the original value of vra. Shift counts greater then 127 bits return zero.

processor	Latency	Throughput
power8	2-13	2 cycle
power9	3-15	2/cycle

Parameters

vra	a 128-bit vector treated as unsigned __int128.
shb	Shift amount in the range 0-127.

Returns

128-bit vector shifted left shb bits.

7.7.7.86 vec_splat_s128()

```
static vi128_t vec_splat_s128 (
            const int sim )  [inline], [static]
```

Vector Splat Immediate Signed Quadword. Extend a signed integer constant across the quadword element of the result. This is the quadword equivalent of Vector Splat Immediate Signed (Byte | Halfword |Word).

Note

POWER9/10 will generate the instruction xxspltib for byte values -128 to 128. But the ISA does not have vextsb2q instructions (so far). So we need to sign extent the byte value using a const quadword (0/-1 depending on the sign) and vsldoi. POWER8 (and earlier) does not have xxspltib but does have vspltisw. For a smaller range (-16 -> 15) POWER8 can use the sequence vec_splat_s8(sim)/vec_splat_s8(0/-1)/vec_sld. Larger values will be loaded as a quadword constant from the read-only data (.rodata) section.

processor	Latency	Throughput
power8	4 - 9	1/cycle
power9	5 - 9	2/cycle

Parameters

sim	a small signed integer const.

Returns

Vector with sim value extended to quadword.

7.7.7.87 vec_splat_u128()

```
static vui128_t vec_splat_u128 (
             const int sim )  [inline], [static]
```

Vector Splat Immediate Unsigned Quadword. Extend a unsigned integer constant across the quadword element of the result. This is the quadword equivalent of Vector Splat Immediate Unsigned (Byte | Halfword |Word).

Note

POWER9/10 will generate the instruction xxspltib for byte values 0 to 255. Then we need to sign extent the byte value using a const quadword 0 and vsldoi. POWER8 (and earlier) does not have xxspltib but does have vspltisw. For a smaller range (0 -> 15) POWER8 can use the sequence vec_splat_s8(sim)/vec_splat_s8(0)/vec_sld. Larger values will be loaded as a quadword constant from the read-only data (.rodata) section.

processor	Latency	Throughput
power8	4 - 9	1/cycle
power9	5 - 9	2/cycle

Parameters

sim	a small unsigned integer const.

Returns

Vector with sim value extended to quadword.

7.7.7.88 vec_sraq()

```
static vi128_t vec_sraq (
             vi128_t vra,
             vui128_t vrb )  [inline], [static]
```

Vector Shift Right Algebraic Quadword.

Vector Shift Right Algebraic Quadword 0-127 bits. The shift amount is from bits 121-127 of vrb.

processor	Latency	Throughput
power8	10	1 cycle
power9	14	1/cycle

Parameters

vra	a 128-bit vector treated as signed __int128.
vrb	Shift amount in bits 121:127.

Returns

Right algebraic shifted vector.

7.7.7.89 vec_sraqi()

```
static vi128_t vec_sraqi (
            vi128_t vra,
            const unsigned int shb ) [inline], [static]
```

Vector Shift Right Algebraic Quadword Immediate.

Vector Shift Right Algebraic Quadword 0-127 bits. The shift amount is from bits 121-127 of vrb.

processor	Latency	Throughput
power8	6-15	1 cycle
power9	9-18	1/cycle

Note

vec_sraqi optimizes for some special cases. For shift by octet (multiple of 8 bits) use vec_setb_sq () to extend sign then vector shift left double by octet immediate by (16 - (shb / 8)) to effect the right octet shift. For _ARCH_PWR8 and shifts less than 64 bits, use both vec_srqi () and vector shift right algebraic doubleword. Then use vec_pasted () to combine the high 64-bits from vec_sradi () and the low 64-bits from vec_srqi ().

Parameters

vra	a 128-bit vector treated as signed __int128.
shb	Shift amount in the range 0-127.

Returns

Right algebraic shifted vector.

7.7.7.90 vec_srq()

```
static vui128_t vec_srq (
             vui128_t vra,
             vui128_t vrb )  [inline], [static]
```

Vector Shift Right Quadword.

Vector Shift Right Quadword 0-127 bits. The shift amount is from bits 121-127 of vrb.

processor	Latency	Throughput
power8	4	1/cycle
power9	6	1/cycle

Parameters

vra	a 128-bit vector treated as unsigned __int128.
vrb	Shift amount in bits 121:127.

Returns

Right shifted vector.

7.7.7.91 vec_srq4()

```
static vui128_t vec_srq4 (
             vui128_t vra )  [inline], [static]
```

Deprecated Vector Shift right 4-bits Quadword. Replaced by vec_srqi with shb param = 4.

Vector Shift Right Quadword 0-127 bits. The shift amount is from bits 121-127 of vrb.

Parameters

vra	a 128-bit vector treated as a __int128.

Returns

Right shifted vector.

7.7.7.92 vec_srq5()

```
static vui128_t vec_srq5 (
            vui128_t vra )  [inline], [static]
```

Deprecated Vector Shift right 5-bits Quadword. Replaced by vec_srqi with shb param = 5.

Vector Shift Right Quadword 0-127 bits. The shift amount is from bits 121-127 of vrb.

Parameters

vra	a 128-bit vector treated a __int128.

Returns

Right shifted vector.

7.7.7.93 vec_srqi()

```
static vui128_t vec_srqi (
            vui128_t vra,
            const unsigned int shb )  [inline], [static]
```

Vector Shift Right Quadword Immediate.

Shift right Quadword 0-127 bits. The shift amount is a const unsigned int in the range 0-127. A shift count of 0 returns the original value of vra. Shift counts greater then 127 bits return zero.

processor	Latency	Throughput
power8	2-13	2 cycle
power9	3-15	2/cycle

Parameters

vra	a 128-bit vector treated as unsigned __int128.
shb	Shift amount in the range 0-127.

Returns

128-bit vector shifted right shb bits.

7.7.7.94 vec_subcuq()

```
static vui128_t vec_subcuq (
            vui128_t vra,
            vui128_t vrb )  [inline], [static]
```

Vector Subtract and Write Carry Unsigned Quadword.

Generate the carry-out of the sum (vra + NOT(vrb) + 1).

processor	Latency	Throughput
power8	4	2/2 cycles
power9	3	2/cycle

Parameters

vra	128-bit vector treated as unsigned __int128.
vrb	128-bit vector treated as unsigned __int128.

Returns

__int128 carry from the unsigned difference vra - vrb.

7.7.7.95 vec_subecuq()

```
static vui128_t vec_subecuq (
            vui128_t vra,
            vui128_t vrb,
            vui128_t vrc )  [inline], [static]
```

Vector Subtract Extended and Write Carry Unsigned Quadword.

Generate the carry-out of the sum (vra + NOT(vrb) + vrc.bit[127]).

processor	Latency	Throughput
power8	4	2/2 cycles
power9	3	2/cycle

Parameters

vra	128-bit vector treated as unsigned __int128.
vrb	128-bit vector treated as unsigned __int128.
vrc	128-bit vector carry-in from bit 127.

Returns

__int128 carry from the extended __int128 difference.

7.7.7.96 vec_subeuqm()

```
static vui128_t vec_subeuqm (
             vui128_t vra,
             vui128_t vrb,
             vui128_t vrc ) [inline], [static]
```

Vector Subtract Extended Unsigned Quadword Modulo.

Subtract two vector __int128 values and return result modulo 128-bits.

processor	Latency	Throughput
power8	4	2/2 cycles
power9	3	2/cycle

Parameters

vra	128-bit vector treated as unsigned __int128.
vrb	128-bit vector treated as unsigned __int128.
vrc	128-bit vector carry-in from bit 127.

Returns

__int128 unsigned difference of vra minus vrb.

7.7.7.97 vec_subuqm()

```
static vui128_t vec_subuqm (
             vui128_t vra,
             vui128_t vrb ) [inline], [static]
```

Vector Subtract Unsigned Quadword Modulo.

Subtract two vector __int128 values and return result modulo 128-bits.

processor	Latency	Throughput
power8	4	2/2 cycles
power9	3	2/cycle

Parameters

vra	128-bit vector treated as unsigned __int128.
vrb	128-bit vector treated as unsigned __int128.

Returns

__int128 unsigned difference of vra minus vrb.

7.7.7.98 vec_vmadd2eud()

```
static vui128_t vec_vmadd2eud (
            vui64_t a,
            vui64_t b,
            vui64_t c,
            vui64_t d )  [inline], [static]
```

Vector Multiply-Add2 Even Unsigned Doublewords.

Multiply the even 64-bit doublewords of vector unsigned long values (a $*$ b) and return sum of the unsigned __int128 product and the even doublewords of c and d (($a_{even} * b_{even}) + c_{even} + d_{even}$).

Note

The advantage of this form (versus Multiply-Sum) is that the final 128 bit sum can not overflow.

This implementation is NOT endian sensitive and the function is stable across BE/LE implementations.

processor	Latency	Throughput
power8	25-28	1/cycle
power9	13-18	2/cycle

Parameters

a	128-bit vector unsigned long int.
b	128-bit vector unsigned long int.
c	128-bit vector unsigned long int.
d	128-bit vector unsigned long int.

Returns

vector unsigned __int128 sum ($a_{even} * b_{even}) + c_{even} + d_{even}$.

7.7.7.99 vec_vmadd2oud()

```
static vui128_t vec_vmadd2oud (
            vui64_t a,
            vui64_t b,
            vui64_t c,
            vui64_t d )  [inline], [static]
```

Vector Multiply-Add2 Odd Unsigned Doublewords.

Multiply the odd 64-bit doublewords of two vector unsigned long values (a $*$ b) and return the sum of the unsigned __int128 product and the odd doublewords of c and d (($a_{odd} * b_{odd}$) + c_{odd} + d_{odd}).

Note

The advantage of this form (versus Multiply-Sum) is that the final 128 bit sum can not overflow.

This implementation is NOT endian sensitive and the function is stable across BE/LE implementations.

processor	Latency	Throughput
power8	25-28	1/cycle
power9	13-18	2/cycle

Parameters

a	128-bit vector unsigned long int.
b	128-bit vector unsigned long int.
c	128-bit vector unsigned long int.
d	128-bit vector unsigned long int.

Returns

vector unsigned __int128 sum ($a_{odd} * b_{odd}$) + c_{odd} + d_{odd}.

7.7.7.100 vec_vmaddeud()

```
static vui128_t vec_vmaddeud (
            vui64_t a,
            vui64_t b,
            vui64_t c )  [inline], [static]
```

Vector Multiply-Add Even Unsigned Doublewords.

Multiply the even 64-bit doublewords of vector unsigned long values (a $*$ b) and return sum of the unsigned __int128 product and the even doubleword of c ($a_{even} * b_{even}$) + c_{even}.

Note

The advantage of this form (versus Multiply-Sum) is that the final 128 bit sum can not overflow.

This implementation is NOT endian sensitive and the function is stable across BE/LE implementations.

processor	Latency	Throughput
power8	25-28	1/cycle
power9	10-13	2/cycle

Parameters

a	128-bit vector unsigned long int.
b	128-bit vector unsigned long int.
c	128-bit vector unsigned long int.

Returns

vector unsigned __int128 sum ($a_{even} * b_{even}) + c_{even}$.

7.7.7.101 vec_vmaddoud()

```
static vui128_t vec_vmaddoud (
            vui64_t a,
            vui64_t b,
            vui64_t c )  [inline], [static]
```

Vector Multiply-Add Odd Unsigned Doublewords.

Multiply the odd 64-bit doublewords of two vector unsigned long values (a $*$ b) and return the sum of the unsigned __int128 product and the odd doubleword of c ($a_{odd} * b_{odd}) + c_{odd}$.

Note

The advantage of this form (versus Multiply-Sum) is that the final 128 bit sum can not overflow.

This implementation is NOT endian sensitive and the function is stable across BE/LE implementations.

processor	Latency	Throughput
power8	25-28	1/cycle
power9	10-13	2/cycle

Parameters

a	128-bit vector unsigned long int.
b	128-bit vector unsigned long int.
c	128-bit vector unsigned long int.

Returns

vector unsigned __int128 sum ($a_{odd} * b_{odd}$) + c_{odd}.

7.7.7.102 vec_vmsumeud()

```
static vui128_t vec_vmsumeud (
            vui64_t a,
            vui64_t b,
            vui128_t c ) [inline], [static]
```

Vector Multiply-Sum Even Unsigned Doublewords.

Multiply the even 64-bit doublewords of vector unsigned long values (a ∗ b) and return sum of the unsigned __int128 product and c ($a_{even} * b_{even}$) + c.

Note

This form (Multiply-Sum) can overflow the final 128 bit sum, unless the addend (c) is restricted to (**INT64_MAX** ∗ 2) or less.

This implementation is NOT endian sensitive and the function is stable across BE/LE implementations.

processor	Latency	Throughput
power8	25-28	1/cycle
power9	10-13	2/cycle

Parameters

a	128-bit vector unsigned long int.
b	128-bit vector unsigned long int.
c	128-bit vector unsigned __int128.

Returns

vector unsigned __int128 sum ($a_{even} * b_{even}$) + c.

7.7.7.103 vec_vmsumoud()

```
static vui128_t vec_vmsumoud (
            vui64_t a,
            vui64_t b,
            vui128_t c )  [inline], [static]
```

Vector Multiply-Sum Odd Unsigned Doublewords.

Multiply the odd 64-bit doublewords of two vector unsigned long values (a ∗ b) and return the sum of the unsigned __int128 product and variable c ($a_{odd} * b_{odd}$) + c>.

Note

> This form (Multiply-Sum) can overflow the final 128 bit sum, unless the addend (c) is restricted to (**INT64_MAX** ∗ 2) or less.
>
> This implementation is NOT endian sensitive and the function is stable across BE/LE implementations.

processor	Latency	Throughput
power8	25-28	1/cycle
power9	10-13	2/cycle

Parameters

a	128-bit vector unsigned long int.
b	128-bit vector unsigned long int.
c	128-bit vector unsigned __int128.

Returns

> vector unsigned __int128 sum ($a_{odd} * b_{odd}$) + c.

7.7.7.104 vec_vmuleud()

```
static vui128_t vec_vmuleud (
            vui64_t a,
            vui64_t b )  [inline], [static]
```

Vector Multiply Even Unsigned Doublewords.

Multiply the even 64-bit doublewords of two vector unsigned long values and return the unsigned __int128 product of the even doublewords.

Note

> This function implements the operation of a Vector Multiply Even Doubleword instruction, as if the PowerISA included such an instruction. This implementation is NOT endian sensitive and the function is stable across BE/LE implementations.

processor	Latency	Throughput
power8	21-23	1/cycle
power9	8-11	2/cycle

Parameters

a	128-bit vector unsigned long int.
b	128-bit vector unsigned long int.

Returns

vector unsigned __int128 product of the even double words of a and b.

7.7.7.105 vec_vmuloud()

```
static vui128_t vec_vmuloud (
            vui64_t a,
            vui64_t b )  [inline], [static]
```

Vector Multiply Odd Unsigned Doublewords.

Multiply the odd 64-bit doublewords of two vector unsigned long values and return the unsigned __int128 product of the odd doublewords.

Note

This function implements the operation of a Vector Multiply Odd Doubleword instruction, as if the PowerISA included such an instruction. This implementation is NOT endian sensitive and the function is stable across BE/LE implementations.

processor	Latency	Throughput
power8	21-23	1/cycle
power9	8-13	2/cycle

Parameters

a	128-bit vector unsigned long int.
b	128-bit vector unsigned long int.

Returns

vector unsigned __int128 product of the odd double words of a and b.

7.7.7.106 vec_vsldbi()

```
static vui128_t vec_vsldbi (
              vui128_t vra,
              vui128_t vrb,
              const unsigned int shb )  [inline], [static]
```

Vector Shift Left Double Quadword by Bit Immediate.

Return a vector __int128 that is bits shb:shb+127 from the (256-bit) double quadword (vra || vrb). The shift amount is constant immediate value in the range 0-7.

processor	Latency	Throughput
power8	8	1 cycle
power9	11	1/cycle

Parameters

vra	upper 128-bits of the 256-bit double quadword vector.
vrb	lower 128-bits of the 256-bit double quadword vector.
shb	Shift amount in the range 0-7.

Returns

128-bits from bits shb:shb+127.

7.7.7.107 vec_vsrdbi()

```
static vui128_t vec_vsrdbi (
              vui128_t vra,
              vui128_t vrb,
              const unsigned int shb )  [inline], [static]
```

Vector Shift Right Double Quadword by Bit Immediate.

Return a vector __int128 that is bits 128-shb:255-shb from the (256-bit) double quadword (vra || vrb). The shift amount is constant immediate value in the range 0-7.

processor	Latency	Throughput
power8	8	1 cycle
power9	11	1/cycle

Parameters

vra	upper 128-bits of the 256-bit double quadword vector.
vrb	lower 128-bits of the 256-bit double quadword vector.
shb	Shift amount in the range 0-7.

Returns

128-bits from bits 128-shb:255-shb.

7.8 src/pveclib/vec_int16_ppc.h File Reference

Header package containing a collection of 128-bit SIMD operations over 16-bit integer elements.

```
#include <pveclib/vec_char_ppc.h>
```

Functions

- static vui16_t vec_absduh (vui16_t vra, vui16_t vrb)

  Vector Absolute Difference Unsigned halfword.
- static vui16_t vec_clzh (vui16_t vra)

  Vector Count Leading Zeros Halfword for unsigned short elements.
- static vui16_t vec_ctzh (vui16_t vra)

  Vector Count Trailing Zeros Halfword for unsigned short elements.
- static vui16_t vec_mrgahh (vui32_t vra, vui32_t vrb)

  Vector Merge Algebraic High Halfword operation.
- static vui16_t vec_mrgalh (vui32_t vra, vui32_t vrb)

  Vector Merge Algebraic Low Halfword operation.
- static vui16_t vec_mrgeh (vui16_t vra, vui16_t vrb)

  Vector Merge Even Halfwords operation.
- static vui16_t vec_mrgoh (vui16_t vra, vui16_t vrb)

  Vector Merge Odd Halfwords operation.
- static vi16_t vec_mulhsh (vi16_t vra, vi16_t vrb)

  Vector Multiply High Signed halfword.
- static vui16_t vec_mulhuh (vui16_t vra, vui16_t vrb)

  Vector Multiply High Unsigned halfword.
- static vui16_t vec_muluhm (vui16_t vra, vui16_t vrb)

  Vector Multiply Unsigned halfword Modulo.
- static vui16_t vec_popcnth (vui16_t vra)

  Vector Population Count halfword.
- static vui16_t vec_revbh (vui16_t vra)

  byte reverse each halfword of a vector unsigned short.
- static vb16_t vec_setb_sh (vi16_t vra)

  Vector Set Bool from Signed Halfword.

- static vui16_t vec_slhi (vui16_t vra, const unsigned int shb)

  Vector Shift left Halfword Immediate.
- static vui16_t vec_srhi (vui16_t vra, const unsigned int shb)

  Vector Shift Right Halfword Immediate.
- static vi16_t vec_srahi (vi16_t vra, const unsigned int shb)

  Vector Shift Right Algebraic Halfword Immediate.
- static vui32_t vec_vmaddeuh (vui16_t a, vui16_t b, vui16_t c)

  Vector Multiply-Add Even Unsigned Halfwords.
- static vui32_t vec_vmaddouh (vui16_t a, vui16_t b, vui16_t c)

  Vector Multiply-Add Odd Unsigned Halfwords.
- static vui16_t vec_vmrgeh (vui16_t vra, vui16_t vrb)

  Vector Merge Even Halfwords.
- static vui16_t vec_vmrgoh (vui16_t vra, vui16_t vrb)

  Vector Merge Odd Halfwords.

7.8.1 Detailed Description

Header package containing a collection of 128-bit SIMD operations over 16-bit integer elements.

Most of these operations are implemented in a single instruction on newer (POWER6/POWER7POWER8/POWER9) processors. This header serves to fill in functional gaps for older (POWER7, POWER8) processors and provides a in-line assembler implementation for older compilers that do not provide the build-ins.

Most vector short (16-bit integer halfword) operations are implemented with PowerISA VMX instructions either defined by the original VMX (AKA Altivec) or added to later versions of the PowerISA. PowerISA 2.07B (POWER8) added several useful halfword operations (count leading zeros, population count) not included in the original VMX. PowerISA 3.0B (P↩ OWER9) adds several more (absolute difference, compare not equal, count trailing zeros, extend sign, extract/insert, and reverse bytes). Most of these intrinsic (compiler built-ins) operations are defined in <altivec.h> and described in the compiler documentation.

Note

> The compiler disables associated <altivec.h> built-ins if the **mcpu** target does not enable the specific instruction. For example if you compile with **-mcpu=power7**, vec_vclz and vec_vclzh will not be defined. Another example if you compile with **-mcpu=power8**, vec_revb will not be defined. But vec_vclzh and vec_revbh is always defined in this header. This header provides the appropriate substitutions, will generate the minimum code, appropriate for the target, and produce correct results.
>
> Most ppc64le compilers will default to -mcpu=power8 if not specified.

This header covers operations that are either:

- Implemented in hardware instructions for later processors and useful to programmers, on slightly older processors, even if the equivalent function requires more instructions. Examples include Count Leading Zeros, Population Count and Byte Reverse.
- Defined in the OpenPOWER ABI but *not* yet defined in <altivec.h> provided by available compilers in common use. Examples include Count Leading Zeros, Population Count and Byte Reverse.
- Commonly used operations, not covered by the ABI or <altivec.h>, and require multiple instructions or are not obvious. Examples include the multiply-add, multiply-high and shift immediate operations.

7.8.2 Recent Additions

Added vec_vmaddeuh() and vec_vmaddouh() as an optimization for the vector multiply quadword implementations on POWER7.

7.8.3 Endian problems with halfword operations

It would be useful to provide a vector multiply high halfword (return the high order 16-bits of the 32-bit product) operation. This can be used for multiplicative inverse (effectively integer divide) operations. Neither integer multiply high nor divide are available as vector instructions. However the multiply high halfword operation can be composed from the existing multiply even/odd halfword operations followed by the vector merge even halfword operation. Similarly a multiply low (modulo) halfword operation can be composed from the existing multiply even/odd halfword operations followed by the vector merge odd halfword operation.

As a prerequisite we need to provide the merge even/odd halfword operations. While PowerISA has added these operations for word and doubleword, instructions are nor defined for byte and halfword. Fortunately vector merge operations are just a special case of vector permute. So the vec_vmrgoh() and vec_vmrgeh() implementation can use vec_perm and appropriate selection vectors to provide these merge operations.

But this is complicated by *little-endian* (LE) support as specified in the OpenPOWER ABI and as implemented in the compilers. Little-endian changes the effective vector element numbering and the location of even and odd elements. This means that the vector built-ins provided by altivec.h may not generate the instructions you would expect.

See also

General Endian Issues

Endian problems with word operations

The OpenPOWER ABI provides a helpful table of `Endian Sensitive Operations`. For for vec_mule (vmuleuh, vmulesh):

> Replace with vmulouh and so on, for LE.

For for vec_mulo (vmulouh, vmulosh):

> Replace with vmuleuh and so on, for LE.

Also for vec_perm (vperm) it specifies:

> For LE, Swap input arguments and complement the selection vector.

The above is just a sampling of a larger list of Endian Sensitive Operations.

The obvious coding for Vector Multiply High Halfword would be:

```
vui16_t
test_mulhw (vui16_t vra, vui16_t vrb)
{
  return vec_mergee ((vui16_t)vec_mule (vra, vrb),
                     (vui16_t)vec_mulo (vra, vrb));
}
```

A couple problems with this:

- vec_mergee is only defined for vector int/long and float/double (word/doubleword) types.
- vec_mergee is endian sensitive and would produce the wrong results in LE mode.
- vec_mule/vec_mulo are endian sensitive and produce the wrong results in LE mode.

The first step is to implement Vector Merge Even Halfword operation:

```
static inline vui16_t
vec_vmrgeh (vui16_t vra, vui16_t vrb)
{
#if __BYTE_ORDER__ == __ORDER_LITTLE_ENDIAN__
  vui16_t permute =
      { 0x0302,0x1312, 0x0706,0x1716, 0x0B0A,0x1B1A, 0x0F0E,0x1F1E };
  return vec_perm (vrb, vra, (vui8_t)permute);
#else
  vui16_t permute =
      { 0x0001,0x1011, 0x0405,0x1415, 0x0809,0x1819, 0x0C0D,0x1C1D};
  return vec_perm (vra, vrb, (vui8_t)permute);
#endif
}
```

For big-endian we have a straight forward vec_perm with a permute select vector interleaving even halfwords from vectors vra and vrb.

For little-endian we need to nullify the LE transform applied by the compiler. So the select vector looks like it interleaves odd halfwords from vectors vrb and vra. It also reverses byte numbering within halfwords. The compiler transforms this back into the operation we wanted in the first place. The result is *not* endian sensitive and is stable across BE/LE implementations. Similarly for the Vector Merge Odd Halfword operation.

As good OpenPOWER ABI citizens we should also provide endian sensitive operations vec_mrgeh() vec_mrgoh(). For example:

```
static inline vui16_t
vec_mrgeh  (vui16_t vra, vui16_t vrb)
{
#if __BYTE_ORDER__ == __ORDER_LITTLE_ENDIAN__
  return vec_vmrgoh ((vui16_t) vrb, (vui16_t) vra);
#else
  return vec_vmrgeh ((vui16_t) vra, (vui16_t) vrb);
#endif
}
```

Note

This is essentially what the compiler would do for vec_mergee.

Also to follow that pattern established for vec_int32_ppc.h we should provide implementations for Vector Merge Algebraic High/Low Halfword. For example:

```
static inline vui16_t
vec_mrgahh  (vui32_t vra, vui32_t vrb)
{
  return vec_vmrgeh ((vui16_t) vra, (vui16_t) vrb);
}
```

This is simpler as we can use the endian invariant vec_vmrgeh() operation. Similarly for Vector Merge Algebraic Low Halfword using vec_vmrgoh().

Note

The inputs are defined as 32-bit to match the results from multiply even/odd halfword.

Now we have all the parts we need to implement multiply high/low halfword. For example Multiply High Unsigned Halfword:

```
static inline vui16_t
vec_mulhuh (vui16_t vra, vui16_t vrb)
{
#if __BYTE_ORDER__ == __ORDER_LITTLE_ENDIAN__
  return vec_mrgahh (vec_mulo (vra, vrb), vec_mule (vra, vrb));
#else
  return vec_mrgahh (vec_mule (vra, vrb), vec_mulo (vra, vrb));
#endif
}
```

Similarly for Multiply High Signed Halfword.

Note

For LE we need to nullify the compiler transform by reversing of the order of vec_mulo/vec_mule. This is required to get the algebraically correct (multiply high) result.

Finally we can implement the Multiply Low Halfword which by PowerISA conventions is called Multiply Unsigned Halfword Modulo:

```
static inline vui16_t
vec_muluhm (vui16_t vra, vui16_t vrb)
{
#if __BYTE_ORDER__ == __ORDER_LITTLE_ENDIAN__
  return vec_mrgalh (vec_mulo (vra, vrb), vec_mule (vra, vrb));
#else
  return vec_mrgalh (vec_mule (vra, vrb), vec_mulo (vra, vrb));
#endif
}
```

Note

We use the endian stable vec_mrgalh() for multiply low. Again for LE we have to nullify the compiler transform by reversing of the order of vec_mulo/vec_mule. This is required to get the algebraically correct (multiply high) result.

vec_muluhm() works for signed and unsigned multiply low (modulo).

7.8.3.1 Multiply High Unsigned Halfword Example

So what does the compiler generate after unwinding three levels of inline functions. For this test case:

```
vui16_t
__test_mulhuh (vui16_t a, vui16_t b)
{
  return vec_mulhuh (a, b);
}
```

The GCC 8 compiler targeting powerpc64le and -mcpu=power8 generates:

```
addis   r9,r2,.rodata.cst16@ha
vmulouh v1,v2,v3
vmuleuh v2,v2,v3
addi    r9,r9,.rodata.cst16@l
lvx     v0,0,r9
xxlnor  vs32,vs32,vs32
vperm   v2,v2,v1,v0
```

The addis, addi, lvx instruction sequence loads the permute selection constant vector. The xxlnor instruction complements the selection vector for LE. These instructions are only needed once per function and can be hoisted out of loops and shared across instances of vec_mulhuh(). Which might look like this:

```
        addis   r9,r2,.rodata.cst16@ha
        addi    r9,r9,.rodata.cst16@l
        lvx     v0,0,r9
        xxlnor  vs32,vs32,vs32
        ...
Loop:
        vmulouh v1,v2,v3
        vmuleuh v2,v2,v3
        vperm   v2,v2,v1,v0
        ...
```

The vmulouh, vmuleuh, vperm instruction sequence is the core of the function. They multiply the elements and selects/merges the high order 16-bits of each product into the result vector.

7.8.4 Examples, Divide by integer constant

Suppose we have a requirement to convert an array of 16-bit unsigned short values to decimal. The classic *itoa* implementation performs a sequence of divide / modulo by 10 operations that produce one (decimal) value per iteration, until the divide returns 0.

For this example we want to vectorize the operation and the PowerISA (and most other platforms) does not provide a vector integer divide instruction. But we do have vector integer multiply. As we will see the multiply high defined above is very handy for applying the multiplicative inverse. Also, the conversion divide is a constant value applied across the vector which simplifies the coding.

Here we can use the multiplicative inverse which is a scaled fixed point fraction calculated from the original divisor. This works nicely if the fixed radix point is just before the 16-bit fraction and we have a multiply high (vec_mulhuh()) operation. Multiplying a 16-bit unsigned integer by a 16-bit unsigned fraction generates a 32-bit product with 16-bits above (integer) and below (fraction) the radix point. The high 16-bits of the product is a good approximation of the integer quotient.

It turns out that generating the multiplicative inverse can be tricky. To produce correct results over the full range, requires possible pre-scaling and post-shifting, and sometimes a corrective addition. Fortunately, the mathematics are well understood and are commonly used in optimizing compilers. Even better, Henry Warren's book has a whole chapter on this topic.

See also

> "Hacker's Delight, 2nd Edition," Henry S. Warren, Jr, Addison Wesley, 2013. Chapter 10, Integer Division by Constants.

7.8.4.1 Divide by constant 10 examples

In the chapter above;

> Figure 10-2 Computing the magic number for unsigned division.

provides a sample C function for generating the magic number (actually a struct containing; the magic multiplicative inverse, "add" indicator, and the shift amount). For the 16-bit unsigned divisor 10, this is { 52429, 0, 3 }:

- the multiplier is 52429.
- no corrective add of the dividend is required.

- the final shift is 3-bits right.

Which could look like this:

```
 vui16_t
__test_div10 (vui16_t n)
{
  vui16_t q;
  // M= 52429, a=0, s=3
  vui16_t magic = vec_splats ((unsigned short) 52429);
  const int s = 3;
  q = vec_mulhuh (magic, n);
  return vec_srhi (q, s);
}
```

But we also need the modulo to extract each digit. The simplest and oldest technique is to multiply the quotient by the divisor (constant 10) and subtract that from the original dividend. Here we can use the vec_muluhm() operation we defined above. Which could look like this:

```
 vui16_t
__test_mod10 (vui16_t n)
{
  vui16_t q;
  // M= 52429, a=0, s=3
  vui16_t magic = vec_splats ((unsigned short) 52429);
  vui16_t c_10 = vec_splats ((unsigned short) 10);
  const int s = 3;
  vui16_t tmp, rem, q_10;
  q = vec_mulhuh (magic, n);
  q_10 = vec_srhi (q, s);
  tmp = vec_muluhm (q_10, c_10);
  rem = vec_sub (n, tmp);
  return rem;
}
```

Note

vec_sub() and vec_splats() are an existing altivec.h generic built-ins.

7.8.4.2 Divide by constant 10000 example

As we mentioned above, some divisors require an add before the shift as a correction. For the 16-bit unsigned divisor 10000 this is { 41839, 1, 14 }:

- the multiplier is 41839.
- corrective add of the dividend is required.
- the final shift is 14-bits right.

In this case the perfect multiplier is too large (>= 2**16). So the magic multiplier is reduced by 2**16 and to correct for this we need to add the dividend to the product. This add may generate a carry that must be included in the shift. Here vec_avg handles the 17-bit sum internally before shifting right 1. But vec_avg adds an extra +1 (for rounding) that we don't want. So we use (n-1) for the product correction then complete the operation with shift right (s-1). Which could look like this:

```
vui16_t
__test_div10000 (vui16_t n)
{
  vui16_t result, q;
  // M= 41839, a=1, s=14
  vui16_t magic = vec_splats ((unsigned short) 41839);
  const int s = 14;
  vui16_t tmp, rem;
  q = vec_mulhuh (magic, n);
    {
      const vui16_t vec_ones = vec_splat_u16 ( 1 );
      vui16_t n_1 = vec_sub (n, vec_ones);
      // avg = (q + (n-1) + 1) » 1
      q = vec_avg (q, n_1);
      result = vec_srhi (q, (s - 1));
    }
  return result;
}
```

Note

vec_avg(), vec_sub(), vec_splats() and vec_splat_u16() are existing altivec.h generic built-ins.

The modulo computation remains the same as Divide by constant 10 examples.

7.8.5 Performance data.

We can use the example above (see Multiply High Unsigned Halfword Example) to illustrate the performance metrics pveclib provides. For vec_mulhuh() the core operation is the sequence vmulouh/vmuleuh/vperm. This represents the best case latency, when it is used multiple times in a single larger function.

The compiler notes that vmulouh/vmuleuh are independent instructions that can execute concurrently (in separate vector pipelines). The compiler schedules them to issue in same cycle. The latency for vmulouh/vmuleuh is listed as 7 cycle and the throughput of 2 per cycle (there are 2 vector pipes for multiply). As we assume this function will use both vector pipelines, the throughput for this function is reduced to 1 per cycle.

We still need to select/merge the results. The vperm instruction is dependent on the execution of both vmulouh/vmuleuh and load of the select vector complete. For this case we assume that the load of the permute select vector has already executed. The processor can not issue the vperm until both vmulouh/vmuleuh instructions execute. The latency for vperm is 2 cycles (3 on POWER9). So the best case latency for this operation is is (7 + 2 = 9) cycles (10 on POWER9).

Looking at the first or only execution of vec_mulhuh() in a function defines the worse case latency. Here we have to include the permute select vector load and (for LE) the select vector complement. However this case provides additional multiple pipe parallelism that needs to be accounted for in the latencies.

The compiler notes that addis/vmulouh/vmuleuh are independent instructions that can execute concurrently in separate pipelines. So the compiler schedules them to issue in same cycle. The latency for vmulouh/vmuleuh is 7 cycles while the addis latency is only 2 cycles. The dependent addi instruction can issue in the 3rd cycle, while vmulouh/vmuleuh are still executing. The addi also has a 2 cycle latency, so the dependent lvx can issue in the 5th cycle, while vmulouh/vmuleuh are still executing. The lvx has a latency of 5 cycles and will not complete execution until 2 cycles after vmulouh/vmuleuh. The dependent xxlnor is waiting of the load (lvx) and has a latency of 2 cycles.

So there are two independent instruction sequences; vmulouh/vmuleuh and addis/addi/lvx/xxlnor. Both must complete execution before the vperm can issue and complete the operation. The later sequence has the longer (2+2+5+2=11) latency and dominates the timing. So the worst latency for the full sequence is (2+2+5+2+2 = 13) cycles (14 on POW↩ ER9).

processor	Latency	Throughput
power8	9-13	1/cycle
power9	10-14	1/cycle

7.8.5.1 More information.

High level performance estimates are provided as an aid to function selection when evaluating algorithms. For background on how *Latency* and *Throughput* are derived see: Performance data.

7.8.6 Function Documentation

7.8.6.1 vec_absduh()

```
static vui16_t vec_absduh (
             vui16_t vra,
             vui16_t vrb )  [inline], [static]
```

Vector Absolute Difference Unsigned halfword.

Compute the absolute difference for each halfword. For each unsigned halfword, subtract VRB[i] from VRA[i] and return the absolute value of the difference.

processor	Latency	Throughput
power8	4	1/cycle
power9	3	2/cycle

Parameters

vra	vector of 8 x unsigned halfword
vrb	vector of 8 x unsigned halfword

Returns

vector of the absolute differences.

7.8.6.2 vec_clzh()

```
static vui16_t vec_clzh (
             vui16_t vra )  [inline], [static]
```

Vector Count Leading Zeros Halfword for unsigned short elements.

Count the number of leading '0' bits (0-16) within each halfword element of a 128-bit vector.

For POWER8 (PowerISA 2.07B) or later use the Vector Count Leading Zeros Halfword instruction **vclzh**. Otherwise use sequence of pre 2.07 VMX instructions.

Note

SIMDized count leading zeros inspired by: Warren, Henry S. Jr and *Hacker's Delight*, 2nd Edition, Addison Wesley, 2013. Chapter 5 Counting Bits, Figure 5-12.

processor	Latency	Throughput
power8	2	2/cycle
power9	3	2/cycle

Parameters

vra	128-bit vector treated as 8 x 16-bit unsigned integer (halfword) elements.

Returns

128-bit vector with the leading zeros count for each halfword element.

7.8.6.3 vec_ctzh()

```
static vui16_t vec_ctzh (
            vui16_t vra )  [inline], [static]
```

Vector Count Trailing Zeros Halfword for unsigned short elements.

Count the number of trailing '0' bits (0-16) within each halfword element of a 128-bit vector.

For POWER9 (PowerISA 3.0B) or later use the Vector Count Trailing Zeros Halfword instruction **vctzh**. Otherwise use a sequence of pre ISA 3.0 VMX instructions. SIMDized count trailing zeros inspired by:

Warren, Henry S. Jr and *Hacker's Delight*, 2nd Edition, Addison Wesley, 2013. Chapter 5 Counting Bits, Section 5-4.

processor	Latency	Throughput
power8	6-8	2/cycle
power9	3	2/cycle

Parameters

vra	128-bit vector treated as 8 x 16-bit unsigned short integer (halfwords) elements.

Returns

128-bit vector with the trailing zeros count for each halfword element.

7.8.6.4 vec_mrgahh()

```
static vui16_t vec_mrgahh (
            vui32_t vra,
            vui32_t vrb )  [inline], [static]
```

Vector Merge Algebraic High Halfword operation.

Merge only the high halfwords from 8 x Algebraic words across vectors vra and vrb. This is effectively the Vector Merge Even Halfword operation that is not modified for endian.

For example merge the high 16-bits from each of 8 x 32-bit products as generated by vec_muleuh/vec_mulouh. This result is effectively a vector multiply high unsigned halfword.

processor	Latency	Throughput
power8	2-13	2/cycle
power9	3-14	2/cycle

Parameters

vra	128-bit vector unsigned int.
vrb	128-bit vector unsigned int.

Returns

A vector merge from only the high halfwords of the 8 x Algebraic words across vra and vrb.

7.8.6.5 vec_mrgalh()

```
static vui16_t vec_mrgalh (
            vui32_t vra,
            vui32_t vrb )  [inline], [static]
```

Vector Merge Algebraic Low Halfword operation.

Merge only the low halfwords from 8 x Algebraic words across vectors vra and vrb. This is effectively the Vector Merge Odd Halfword operation that is not modified for endian.

For example merge the low 16-bits from each of 8 x 32-bit products as generated by vec_muleuh/vec_mulouh. This result is effectively a vector multiply low unsigned halfword.

processor	Latency	Throughput
power8	2-13	2/cycle
power9	3-14	2/cycle

Parameters

vra	128-bit vector unsigned int.
vrb	128-bit vector unsigned int.

Returns

A vector merge from only the high halfwords of the 8 x Algebraic words across vra and vrb.

7.8.6.6 vec_mrgeh()

```
static vui16_t vec_mrgeh (
            vui16_t vra,
            vui16_t vrb )  [inline], [static]
```

Vector Merge Even Halfwords operation.

Merge the even halfword elements from the concatenation of 2 x vectors (vra and vrb).

Note

The element numbering changes between big and little-endian. So the compiler and this implementation adjusts the generated code to reflect this.

processor	Latency	Throughput
power8	2-13	2/cycle
power9	3-14	2/cycle

Parameters

vra	128-bit vector unsigned short.
vrb	128-bit vector unsigned short.

Returns

A vector merge from only the even halfwords of vra and vrb.

7.8.6.7 vec_mrgoh()

```
static vui16_t vec_mrgoh (
            vui16_t vra,
            vui16_t vrb )  [inline], [static]
```

Vector Merge Odd Halfwords operation.

Merge the odd halfword elements from the concatenation of 2 x vectors (vra and vrb).

Note

> The element numbering changes between big and little-endian. So the compiler and this implementation adjusts the generated code to reflect this.

processor	Latency	Throughput
power8	2-13	2/cycle
power9	3-14	2/cycle

Parameters

vra	128-bit vector unsigned short.
vrb	128-bit vector unsigned short.

Returns

> A vector merge from only the odd halfwords of vra and vrb.

7.8.6.8 vec_mulhsh()

```
static vi16_t vec_mulhsh (
            vi16_t vra,
            vi16_t vrb )  [inline], [static]
```

Vector Multiply High Signed halfword.

Multiple the corresponding halfword elements of two vector signed short values and return the high order 16-bits, for each 32-bit product element.

processor	Latency	Throughput
power8	9-13	1/cycle
power9	10-14	1/cycle

Parameters

vra	128-bit vector signed short.
vrb	128-bit vector signed short.

Returns

vector of the high order 16-bits of the product of the halfword elements from vra and vrb.

7.8.6.9 vec_mulhuh()

```
static vui16_t vec_mulhuh (
            vui16_t vra,
            vui16_t vrb )  [inline], [static]
```

Vector Multiply High Unsigned halfword.

Multiply the corresponding halfword elements of two vector unsigned short values and return the high order 16-bits, for each 32-bit product element.

processor	Latency	Throughput
power8	9-13	1/cycle
power9	10-14	1/cycle

Parameters

vra	128-bit vector unsigned short.
vrb	128-bit vector unsigned short.

Returns

vector of the high order 16-bits of the product of the halfword elements from vra and vrb.

7.8.6.10 vec_muluhm()

```
static vui16_t vec_muluhm (
            vui16_t vra,
            vui16_t vrb )  [inline], [static]
```

Vector Multiply Unsigned halfword Modulo.

Multiply the corresponding halfword elements of two vector unsigned short values and return the low order 16-bits of the 32-bit product for each element.

Note

vec_muluhm can be used for unsigned or signed short integers. It is the vector equivalent of Multiply Low Halfword.

processor	Latency	Throughput
power8	9-13	1/cycle
power9	10-14	1/cycle

Parameters

vra	128-bit vector unsigned short.
vrb	128-bit vector unsigned short.

Returns

vector of the low order 16-bits of the unsigned product of the halfword elements from vra and vrb.

7.8.6.11 vec_popcnth()

```
static vui16_t vec_popcnth (
            vui16_t vra )  [inline], [static]
```

Vector Population Count halfword.

Count the number of '1' bits (0-16) within each byte element of a 128-bit vector.

For POWER8 (PowerISA 2.07B) or later use the Vector Population Count Halfword instruction. Otherwise use simple Vector (VMX) instructions to count bits in bytes in parallel.

Note

SIMDized population count inspired by: Warren, Henry S. Jr and *Hacker's Delight*, 2nd Edition, Addison Wesley, 2013. Chapter 5 Counting Bits, Figure 5-2.

processor	Latency	Throughput
power8	2	2/cycle
power9	3	2/cycle

Parameters

vra	128-bit vector treated as 8 x 16-bit integers (halfword) elements.

Returns

128-bit vector with the population count for each halfword element.

7.8.6.12 vec_revbh()

```
static vui16_t vec_revbh (
             vui16_t vra )  [inline], [static]
```

byte reverse each halfword of a vector unsigned short.

For each halfword of the input vector, reverse the order of bytes / octets within the halfword.

processor	Latency	Throughput
power8	2-11	2/cycle
power9	3	2/cycle

Parameters

vra	a 128-bit vector unsigned short.

Returns

a 128-bit vector with the bytes of each halfword reversed.

7.8.6.13 vec_setb_sh()

```
static vb16_t vec_setb_sh (
             vi16_t vra )  [inline], [static]
```

Vector Set Bool from Signed Halfword.

For each halfword, propagate the sign bit to all 16-bits of that halfword. The result is vector bool short reflecting the sign bit of each 16-bit halfword.

processor	Latency	Throughput
power8	2-4	2/cycle
power9	2-5	2/cycle

Parameters

vra	Vector signed short.

Returns

vector bool short reflecting the sign bit of each halfword.

7.8.6.14 vec_slhi()

```
static vui16_t vec_slhi (
             vui16_t vra,
             const unsigned int shb )  [inline], [static]
```

Vector Shift left Halfword Immediate.

Shift left each halfword element [0-7], 0-15 bits, as specified by an immediate value. The shift amount is a const unsigned int in the range 0-15. A shift count of 0 returns the original value of vra. Shift counts greater then 15 bits return zero.

processor	Latency	Throughput
power8	4-11	2/cycle
power9	5-11	2/cycle

Parameters

vra	a 128-bit vector treated as a vector unsigned short.
shb	Shift amount in the range 0-15.

Returns

128-bit vector unsigned short, shifted left shb bits.

7.8.6.15 vec_srahi()

```
static vi16_t vec_srahi (
             vi16_t vra,
             const unsigned int shb )  [inline], [static]
```

Vector Shift Right Algebraic Halfword Immediate.

Shift right algebraic each halfword element [0-7], 0-15 bits, as specified by an immediate value. The shift amount is a const unsigned int in the range 0-15. A shift count of 0 returns the original value of vra. Shift counts greater then 7 bits return the sign bit propagated to each bit of each element.

processor	Latency	Throughput
power8	4-11	2/cycle
power9	5-11	2/cycle

Parameters

vra	a 128-bit vector treated as a vector signed char.
shb	Shift amount in the range 0-7.

Returns

128-bit vector signed short, shifted right shb bits.

7.8.6.16 vec_srhi()

```
static vui16_t vec_srhi (
            vui16_t vra,
            const unsigned int shb )  [inline], [static]
```

Vector Shift Right Halfword Immediate.

Shift right each halfword element [0-7], 0-15 bits, as specified by an immediate value. The shift amount is a const unsigned int in the range 0-15. A shift count of 0 returns the original value of vra. Shift counts greater then 15 bits return zero.

processor	Latency	Throughput
power8	4-11	2/cycle
power9	5-11	2/cycle

Parameters

vra	a 128-bit vector treated as a vector unsigned short.
shb	Shift amount in the range 0-15.

Returns

128-bit vector unsigned short, shifted right shb bits.

7.8.6.17 vec_vmaddeuh()

```
static vui32_t vec_vmaddeuh (
            vui16_t a,
            vui16_t b,
            vui16_t c )  [inline], [static]
```

Vector Multiply-Add Even Unsigned Halfwords.

Multiply the even 16-bit Words of vector unsigned short values (a $*$ b) and return sums of the unsigned 32-bit product and the even 16-bit halfwords of c ($a_{even} * b_{even}$) + EXTZ(c_{even}).

Note

The advantage of this form (versus Multiply-Sum) is that the final 32 bit sums can not overflow.

This implementation is NOT endian sensitive and the function is stable across BE/LE implementations.

processor	Latency	Throughput
power8	9-18	2/cycle
power9	9-16	2/cycle

Parameters

a	128-bit vector unsigned short.
b	128-bit vector unsigned short.
c	128-bit vector unsigned short.

Returns

> vector unsigned int sum ($a_{even} * b_{even}$) + EXTZ(c_{even}).

7.8.6.18 vec_vmaddouh()

```
static vui32_t vec_vmaddouh (
            vui16_t a,
            vui16_t b,
            vui16_t c )  [inline], [static]
```

Vector Multiply-Add Odd Unsigned Halfwords.

Multiply the odd 16-bit Halfwords of vector unsigned short values (a ∗ b) and return sums of the unsigned 32-bit product and the odd 16-bit halfwords of c ($a_{odd} * b_{odd}$) + EXTZ(c_{odd}).

Note

> The advantage of this form (versus Multiply-Sum) is that the final 32 bit sums can not overflow.
>
> This implementation is NOT endian sensitive and the function is stable across BE/LE implementations.

processor	Latency	Throughput
power8	9-18	2/cycle
power9	9-16	2/cycle

Parameters

a	128-bit vector unsigned short.
b	128-bit vector unsigned short.
c	128-bit vector unsigned short.

Returns

vector unsigned int sum ($a_{odd} * b_{odd}$) + EXTZ(c_{odd}).

7.8.6.19 vec_vmrgeh()

```
static vui16_t vec_vmrgeh (
            vui16_t vra,
            vui16_t vrb ) [inline], [static]
```

Vector Merge Even Halfwords.

Merge the even halfword elements from the concatenation of 2 x vectors (vra and vrb).

Note

This function implements the operation of a Vector Merge Even Halfword instruction, if the PowerISA included such an instruction. This implementation is NOT endian sensitive and the function is stable across BE/LE implementations. Using big-endian element numbering:

- res[0] = vra[0];
- res[1] = vrb[0];
- res[2] = vra[2];
- res[3] = vrb[2];
- res[4] = vra[4];
- res[5] = vrb[4];
- res[6] = vra[6];
- res[7] = vrb[6];

processor	Latency	Throughput
power8	2-13	2/cycle
power9	3-14	2/cycle

Parameters

vra	128-bit vector unsigned short.
vrb	128-bit vector unsigned short.

Returns

A vector merge from only the even halfwords of vra and vrb.

7.8.6.20 vec_vmrgoh()

```
static vui16_t vec_vmrgoh (
            vui16_t vra,
            vui16_t vrb )  [inline], [static]
```

Vector Merge Odd Halfwords.

Merge the odd halfword elements from the concatenation of 2 x vectors (vra and vrb).

Note

This function implements the operation of a Vector Merge Odd Halfword instruction, if the PowerISA included such an instruction. This implementation is NOT endian sensitive and the function is stable across BE/LE implementations. Using big-endian element numbering:

- res[0] = vra[1];
- res[1] = vrb[1];
- res[2] = vra[3];
- res[3] = vrb[3];
- res[4] = vra[5];
- res[5] = vrb[5];
- res[6] = vra[7];
- res[7] = vrb[7];

processor	Latency	Throughput
power8	2-13	2/cycle
power9	3-14	2/cycle

Parameters

vra	128-bit vector unsigned short.
vrb	128-bit vector unsigned short.

Returns

A vector merge from only the odd halfwords of vra and vrb.

7.9 src/pveclib/vec_int32_ppc.h File Reference

Header package containing a collection of 128-bit SIMD operations over 32-bit integer elements.

```
#include <pveclib/vec_int16_ppc.h>
```

Functions

- static vui32_t vec_absduw (vui32_t vra, vui32_t vrb)

  Vector Absolute Difference Unsigned Word.
- static vui32_t vec_clzw (vui32_t vra)

  Vector Count Leading Zeros word.
- static vui32_t vec_ctzw (vui32_t vra)

  Vector Count Trailing Zeros word.
- static vui32_t vec_mrgahw (vui64_t vra, vui64_t vrb)

  Vector Merge Algebraic High Words.
- static vui32_t vec_mrgalw (vui64_t vra, vui64_t vrb)

  Vector merge Algebraic low words.
- static vui32_t vec_mrgew (vui32_t vra, vui32_t vrb)

  Vector Merge Even Words.
- static vui32_t vec_mrgow (vui32_t vra, vui32_t vrb)

  Vector Merge Odd Words.
- static vi64_t vec_mulesw (vi32_t a, vi32_t b)

  Vector multiply even signed words.
- static vi64_t vec_mulosw (vi32_t a, vi32_t b)

  Vector multiply odd signed words.
- static vui64_t vec_muleuw (vui32_t a, vui32_t b)

  Vector multiply even unsigned words.
- static vui64_t vec_mulouw (vui32_t a, vui32_t b)

  Vector multiply odd unsigned words.
- static vi32_t vec_mulhsw (vi32_t vra, vi32_t vrb)

  Vector Multiply High Signed Word.
- static vui32_t vec_mulhuw (vui32_t vra, vui32_t vrb)

  Vector Multiply High Unsigned Word.
- static vui32_t vec_muluwm (vui32_t a, vui32_t b)

  Vector Multiply Unsigned Word Modulo.
- static vui32_t vec_popcntw (vui32_t vra)

  Vector Population Count word.
- static vui32_t vec_revbw (vui32_t vra)

  byte reverse each word of a vector unsigned int.
- static vb32_t vec_setb_sw (vi32_t vra)

  Vector Set Bool from Signed Word.
- static vui32_t vec_slwi (vui32_t vra, const unsigned int shb)

  Vector Shift left Word Immediate.
- static vi32_t vec_srawi (vi32_t vra, const unsigned int shb)

  Vector Shift Right Algebraic Word Immediate.
- static vui32_t vec_srwi (vui32_t vra, const unsigned int shb)

  Vector Shift Right Word Immediate.
- static vui32_t vec_vgl4wso (unsigned int ∗array, const long long offset0, const long long offset1, const long long offset2, const long long offset3)

  Vector Gather-Load 4 Words from scalar Offsets.
- static vui32_t vec_vgl4wwo (unsigned int ∗array, vi32_t vra)

  Vector Gather-Load 4 Words from Vector Word Offsets.

- static vui32_t vec_vgl4wwsx (unsigned int ∗array, vi32_t vra, const unsigned char scale)

  Vector Gather-Load 4 Words from Vector Word Scaled Indexes.
- static vui32_t vec_vgl4wwx (unsigned int ∗array, vi32_t vra)

  Vector Gather-Load 4 Words from Vector Word Indexes.
- static vi64_t vec_vglswso (signed int ∗array, const long long offset0, const long long offset1)

  Vector Gather-Load Signed Word from Scalar Offsets.
- static vi64_t vec_vglswdo (signed int ∗array, vi64_t vra)

  Vector Gather-Load Signed Words from Vector Doubleword Offsets.
- static vi64_t vec_vglswdsx (signed int ∗array, vi64_t vra, const unsigned char scale)

  Vector Gather-Load Signed Words from Vector Doubleword Scaled Indexes.
- static vi64_t vec_vglswdx (signed int ∗array, vi64_t vra)

  Vector Gather-Load Signed Words from Vector Doubleword Indexes.
- static vui64_t vec_vgluwso (unsigned int ∗array, const long long offset0, const long long offset1)

  Vector Gather-Load Unsigned Word from Scalar Offsets.
- static vui64_t vec_vgluwdo (unsigned int ∗array, vi64_t vra)

  Vector Gather-Load Unsigned Words from Vector Doubleword Offsets.
- static vui64_t vec_vgluwdsx (unsigned int ∗array, vi64_t vra, const unsigned char scale)

  Vector Gather-Load Unsigned Words from Vector Doubleword Scaled Indexes.
- static vui64_t vec_vgluwdx (unsigned int ∗array, vi64_t vra)

  Vector Gather-Load Unsigned Words from Vector Doubleword Indexes.
- static vi64_t vec_vlxsiwax (const signed long long ra, const signed int ∗rb)

  Vector Load Scalar Integer Word Algebraic Indexed.
- static vui64_t vec_vlxsiwzx (const signed long long ra, const unsigned int ∗rb)

  Vector Load Scalar Integer Word and Zero Indexed.
- static vui64_t vec_vmadd2euw (vui32_t a, vui32_t b, vui32_t c, vui32_t d)

  Vector Multiply-Add2 Even Unsigned Words.
- static vui64_t vec_vmadd2ouw (vui32_t a, vui32_t b, vui32_t c, vui32_t d)

  Vector Multiply-Add2 Odd Unsigned Words.
- static vui64_t vec_vmaddeuw (vui32_t a, vui32_t b, vui32_t c)

  Vector Multiply-Add Even Unsigned Words.
- static vui64_t vec_vmaddouw (vui32_t a, vui32_t b, vui32_t c)

  Vector Multiply-Add Odd Unsigned Words.
- static vui64_t vec_vmsumuwm (vui32_t a, vui32_t b, vui64_t c)

  Vector Multiply-Sum Unsigned Word Modulo.
- static vui64_t vec_vmuleuw (vui32_t vra, vui32_t vrb)

  Vector Multiply Even Unsigned words.
- static vui64_t vec_vmulouw (vui32_t vra, vui32_t vrb)

  Vector Multiply Odd Unsigned Words.
- static void vec_vsst4wso (vui32_t xs, unsigned int ∗array, const long long offset0, const long long offset1, const long long offset2, const long long offset3)

  Vector Scatter-Store 4 words to Scalar Offsets.
- static void vec_vsst4wwo (vui32_t xs, unsigned int ∗array, vi32_t vra)

  Vector Scatter-Store 4 words to Vector Word Offsets.
- static void vec_vsst4wwsx (vui32_t xs, unsigned int ∗array, vi32_t vra, const unsigned char scale)

  Vector Scatter-Store 4 words to Vector Word Indexes.
- static void vec_vsst4wwx (vui32_t xs, unsigned int ∗array, vi32_t vra)

  Vector Scatter-Store 4 words to Vector Word Indexes.

- static void vec_vsstwdo (vui64_t xs, unsigned int ∗array, vi64_t vra)

  Vector Scatter-Store Words to Vector Doubleword Offsets.
- static void vec_vsstwdsx (vui64_t xs, unsigned int ∗array, vi64_t vra, const unsigned char scale)

  Vector Scatter-Store Words to Vector Doubleword Scaled Indexes.
- static void vec_vsstwdx (vui64_t xs, unsigned int ∗array, vi64_t vra)

  Vector Scatter-Store Words to Vector Doubleword Indexes.
- static void vec_vsstwso (vui64_t xs, unsigned int ∗array, const long long offset0, const long long offset1)

  Vector Scatter-Store Words to Scalar Offsets.
- static void vec_vstxsiwx (vui32_t xs, const signed long long ra, unsigned int ∗rb)

  Vector Store Scalar Integer Word Indexed.
- static vi32_t vec_vsum2sw (vi32_t vra, vi32_t vrb)

  Vector Sum-across Half Signed Word Saturate.
- static vi32_t vec_vsumsw (vi32_t vra, vi32_t vrb)

  Vector Sum-across Signed Word Saturate.
- static vi64_t vec_vupkhsw (vi32_t vra)

  Vector Unpack High Signed Word.
- static vui64_t vec_vupkhuw (vui32_t vra)

  Vector Unpack High Unsigned Word.
- static vi64_t vec_vupklsw (vi32_t vra)

  Vector Unpack Low Signed Word.
- static vui64_t vec_vupkluw (vui32_t vra)

  Vector Unpack Low Unsigned Word.

7.9.1 Detailed Description

Header package containing a collection of 128-bit SIMD operations over 32-bit integer elements.

Most of these operations are implemented in a single instruction on newer (POWER8/POWER9) processors. This header serves to fill in functional gaps for older (POWER7, POWER8) processors and provides a in-line assembler implementation for older compilers that do not provide the build-ins.

Most vector int (32-bit integer word) operations are implemented with PowerISA VMX instructions either defined by the original VMX (AKA Altivec) or added to later versions of the PowerISA. Vector word-wise merge, shift, and splat operations were added with VSX in PowerISA 2.06B (POWER7). PowerISA 2.07B (POWER8) added several useful word wise operations (multiply, merge even/odd, count leading zeros, population count) not included in the original V↩ MX. PowerISA 3.0B (POWER9) adds several more (compare not equal, count trailing zeros, extend sign, extract/insert, and parity). Most of these intrinsic (compiler built-ins) operations are defined in <altivec.h> and described in the compiler documentation.

Note

> The compiler disables associated <altivec.h> built-ins if the **mcpu** target does not enable the specific instruction. For example if you compile with **-mcpu=power7**, vec_vclz and vec_vclzw will not be defined. Another example if you compile with **-mcpu=power8**, vec_revb will not be defined. This header provides the appropriate substitutions, will generate the minimum code, appropriate for the target, and produce correct results.
>
> Most ppc64le compilers will default to -mcpu=power8 if not specified.

The newly introduced vector operations imply some useful composite operations. For example, we can make the vector multiply even/odd/modulo word operations available for older compilers. And provide implementations for older (POW↩ER7 and earlier) processors using the original VMX operations.

This header covers operations that are either:

- Implemented in hardware instructions for later processors and useful to programmers, on slightly older processors, even if the equivalent function requires more instructions. Examples include the multiply even/odd/modulo word operations.
- Defined in the OpenPOWER ABI but *not* yet defined in <altivec.h> provided by available compilers in common use. Examples include Count Leading Zeros, Population Count and Byte Reverse.
- Commonly used operations, not covered by the ABI or <altivec.h>, and require multiple instructions or are not obvious. Examples include the shift immediate, merge algebraic high/low, and multiply high operations.

7.9.2 Recent Additions

Added vec_vmaddeuw(), vec_vmaddouw(), vec_vmadd2euw(), and vec_vmadd2ouw() as an optimization for the vector multiply quadword implementations on POWER8.

7.9.3 Endian problems with word operations

It would be useful to provide a vector multiply high word (return the high order 32-bits of the 64-bit product) operation. This can be used for multiplicative inverse (effectively integer divide) operations. Neither integer multiply high nor divide are available as vector instructions. However the multiply high word operation can be composed from the existing multiply even/odd word operations followed by the vector merge even word instruction.

As a prerequisite we need to provide the merge even/odd word operations for older compilers and an implementation for older (POWER7) processors. Fortunately vector merge operations are just a special case of vector permute. So the POWER7 (and earlier) implementation can use vec_perm and appropriate selection vectors to provide these merge operations.

But this is complicated by *little-endian* (LE) support as specified in the OpenPOWER ABI and as implemented in the compilers. Little-endian changes the effective vector element numbering and the location of even and odd elements. This means that the vector built-ins provided by altivec.h may not generate the instructions you would expect.

See also

General Endian Issues

The OpenPOWER ABI provides a helpful table of `Endian Sensitive Operations`. For vec_mergee (vmrgew) it specifies:

> Swap inputs and use vmrgow, for LE.

Also for vec_mule (vmuleuw, vmulesw):

> Replace with vmulouw and so on, for LE.

Also for vec_perm (vperm) it specifies:

> For LE, Swap input arguments and complement the selection vector.

The above is just a sampling of a larger list of Endian Sensitive Operations.

So the obvious coding for Vector Multiply High Word:

```
vui32_t
test_mulhw (vui32_t vra, vui32_t vrb)
{
  return vec_mergee ((vui32_t)vec_mule (vra, vrb),
                     (vui32_t)vec_mulo (vra, vrb));
}
```

Would produce the expected code and correct results when compiled for BE:

```
<test_mulhw>:
        vmuleuw v0,v2,v3
        vmuluuw v2,v2,v3
        vmrgew  v2,v0,v2
        blr
```

But the following and wrong code for LE:

```
<test_mulhw>:
        vmulouw v0,v2,v3
        vmuleuw v2,v2,v3
        vmrgow  v2,v2,v0
        blr
```

The compiler swapped the multiplies even for odd and odd of even. That is somewhat mitigated by swapping the input arguments in the merge. But changing the merge from even to odd actually returns the low order 32-bits of the product. This is not the correct result for multiply high.

This header provides implementations of vector merge even/odd word (vec_mrgew() and vec_mrgow()) that support older compilers and older (POWER7) processor. Similarly for the multiply Even/odd unsigned/signed word instructions (vec_mulesw(), vec_mulosw(), vec_muleuw() and vec_mulouw()). These implementations include the mandated LE transforms.

7.9.3.1 Vector Merge Algebraic High Word example

This header also provides the higher level operations Vector Merge Algebraic High/low Word (vec_mrgahw() and vec_mrgalw()). These implementations generate the correct merge even/odd word instruction for the operation independent of endian.

Note

The parameters are vector unsigned long (vui64_t) to match results from vec_muleuw() and vec_mulouw().

```
static inline vui32_t
vec_mrgahw (vui64_t vra, vui64_t vrb)
{
  vui32_t res;
#ifdef _ARCH_PWR8
#ifdef vec_vmrgew // Use altivec.h builtins
#if __BYTE_ORDER__ == __ORDER_LITTLE_ENDIAN__
  // really want vmrgew here! So do the opposite.
  res = vec_vmrgow ((vui32_t)vrb, (vui32_t)vra);
#else
  res = vec_vmrgew ((vui32_t)vra, (vui32_t)vrb);
#endif
#else // Generate vmrgew directly in assembler
  __asm__(
      "vmrgew %0,%1,%2;\n"
      : "=v" (res)
      : "v" (vra),
      "v" (vrb)
      : );
#endif
#else // POWER7 and earlier, Assume BE only
  const vui32_t vconstp =
      CONST_VINT32_W(0x00010203,  0x10111213, 0x08090a0b,  0x18191a1b);
  res = (vui32_t) vec_perm ((vui8_t) vra, (vui8_t) vrb, (vui8_t) vconstp);
#endif
  return (res);
}
```

The implementation is a bit complicated so that is can nullify the unwanted LE transformation of vec_vmrgew(), in addition to handling older and compilers and processors.

7.9.3.2 Vector Multiply High Unsigned Word example

Now we can implement Vector Multiply High Unsigned Word (vec_mulhuw()):

```
static inline vui32_t
vec_mulhuw (vui32_t vra, vui32_t vrb)
{
#if __BYTE_ORDER__ == __ORDER_LITTLE_ENDIAN__
  return vec_mrgahw (vec_mulouw (vra, vrb), vec_muleuw (vra, vrb));
#else
  return vec_mrgahw (vec_muleuw (vra, vrb), vec_mulouw (vra, vrb));
#endif
}
```

Again the implementation is more complicated than expected as we still have to nullify the LE transformation associated with multiply even/odd.

The good news is all this complexity is contained within pveclib and the generated code is still just 3 instructions.

```
vmulouw v0,v2,v3
vmuleuw v2,v2,v3
vmrgew  v2,v2,v0
```

7.9.4 Vector Word Examples

Suppose we have a requirement to convert an array of 32-bit time-interval values that need to convert to timespec format. For simplicity we will also assume that the array is nicely (Quadword) aligned and an integer multiple of 4 words.

The PowerISA provides a 64-bit TimeBase register that clocks at a constant 512MHz. The TimeBase can be read directly as either the full 64-bit value or as 32-bit upper and lower halves. For this example we assume that the lower 32-bits of the TimeBase is sufficient to compute intervals (∼8.38 seconds). TimeBase values of adjacent events are subtracted to generate the intervals stored in the array.

The timespec format it a struct of unsigned int fields for seconds and microseconds. So the task is to convert the 512MHz TimeBase intervals to microseconds and then split the integer seconds and microseconds for the timespec.

First the TimeBase to microseconds conversion is simply (1000000 / 512000000) which reduces to (1 / 512) or divide by 512. The vector unit does not provide integer divide but luckily, 512 is a power of 2 and we can shift right. If we don't care for the niceties of rounding we can simply shift right 9 bits:

```
tb_usec = vec_srwi (*tb++, 9);
```

But if we decide that rounding is important we can leverage the Vector Average Unsigned Word (vavguw) instruction. Here we need to add 256 (512 / 2 = 256) to the timeBase interval before we shift right.

But we need to reverse engineer the vavguw operation to get the results we want. For each word, vavguw computes the sum of A and B plus 1, then shifts the 33-bit sum right 1 bit. We can effectively round by passing the rounding factor as the B operand to the vec_avg() built-in. But we get a +1 and 1 bit right shift for free. So in this case the rounding constant is 256-1 = 255. And we only need to shift an additional 8 bits to complete the conversion:

```
const vui32_t rnd_512 =
  { (256-1), (256-1), (256-1), (256-1) };
// Convert 512MHz timebase to microseconds with rounding.
tmp = vec_avg (*tb++, rnd_512);
tb_usec = vec_srwi (tmp, 8);
```

Note

> vec_avg() is an existing altivec.h generic built-in.

Next we need to separate TimeBase microseconds into the integer seconds and microseconds. Normally scalar codes would use integer divide/modulo by 1000000. Did I mention that the PowerISA vector unit does not have a integer divide operation?

Instead we can use the multiplicative inverse which is a scaled fixed point fraction calculated from the original divisor. This works nicely if the fixed radix point is just before the 32-bit fraction and we have a multiply high (vec_mulhuw()) operation. Multiplying a 32-bit unsigned integer by a 32-bit unsigned fraction generates a 64-bit product with 32-bits above (integer) and below (fraction) the radix point. The high 32-bits of the product is the integer quotient.

It turns out that generating the multiplicative inverse can be tricky. To produce correct results over the full analysis, possible pre-scaling and post-shifting, and sometimes a corrective addition is necessary. Fortunately the mathematics are well understood and are commonly used in optimizing compilers. Even better, Henry Warren's book has a whole chapter on this topic.

See also

> "Hacker's Delight, 2nd Edition," Henry S. Warren, Jr, Addison Wesley, 2013. Chapter 10, Integer Division by Constants.

In the chapter above;

> Figure 10-2 Computing the magic number for unsigned division.

provides a sample C function for generating the magic number (actually a struct containing; the magic multiplicative inverse, "add" indicator, and the shift amount.). For the divisor 1000000 this is { 1125899907, 0 , 18 }:

- the multiplier is 1125899907.
- no corrective add of the dividend is required.

- the final shift is 18-bits right.

```
const vui32_t mul_invs_1m =
  { 1125899907, 1125899907, 1125899907, 1125899907 };
const int shift_1m = 18;
tmp = vec_mulhuw (tb_usec, mul_invs_1m);
seconds = vec_srwi (tmp, shift_1m);
```

Now we need to compute the remainder to get microseconds.

```
const vui32_t usec_sec =
  { 1000000, 1000000, 1000000, 1000000 };
tmp = vec_muluwm (seconds, usec_sec);
useconds = vec_sub (tb_usec, tmp);
```

Finally we need to merge the vectors of seconds and useconds into vectors of timespec.

```
timespec1 = vec_mergeh (seconds, useconds);
timespec2 = vec_mergel (seconds, useconds);
```

Note

vec_sub(), vec_mergeh(), and vec_mergel() are an existing altivec.h generic built-ins.

7.9.4.1 Vectorized TimeBase conversion example

Here is the complete vectorized TimeBase to timespec conversion example:

```
void
example_convert_timebase (vui32_t *tb, vui32_t *timespec, int n)
{
  const vui32_t rnd_512 =
    { (256-1), (256-1), (256-1), (256-1) };
  // Magic numbers for multiplicative inverse to divide by 1,000,000
  // are 1125899907 and shift right 18 bits.
  const vui32_t mul_invs_1m =
    { 1125899907, 1125899907, 1125899907, 1125899907 };
  const int shift_1m = 18;
  // Need const for microseconds/second to extract remainder.
  const vui32_t usec_sec =
    { 1000000, 1000000, 1000000, 1000000 };
  vui32_t tmp, tb_usec, seconds, useconds;
  vui32_t timespec1, timespec2;
  int i;
  for (i = 0; i < n; i++)
    {
      // Convert 512MHz timebase to microseconds with rounding.
      tmp = vec_avg (*tb++, rnd_512);
      tb_usec = vec_srwi (tmp, 8);
      // extract integer seconds from tb_usec.
      tmp = vec_mulhuw (tb_usec, mul_invs_1m);
      seconds = vec_srwi (tmp, shift_1m);
      // Extract remainder microseconds.
      tmp = vec_muluwm (seconds, usec_sec);
      useconds = vec_sub (tb_usec, tmp);
      // Use merge high/low to interleave seconds and useconds in timespec.
      timespec1 = vec_mergeh (seconds, useconds);
      timespec2 = vec_mergel (seconds, useconds);
      // Store timespec.
      *timespec++ = timespec1;
      *timespec++ = timespec2;
    }
}
```

7.9.5 Performance data.

High level performance estimates are provided as an aid to function selection when evaluating algorithms. For background on how *Latency* and *Throughput* are derived see: Performance data.

7.9.6 Function Documentation

7.9.6.1 vec_absduw()

```
static vui32_t vec_absduw (
             vui32_t vra,
             vui32_t vrb )  [inline], [static]
```

Vector Absolute Difference Unsigned Word.

Compute the absolute difference for each word. For each unsigned word, subtract VRB[i] from VRA[i] and return the absolute value of the difference.

processor	Latency	Throughput
power8	4	1/cycle
power9	3	2/cycle

Parameters

vra	vector of 4 x unsigned words
vrb	vector of 4 x unsigned words

Returns

vector of the absolute differences.

7.9.6.2 vec_clzw()

```
static vui32_t vec_clzw (
             vui32_t vra )  [inline], [static]
```

Vector Count Leading Zeros word.

Count the number of leading '0' bits (0-32) within each word element of a 128-bit vector.

For POWER8 (PowerISA 2.07B) or later use the Vector Count Leading Zeros Word instruction **vclzw**. Otherwise use sequence of pre 2.07 VMX instructions. SIMDized count leading zeros inspired by:

Warren, Henry S. Jr and *Hacker's Delight*, 2nd Edition, Addison Wesley, 2013. Chapter 5 Counting Bits, Figure 5-12.

processor	Latency	Throughput
power8	2	2/cycle
power9	3	2/cycle

Parameters

vra	128-bit vector treated as 4 x 32-bit integer (words) elements.

Returns

128-bit vector with the Leading Zeros count for each word element.

7.9.6.3 vec_ctzw()

```
static vui32_t vec_ctzw (
            vui32_t vra )  [inline], [static]
```

Vector Count Trailing Zeros word.

Count the number of trailing '0' bits (0-32) within each word element of a 128-bit vector.

For POWER9 (PowerISA 3.0B) or later use the Vector Count Trailing Zeros Word instruction **vctzw**. Otherwise use a sequence of pre ISA 3.0 VMX instructions. SIMDized count Trailing zeros inspired by:

Warren, Henry S. Jr and *Hacker's Delight*, 2nd Edition, Addison Wesley, 2013. Chapter 5 Counting Bits, Section 5-4.

processor	Latency	Throughput
power8	6-8	2/cycle
power9	3	2/cycle

Parameters

vra	128-bit vector treated as 4 x 32-bit integer (words) elements.

Returns

128-bit vector with the Trailng Zeros count for each word element.

7.9.6.4 vec_mrgahw()

```
static vui32_t vec_mrgahw (
            vui64_t vra,
            vui64_t vrb )  [inline], [static]
```

Vector Merge Algebraic High Words.

Merge only the high words from 4 x Algebraic doublewords across vectors vra and vrb. This effectively the Vector Merge Even Word operation that is not modified for endian.

For example merge the high 32-bits from 4 x 64-bit products as generated by vec_muleuw/vec_mulouw. This result is effectively a vector multiply high unsigned word.

Note

This implementation is NOT endian sensitive and the function is stable across BE/LE implementations.

processor	Latency	Throughput
power8	2	2/cycle
power9	2	2/cycle

Parameters

vra	128-bit vector unsigned long.
vrb	128-bit vector unsigned long.

Returns

A vector merge from only the high words of the 4 x Algebraic doublewords across vra and vrb.

7.9.6.5 vec_mrgalw()

```
static vui32_t vec_mrgalw (
            vui64_t vra,
            vui64_t vrb )  [inline], [static]
```

Vector merge Algebraic low words.

Merge the arithmetic low words 4 x Algebraic doublewords across vectors vra and vrb. This is effectively the Vector Merge Odd Word operation that is not modified for endian.

For example merge the low 32-bits from 4 x 64-bit products as generated by vec_muleuw/vec_mulouw. This result is effectively a vector multiply low unsigned word (multiply unsigned word modulo).

Note

This implementation is NOT endian sensitive and the function is stable across BE/LE implementations.

processor	Latency	Throughput
power8	2	2/cycle
power9	2	2/cycle

Parameters

vra	128-bit vector unsigned long.
vrb	128-bit vector unsigned long.

Returns

A vector merge from only the low words of the 4 x Algebraic doublewords across vra and vrb.

7.9.6.6 vec_mrgew()

```
static vui32_t vec_mrgew (
            vui32_t vra,
            vui32_t vrb )  [inline], [static]
```

Vector Merge Even Words.

Merge the even word elements from the concatenation of 2 x vectors (vra and vrb).

- res[0] = vra[0];
- res[1] = vrb[0];
- res[2] = vra[2];
- res[3] = vrb[2];

The element numbering changes between big and little-endian environements. So the compiler and this implementation adjusts the generated code to reflect this.

processor	Latency	Throughput
power8	2	2/cycle
power9	2	2/cycle

Parameters

vra	128-bit vector unsigned int.
vrb	128-bit vector unsigned int.

Returns

A vector merge from only the even words of vra and vrb.

7.9.6.7 vec_mrgow()

```
static vui32_t vec_mrgow (
             vui32_t vra,
             vui32_t vrb )  [inline], [static]
```

Vector Merge Odd Words.

Merge the odd word elements from the concatenation of 2 x vectors (vra and vrb).

- res[0] = vra[1];
- res[1] = vrb[1];
- res[2] = vra[3];
- res[3] = vrb[3];

The element numbering changes between big and little-endian environements. So the compiler and this implementation adjusts the generated code to reflect this.

processor	Latency	Throughput
power8	2	2/cycle
power9	2	2/cycle

Parameters

vra	128-bit vector unsigned int.
vrb	128-bit vector unsigned int.

Returns

A vector merge from only the even words of vra and vrb.

7.9.6.8 vec_mulesw()

```
static vi64_t vec_mulesw (
             vi32_t a,
             vi32_t b )  [inline], [static]
```

Vector multiply even signed words.

Multiple the even words of two vector signed int values and return the signed long product of the even words.

For POWER8 and later we can use the vmulesw instruction. But for POWER7 and earlier we have to construct word multiplies from halfword multiplies. See vec_muleuw().

Here we start with a unsigned vec_muleuw product, then correct the high 32-bits of the product to signed. Based on: Warren, Henry S. Jr and *Hacker's Delight*, 2nd Edition, Addison Wesley, 2013. Chapter 8 Multiplication, Section 8-3 High-Order Product Signed from/to Unsigned.

processor	Latency	Throughput
power8	7	2/cycle
power9	7	2/cycle

Parameters

a	128-bit vector signed int.
b	128-bit vector signed int.

Returns

vector signed long product of the even words of a and b.

7.9.6.9 vec_muleuw()

```
static vui64_t vec_muleuw (
            vui32_t a,
            vui32_t b ) [inline], [static]
```

Vector multiply even unsigned words.

Multiple the even words of two vector unsigned int values and return the unsigned long product of the even words.

For POWER8 and later we can use the vmuleuw instruction. But for POWER7 and earlier we have to construct word multiplies from two halfword multiplies (vmuleuh and vmulouh). Then sum the partial products for the final doubleword results. This is complicated by the fact that vector add doubleword is not available for POWER7. So we need to construct the doubleword add from Vector Add Unsigned Word Modulo (vadduwm) and Vector Add and Write Carry-Out Unsigned Word (vaddcuw) with shift double quadword to reposition the low word carry and a final vadduwm to complete the carry propagation for the doubleword add.

processor	Latency	Throughput
power8	7	2/cycle
power9	7	2/cycle

Parameters

a	128-bit vector unsigned int.
b	128-bit vector unsigned int.

Returns

vector unsigned long product of the even words of a and b.

7.9.6.10 vec_mulhsw()

```
static vi32_t vec_mulhsw (
             vi32_t vra,
             vi32_t vrb )  [inline], [static]
```

Vector Multiply High Signed Word.

Multiple the corresponding word elements of two vector signed int values and return the high order 32-bits, for each 64-bit product element.

processor	Latency	Throughput
power8	9	1/cycle
power9	9	1/cycle

Parameters

vra	128-bit vector signed int.
vrb	128-bit vector signed int.

Returns

vector of the high order 32-bits of the product of the word elements from vra and vrb.

7.9.6.11 vec_mulhuw()

```
static vui32_t vec_mulhuw (
             vui32_t vra,
             vui32_t vrb )  [inline], [static]
```

Vector Multiply High Unsigned Word.

Multiple the corresponding word elements of two vector unsigned int values and return the high order 32-bits, from each 64-bit product.

processor	Latency	Throughput
power8	9	1/cycle
power9	9	1/cycle

Note

This operation can be used to effectively perform a divide by multiplying by the scaled multiplicative inverse (reciprocal).

Warren, Henry S. Jr and *Hacker's Delight*, 2nd Edition, Addison Wesley, 2013. Chapter 10, Integer Division by Constants.

Parameters

vra	128-bit vector unsigned int.
vrb	128-bit vector unsigned int.

Returns

vector of the high order 32-bits of the signed product of the word elements from vra and vrb.

7.9.6.12 vec_mulosw()

```
static vi64_t vec_mulosw (
            vi32_t a,
            vi32_t b )  [inline], [static]
```

Vector multiply odd signed words.

Multiple the odd words of two vector signed int values and return the signed long product of the odd words.

For POWER8 and later we can use the vmulosw instruction. But for POWER7 and earlier we have to construct word multiplies from halfword multiplies. See vec_mulouw().

Here we start with a unsigned vec_mulouw product, then correct the high-order 32-bits of the product to signed. Based on: Warren, Henry S. Jr and *Hacker's Delight*, 2nd Edition, Addison Wesley, 2013. Chapter 8 Multiplication, Section 8-3 High-Order Product Signed from/to Unsigned.

processor	Latency	Throughput
power8	7	2/cycle
power9	7	2/cycle

Parameters

a	128-bit vector signed int.
b	128-bit vector signed int.

Returns

vector signed long product of the odd words of a and b.

7.9.6.13 vec_mulouw()

```
static vui64_t vec_mulouw (
             vui32_t a,
             vui32_t b )  [inline], [static]
```

Vector multiply odd unsigned words.

Multiple the odd words of two vector unsigned int values and return the unsigned long product of the odd words.

For POWER8 and later we can use the vmulouw instruction. But for POWER7 and earlier we have to construct word multiplies from two halfword multiplies (vmuleuh and vmulouh). Then sum the partial products for the final doubleword results. This is complicated by the fact that vector add doubleword is not available for POWER7. So we need to construct the doubleword add from Vector Add Unsigned Word Modulo (vadduwm) and Vector Add and Write Carry-Out Unsigned Word (vaddcuw) with shift double quadword to reposition the low word carry and a final vadduwm to complete the carry propagation for the doubleword add.

processor	Latency	Throughput
power8	7	2/cycle
power9	7	2/cycle

Parameters

a	128-bit vector unsigned int.
b	128-bit vector unsigned int.

Returns

vector unsigned long product of the odd words of a and b.

7.9.6.14 vec_muluwm()

```
static vui32_t vec_muluwm (
             vui32_t a,
             vui32_t b )  [inline], [static]
```

Vector Multiply Unsigned Word Modulo.

Multiple the corresponding word elements of two vector unsigned int values and return the low order 32-bits of the 64-bit product for each element.

Note

vec_muluwm can be used for unsigned or signed integers. It is the vector equivalent of Multiply Low Word.

processor	Latency	Throughput
power8	7	2/cycle
power9	7	2/cycle

Parameters

a	128-bit vector signed int.
b	128-bit vector signed int.

Returns

vector of the low order 32-bits of the unsigned product of the word elements from vra and vrb.

7.9.6.15 vec_popcntw()

```
static vui32_t vec_popcntw (
             vui32_t vra )  [inline], [static]
```

Vector Population Count word.

Count the number of '1' bits (0-32) within each word element of a 128-bit vector.

For POWER8 (PowerISA 2.07B) or later use the Vector Population Count Word instruction. Otherwise use the pveclib vec_popcntb to count each byte then sum across with Vector Sum across Quarter Unsigned Byte Saturate.

processor	Latency	Throughput
power8	2	2/cycle
power9	3	2/cycle

Parameters

vra	128-bit vector treated as 4 x 32-bit integer (words) elements.

Returns

128-bit vector with the population count for each word element.

7.9.6.16 vec_revbw()

```
static vui32_t vec_revbw (
             vui32_t vra )  [inline], [static]
```

byte reverse each word of a vector unsigned int.

For each word of the input vector, reverse the order of bytes / octets within the word.

processor	Latency	Throughput
power8	2-11	2/cycle
power9	3	2/cycle

Parameters

vra	a 128-bit vector unsigned int.

Returns

a 128-bit vector with the bytes of each word reversed.

7.9.6.17 vec_setb_sw()

```
static vb32_t vec_setb_sw (
             vi32_t vra )  [inline], [static]
```

Vector Set Bool from Signed Word.

For each word, propagate the sign bit to all 32-bits of that word. The result is vector bool int reflecting the sign bit of each 32-bit word.

processor	Latency	Throughput
power8	2-4	2/cycle
power9	2-5	2/cycle

Parameters

vra	Vector signed int.

Returns

vector bool int reflecting the sign bits of each word.

7.9.6.18 vec_slwi()

```
static vui32_t vec_slwi (
            vui32_t vra,
            const unsigned int shb )  [inline], [static]
```

Vector Shift left Word Immediate.

Shift left each word element [0-3], 0-31 bits, as specified by an immediate value. The shift amount is a const unsigned int in the range 0-31. A shift count of 0 returns the original value of vra. Shift counts greater then 31 bits return zero.

processor	Latency	Throughput
power8	4-11	2/cycle
power9	5-11	2/cycle

Parameters

vra	a 128-bit vector treated as a vector unsigned int.
shb	shift amount in the range 0-31.

Returns

128-bit vector unsigned int, shifted left shb bits.

7.9.6.19 vec_srawi()

```
static vi32_t vec_srawi (
            vi32_t vra,
            const unsigned int shb )  [inline], [static]
```

Vector Shift Right Algebraic Word Immediate.

Shift Right Algebraic each word element [0-3], 0-31 bits, as specified by an immediate value. The shift amount is a const unsigned int in the range 0-31. A shift count of 0 returns the original value of vra. Shift counts greater then 31 bits return the sign bit propagated to each bit of each element.

processor	Latency	Throughput
power8	4-11	2/cycle
power9	5-11	2/cycle

Parameters

vra	a 128-bit vector treated as a vector signed int.
shb	shift amount in the range 0-31.

Returns

128-bit vector signed int, shifted right shb bits.

7.9.6.20 vec_srwi()

```
static vui32_t vec_srwi (
            vui32_t vra,
            const unsigned int shb )  [inline], [static]
```

Vector Shift Right Word Immediate.

Shift right each word element [0-3], 0-31 bits, as specified by an immediate value. The shift amount is a const unsigned int in the range 0-31. A shift count of 0 returns the original value of vra. Shift counts greater then 31 bits return zero.

processor	Latency	Throughput
power8	4-11	2/cycle
power9	5-11	2/cycle

Parameters

vra	a 128-bit vector treated as a vector unsigned char.
shb	shift amount in the range 0-31.

Returns

128-bit vector unsigned int, shifted right shb bits.

7.9.6.21 vec_vgl4wso()

```
static vui32_t vec_vgl4wso (
            unsigned int * array,
            const long long offset0,
            const long long offset1,
            const long long offset2,
            const long long offset3 )  [inline], [static]
```

Vector Gather-Load 4 Words from scalar Offsets.

For each scalar offset[0,1,2,3], load the word from the effective address formed by ∗(char∗)array+offset[0-3]. Merge resulting word elements [0,1,2,3] and return the resulting vector.

processor	Latency	Throughput
power8	10	1/cycle
power9	11	1/cycle

Parameters

array	Pointer to array of integer words.
offset0	Scalar (64-bit) byte offset from &array.
offset1	Scalar (64-bit) byte offset from &array.
offset2	Scalar (64-bit) byte offset from &array.
offset3	Scalar (64-bit) byte offset from &array.

Returns

vector word containing word elements [0-3] loaded from ∗(char∗)array+offset[0-3].

7.9.6.22 vec_vgl4wwo()

```
static vui32_t vec_vgl4wwo (
            unsigned int * array,
            vi32_t vra )  [inline], [static]
```

Vector Gather-Load 4 Words from Vector Word Offsets.

For each signed word element [i] of vra, load the word element at ∗(char∗)array+vra[i]. Merge those word elements and return the resulting vector.

processor	Latency	Throughput
power8	14	1/cycle
power9	15	1/cycle

Parameters

array	Pointer to array of integer words.
vra	Vector of signed word (32-bit) byte offsets from &array.

Returns

vector word containing word elements [0-3], each loaded from ∗(char∗)array+vra[0-3].

7.9.6.23 vec_vgl4wwsx()

```
static vui32_t vec_vgl4wwsx (
            unsigned int * array,
            vi32_t vra,
            const unsigned char scale )  [inline], [static]
```

Vector Gather-Load 4 Words from Vector Word Scaled Indexes.

For each signed word element [i] of vra, load the word element at array[vra[i] << scale]. Merge those word elements and return the resulting vector.

Note

Signed word indexes are expanded (unpacked) to doublewords before shifting left 2+scale bits. This converts each index to an 64-bit offset for effective address calculation.

processor	Latency	Throughput
power8	16-25	1/cycle
power9	18-27	1/cycle

Parameters

array	Pointer to array of integer words.
vra	Vector of signed word (32-bit) indexes.
scale	8-bit integer. Indexes are multiplying by 2^{scale}.

Returns

vector word containing word elements [0-3] each loaded from array[vra[0-3] << scale].

7.9.6.24 vec_vgl4wwx()

```
static vui32_t vec_vgl4wwx (
             unsigned int * array,
             vi32_t vra )  [inline], [static]
```

Vector Gather-Load 4 Words from Vector Word Indexes.

For word element [i] of vra, load the word element at array[vra[i]]. Merge those word elements and return the resulting vector.

Note

Signed word indexes are expanded (unpacked) to doublewords before shifting left 2 bits. This converts each index to an 64-bit offset for effective address calculation.

processor	Latency	Throughput
power8	16-25	1/cycle
power9	18-27	1/cycle

Parameters

array	Pointer to array of integer words.
vra	Vector of signed word (32-bit) indexes.

Returns

vector word containing word elements [0-3], each loaded from array[vra[0-3]].

7.9.6.25 vec_vglswdo()

```
static vi64_t vec_vglswdo (
             signed int * array,
             vi64_t vra )  [inline], [static]
```

Vector Gather-Load Signed Words from Vector Doubleword Offsets.

For each doubleword element [i] of vra, load the sign extended word element at ∗(char∗)array+vra[i]. Merge doubleword elements [0,1] and return the resulting vector.

processor	Latency	Throughput
power8	12	1/cycle
power9	11	1/cycle

Parameters

array	Pointer to array of signed words.
vra	Vector of doubleword (64-bit) byte offsets from &array.

Returns

vector doubleword elements [0,1] loaded from sign extended words at ∗(char∗)array+vra[0,1].

7.9.6.26 vec_vglswdsx()

```
static vi64_t vec_vglswdsx (
            signed int * array,
            vi64_t vra,
            const unsigned char scale )  [inline], [static]
```

Vector Gather-Load Signed Words from Vector Doubleword Scaled Indexes.

For each doubleword element [i] of vra, load the sign extended word element at array[vra[i] << scale)]. Merge doubleword elements [0,1] and return the resulting vector.

processor	Latency	Throughput
power8	14-23	1/cycle
power9	13-22	1/cycle

Parameters

array	Pointer to array of signed words.
vra	Vector of doubleword indexes from &array.
scale	8-bit integer. Indexes are multiplying by 2^{scale}.

Returns

vector doubleword elements [0,1] loaded from the sign extended words at array[vra[0,1]<<scale].

7.9.6.27 vec_vglswdx()

```
static vi64_t vec_vglswdx (
            signed int * array,
            vi64_t vra )  [inline], [static]
```

Vector Gather-Load Signed Words from Vector Doubleword Indexes.

For each doubleword element [i] of vra, load the sign extended word element at array[vra[i]]. Merge doubleword elements [0,1] and return the resulting vector.

Note

As effective address calculation is modulo 64-bits, signed or unsigned doubleword offsets are equivalent.

processor	Latency	Throughput
power8	14-23	1/cycle
power9	13-22	1/cycle

Parameters

array	Pointer to array of signed words.
vra	Vector of doubleword indexes from &array.

Returns

vector doubleword elements [0,1] loaded from sign extended words at array[vra[0,1]].

7.9.6.28 vec_vglswso()

```
static vi64_t vec_vglswso (
            signed int * array,
            const long long offset0,
            const long long offset1 )  [inline], [static]
```

Vector Gather-Load Signed Word from Scalar Offsets.

For each scalar offset[0|1], load the signed word (sign extended) from the effective address formed by *(char*)array+offset[0|1]. Merge resulting doubleword elements and return the resulting vector.

processor	Latency	Throughput
power8	7	1/cycle
power9	8	1/cycle

Parameters

array	Pointer to array of words.
offset0	Scalar (64-bit) byte offsets from &array.
offset1	Scalar (64-bit) byte offsets from &array.

Returns

vector doubleword elements [0,1] loaded from sign extend words at ∗(char∗)array+offset[0,1].

7.9.6.29 vec_vgluwdo()

```
static vui64_t vec_vgluwdo (
             unsigned int * array,
             vi64_t vra )  [inline], [static]
```

Vector Gather-Load Unsigned Words from Vector Doubleword Offsets.

For each doubleword element [0,1] of vra, load the zero extended word element at ∗(char∗)array+vra[0,1]. Merge those doubleword elements [0,1] and return the resulting vector.

processor	Latency	Throughput
power8	12	1/cycle
power9	11	1/cycle

Parameters

array	Pointer to array of unsigned words.
vra	Vector of doubleword (64-bit) byte offsets from &array.

Returns

vector doubleword elements [0,1] loaded from zero extended words at ∗(char∗)array+vra[0,1].

7.9.6.30 vec_vgluwdsx()

```
static vui64_t vec_vgluwdsx (
             unsigned int * array,
             vi64_t vra,
             const unsigned char scale )  [inline], [static]
```

Vector Gather-Load Unsigned Words from Vector Doubleword Scaled Indexes.

For each doubleword element [0,1] of vra, load the zero extended word element at array[vra[0,1] << scale)]. Merge doubleword elements [0,1] and return the resulting vector.

processor	Latency	Throughput
power8	14-23	1/cycle
power9	13-22	1/cycle

Parameters

array	Pointer to array of unsigned words.
vra	Vector of doubleword indexes from &array.
scale	8-bit integer. Indexes are multiplying by 2^{scale}.

Returns

vector doubleword elements [0,1] loaded from zero extended words at array[vra[0,1]<<scale].

7.9.6.31 vec_vgluwdx()

```
static vui64_t vec_vgluwdx (
            unsigned int * array,
            vi64_t vra )  [inline], [static]
```

Vector Gather-Load Unsigned Words from Vector Doubleword Indexes.

For each doubleword element [0,1] of vra, load the zero extended word element at array[vra[0,1]]. Merge those doubleword elements [0,1] and return the resulting vector.

processor	Latency	Throughput
power8	14-23	1/cycle
power9	13-22	1/cycle

Parameters

array	Pointer to array of unsigned words.
vra	Vector of doubleword indexes from &array.

Returns

Vector doubleword [0,1] loaded from zero extended words at array[vra[0,1]].

7.9.6.32 vec_vgluwso()

```
static vui64_t vec_vgluwso (
            unsigned int * array,
            const long long offset0,
            const long long offset1 )  [inline], [static]
```

Vector Gather-Load Unsigned Word from Scalar Offsets.

For each scalar offset[0,1], load the unsigned word (zero extended) from the effective address formed by ∗(char∗)array+offset[0,1] Merge resulting doubleword [0,1] elements and return the resulting vector.

processor	Latency	Throughput
power8	7	1/cycle
power9	8	1/cycle

Parameters

array	Pointer to array of words.
offset0	Scalar (64-bit) byte offsets from &array.
offset1	Scalar (64-bit) byte offsets from &array.

Returns

vector doubleword elements [0,1] loaded from zero extened words at ∗(char∗)array+offset[0,1].

7.9.6.33 vec_vlxsiwax()

```
static vi64_t vec_vlxsiwax (
            const signed long long ra,
            const signed int * rb )  [inline], [static]
```

Vector Load Scalar Integer Word Algebraic Indexed.

Load the left most doubleword of vector **xt** as a scalar sign extended word from the effective address formed by **rb+ra**. The operand **rb** is a pointer to an array of words. The operand **ra** is a doubleword integer byte offset from **rb**. The result **xt** is returned as a vi64_t vector. For best performance **rb** and **ra** should be word aligned (integer multiple of 4).

Note

The right most doubleword of vector **xt** is left *undefined* by this operation.

This operation is an alternate form of Vector Load Element (vec_lde), with the added simplification that data is always left justified in the vector. Another advantage for Power8 and later, the lxsiwax instruction combines load with sign extend word and can load directly into any of the 64 VSRs. Both simplify merging elements for gather operations.

Note

The lxsiwax instruction was introduced in PowerISA 2.07 (POWER8). Power7 and earlier will use lvewx.

processor	Latency	Throughput
power8	5	2/cycle
power9	5	2/cycle

Parameters

ra	const doubleword index (offset/displacement).
rb	const word pointer to an array of integers.

Returns

The word stored at (ra + rb) is sign extended and loaded into vector doubleword element 0. Element 1 is undefined.

7.9.6.34 vec_vlxsiwzx()

```
static vui64_t vec_vlxsiwzx (
             const signed long long ra,
             const unsigned int * rb )  [inline], [static]
```

Vector Load Scalar Integer Word and Zero Indexed.

Load the left most doubleword of vector **xt** as a scalar unsigned word (zero extended to doubleword) from the effective address formed by **rb+ra**. The operand **rb** is a pointer to an array of words. The operand **ra** is a doubleword integer byte offset from **rb**. The result **xt** is returned as a vui64_t vector. For best performance **rb** and **ra** should be word aligned (integer multiple of 4).

Note

the right most doubleword of vector **xt** is left *undefined* by this operation.

This operation is an alternate form of Vector Load Element (vec_lde), with the added simplification that data is always left justified in the vector. Another advantage for Power8 and later, the lxsiwzx instruction combines load with zero extend word and can load directly into any of the 64 VSRs. Both simplify merging elements for gather operations.

Note

The lxsiwzx instruction was introduced in PowerISA 2.07 (POWER8). Power7 and earlier will use lvewx.

processor	Latency	Throughput
power8	5	2/cycle
power9	5	2/cycle

Parameters

ra	const doubleword index (offset/displacement).
rb	const word pointer to an array of integers.

Returns

The word stored at (ra + rb) is zero extended and loaded into vector doubleword element 0. Element 1 is undefined.

7.9.6.35 vec_vmadd2euw()

```
static vui64_t vec_vmadd2euw (
            vui32_t a,
            vui32_t b,
            vui32_t c,
            vui32_t d )  [inline], [static]
```

Vector Multiply-Add2 Even Unsigned Words.

Note

this implementation exists in vec_int64_ppc::h::vec_vmadd2euw() as it requires vec_addudm().

7.9.6.36 vec_vmadd2ouw()

```
static vui64_t vec_vmadd2ouw (
            vui32_t a,
            vui32_t b,
            vui32_t c,
            vui32_t d )  [inline], [static]
```

Vector Multiply-Add2 Odd Unsigned Words.

Note

this implementation exists in vec_int64_ppc::h::vec_vmadd2ouw() as it requires vec_addudm().

7.9.6.37 vec_vmaddeuw()

```
static vui64_t vec_vmaddeuw (
            vui32_t a,
            vui32_t b,
            vui32_t c )  [inline], [static]
```

Vector Multiply-Add Even Unsigned Words.

Note

this implementation exists in vec_int64_ppc::h::vec_vmaddeuw() as it requires vec_addudm().

7.9.6.38 vec_vmaddouw()

```
static vui64_t vec_vmaddouw (
            vui32_t a,
            vui32_t b,
            vui32_t c )  [inline], [static]
```

Vector Multiply-Add Odd Unsigned Words.

Note

this implementation exists in vec_int64_ppc::h::vec_vmaddouw() as it requires vec_addudm().

7.9.6.39 vec_vmsumuwm()

```
static vui64_t vec_vmsumuwm (
            vui32_t a,
            vui32_t b,
            vui64_t c )  [inline], [static]
```

Vector Multiply-Sum Unsigned Word Modulo.

Note

this implementation exists in vec_int64_ppc::h::vec_vmsumuwm() as it requires vec_addudm().

7.9.6.40 vec_vmuleuw()

```
static vui64_t vec_vmuleuw (
            vui32_t vra,
            vui32_t vrb ) [inline], [static]
```

Vector Multiply Even Unsigned words.

Multiply the even words of two vector unsigned int values and return the unsigned long product of the even words.

For POWER8 and later we can use the vmuleuw instruction. But for POWER7 and earlier we have to construct word multiplies from two halfword multiplies (vmuleuh and vmulouh). Then sum the partial products for the final doubleword results. This is complicated by the fact that vector add doubleword is not available for POWER7. So we need to construct the doubleword add from Vector Add Unsigned Word Modulo (vadduwm) and Vector Add and Write Carry-Out Unsigned Word (vaddcuw) with shift double quadword to reposition the low word carry and a final vadduwm to complete the carry propagation for the doubleword add.

Note

This implementation is NOT endian sensitive and the function is stable across BE/LE implementations.

processor	Latency	Throughput
power8	7	2/cycle
power9	7	2/cycle

Parameters

vra	128-bit vector unsigned int.
vrb	128-bit vector unsigned int.

Returns

vector unsigned long product of the even words of a and b.

7.9.6.41 vec_vmulouw()

```
static vui64_t vec_vmulouw (
            vui32_t vra,
            vui32_t vrb ) [inline], [static]
```

Vector Multiply Odd Unsigned Words.

Multiply the odd words of two vector unsigned int values and return the unsigned long product of the odd words.

For POWER8 and later we can use the vmulouw instruction. But for POWER7 and earlier we have to construct word multiplies from two halfword multiplies (vmuleuh and vmulouh). Then sum the partial products for the final doubleword results. This is complicated by the fact that vector add doubleword is not available for POWER7. So we need to construct the doubleword add from Vector Add Unsigned Word Modulo (vadduwm) and Vector Add and Write Carry-Out Unsigned Word (vaddcuw) with shift double quadword to reposition the low word carry and a final vadduwm to complete the carry propagation for the doubleword add.

processor	Latency	Throughput
power8	7	2/cycle
power9	7	2/cycle

Parameters

vra	128-bit vector unsigned int.
vrb	128-bit vector unsigned int.

Returns

vector unsigned long product of the odd words of a and b.

7.9.6.42 vec_vsst4wso()

```
static void vec_vsst4wso (
            vui32_t xs,
            unsigned int * array,
            const long long offset0,
            const long long offset1,
            const long long offset2,
            const long long offset3 )  [inline], [static]
```

Vector Scatter-Store 4 words to Scalar Offsets.

For each word element [i] of xs, store the element xs[i] at ∗(char∗)array+offset[i].

processor	Latency	Throughput
power8	6	1/cycle
power9	4	2/cycle

Parameters

xs	Vector integer word elements to scatter store.
array	Pointer to array of integer words.
offset0	Scalar (64-bit) byte offset from &array.
offset1	Scalar (64-bit) byte offset from &array.
offset2	Scalar (64-bit) byte offset from &array.
offset3	Scalar (64-bit) byte offset from &array.

7.9.6.43 vec_vsst4wwo()

```
static void vec_vsst4wwo (
            vui32_t xs,
            unsigned int * array,
            vi32_t vra )  [inline], [static]
```

Vector Scatter-Store 4 words to Vector Word Offsets.

For each word element [i] of xs, store the element xs[i] at ∗(char∗)array+vra[i].

Note

> Signed word offsets are expanded (unpacked) to doublewords before transfer to GRPs for effective address calculation.

processor	Latency	Throughput
power8	10	1/cycle
power9	12	2/cycle

Parameters

xs	Vector integer word elements to scatter store.
array	Pointer to array of integer words.
vra	Vector of signed word (32-bit) byte offsets from &array.

7.9.6.44 vec_vsst4wwsx()

```
static void vec_vsst4wwsx (
            vui32_t xs,
            unsigned int * array,
            vi32_t vra,
            const unsigned char scale )  [inline], [static]
```

Vector Scatter-Store 4 words to Vector Word Indexes.

For each word element [i] of xs, store the element xs[i] at ∗(char∗)array[vra[i]<<scale].

Note

> Signed word indexes are expanded (unpacked) to doublewords before shifting left (2+scale) bits before transfer to GRPs for effective address calculation. This converts each index to an 64-bit offset.

processor	Latency	Throughput
power8	12-21	1/cycle
power9	15-24	2/cycle

Parameters

xs	Vector integer word elements to scatter store.
array	Pointer to array of integer words.
vra	Vector of signed word (32-bit) indexes from array.
scale	8-bit integer. Indexes are multiplying by 2^{scale}.

7.9.6.45 vec_vsst4wwx()

```
static void vec_vsst4wwx (
             vui32_t xs,
             unsigned int * array,
             vi32_t vra )  [inline], [static]
```

Vector Scatter-Store 4 words to Vector Word Indexes.

For each word element [i] of xs, store the element xs[i] at ∗(char∗)array[vra[i]].

Note

> Signed word indexes are expanded (unpacked) to doublewords before shifting left 2 bits. This converts each index to an 64-bit offset for effective address calculation.

processor	Latency	Throughput
power8	12-21	1/cycle
power9	15-24	2/cycle

Parameters

xs	Vector doubleword elements to scatter store.
array	Pointer to array of integer words.
vra	Vector of signed word (32-bit) indexes from array.

7.9.6.46 vec_vsstwdo()

```
static void vec_vsstwdo (
            vui64_t xs,
            unsigned int * array,
            vi64_t vra )  [inline], [static]
```

Vector Scatter-Store Words to Vector Doubleword Offsets.

For each doubleword element [i] of vra, Store the low order word element xs[i+1] at ∗(char∗)array+offset[0|1].

processor	Latency	Throughput
power8	8	1/cycle
power9	9	2/cycle

Parameters

xs	Vector doubleword elements to scatter store low order words of each doubleword.
array	Pointer to array of integer words.
vra	Vector of doubleword (64-bit) byte offsets from &array.

7.9.6.47 vec_vsstwdsx()

```
static void vec_vsstwdsx (
            vui64_t xs,
            unsigned int * array,
            vi64_t vra,
            const unsigned char scale )  [inline], [static]
```

Vector Scatter-Store Words to Vector Doubleword Scaled Indexes.

For each doubleword element [i] of vra, Store the low order word element xs[i+1] at array[vra[i]<<scale].

processor	Latency	Throughput
power8	10-19	1/cycle
power9	10-19	1/cycle

Parameters

xs	Vector doubleword elements to scatter store low order words of each doubleword.
array	Pointer to array of integer words.
vra	Vector of doubleword (64-bit) indexes from &array.
scale	8-bit integer. Indexes are multiplying by 2^{scale}.

7.9.6.48 vec_vsstwdx()

```
static void vec_vsstwdx (
             vui64_t xs,
             unsigned int * array,
             vi64_t vra )  [inline], [static]
```

Vector Scatter-Store Words to Vector Doubleword Indexes.

For each doubleword element [i] of vra, Store the low order word element xs[i+1] at array[vra[i]].

processor	Latency	Throughput
power8	10-19	1/cycle
power9	10-19	1/cycle

Parameters

xs	Vector doubleword elements to scatter store low order words of each doubleword.
array	Pointer to array of integer words.
vra	Vector of doubleword (64-bit) indexes from &array.

7.9.6.49 vec_vsstwso()

```
static void vec_vsstwso (
             vui64_t xs,
             unsigned int * array,
             const long long offset0,
             const long long offset1 )  [inline], [static]
```

Vector Scatter-Store Words to Scalar Offsets.

For each doubleword element [i] of vra, Store the low order word element xs[i+1] at ∗(char∗)array+offset[0|1].

processor	Latency	Throughput
power8	3	1/cycle
power9	3	2/cycle

Parameters

xs	Vector doubleword elements to scatter store low order words of each doubleword.
array	Pointer to array of integer words.
offset0	Scalar (64-bit) byte offset from &array.
offset1	Scalar (64-bit) byte offset from &array.

7.9.6.50 vec_vstxsiwx()

```
static void vec_vstxsiwx (
             vui32_t xs,
             const signed long long ra,
             unsigned int * rb )  [inline], [static]
```

Vector Store Scalar Integer Word Indexed.

Stores word element 1 of vector **xs** as a scalar word at the effective address formed by **rb+ra**. The operand **rb** is a pointer to an array of integer words. The operand **ra** is a doubleword integer byte offset from **rb**. For best performance **rb** and **ra** should be word aligned (integer multiple of 4).

This operation is an alternate form of vector store element (vec_ste), with the added simplification that data is always left justified in the vector. Another advantage for Power8 and later, the stxsiwx instruction can load directly into any of the 64 VSRs. Both simplify scatter operations.

Note

The stxsiwx instruction was introduced in PowerISA 2.07 (POWER8). Power7 and earlier will use stvewx.

processor	Latency	Throughput
power8	0 - 2	2/cycle
power9	0 - 2	4/cycle

Parameters

xs	vector doubleword element 0 to be stored.
ra	const doubleword index (offset/displacement).
rb	const doubleword pointer to an array of doubles.

7.9.6.51 vec_vsum2sw()

```
static vi32_t vec_vsum2sw (
             vi32_t vra,
             vi32_t vrb )  [inline], [static]
```

Vector Sum-across Half Signed Word Saturate.

Sum across adjacent signed words within doublewords from *vra* and word addends from *vrb*. This is effectively the vec_sum2s built-in operation (vsum2sws instruction) without the endian sensitive modifications mandated by the ABI.

This is useful for computing the final doubleword counts for operations like population count and count leading/trailing zeros. These results are often used as inputs to shift operations that require shift counts in bits 58:63 of the doubleword element (word elements 1 and 3).

For vec_sum2s and little endian the ABI mandates that the addend words from vrb be from little endian word elements 1 and 3 (vector element 0 and 2) be used for the sum. The ABI also mandates that saturated word sum results are are in little endian elements 1 and 3 (vector element 0 and 2). This requires a 3 instruction dependent sequence to precondition vrb and and rotate the vsum2sws result to match little endian element numbering. This adds 4 (6 for POWER9) cycles latency.

This also leaves the sums in bits 26:31 of the doubleword element and out of position for doubleword shift/rotate. This in turn requires an additional corrective shift/rotate before using the sums. Or use this operation instead of vec_sum2s.

Note

This implementation is NOT endian sensitive and the function is stable across BE/LE implementations.

processor	Latency	Throughput
power8	7	2/cycle
power9	7	2/cycle

Parameters

vra	Vector signed int as adjcent words within doublewords.
vrb	Vector signed int where odd words are summed with adjacent words from vra.

Returns

Vector signed int with even words set to 0 and odd words containing the word sums within doublewords.

7.9.6.52 vec_vsumsw()

```
static vi32_t vec_vsumsw (
            vi32_t vra,
            vi32_t vrb )  [inline], [static]
```

Vector Sum-across Signed Word Saturate.

Sum across the 4 signed words from *vra* and word element 3 from *vrb*. This is effectively the vec_sums built-in operation (vsumsws instruction) without the endian sensitive modifications mandated by the ABI.

This is useful for computing the final quadword counts for operations like population count and count leading/trailing zeros. These results are often used as inputs to shift operations that require shift counts in bits 121:127 of the quadword (word element 3).

For vec_sums and little endian the ABI mandates that the addend word from vrb be from little endian word elements 3 (vector element 0) be used for the sum. The ABI also mandates that saturated word sum results are are in little endian elements 3 (vector element 0). This requires a 3 instruction dependent sequence to precondition vrb and and rotate the vsumsws result to match little endian element numbering. This adds 4 (6 for POWER9) cycles latency.

This also leaves the sums in bits 25:31 of the quadword and out of position for quadword shift/rotate. This in turn requires an additional corrective shift/rotate before using the sums. Or use this operation instead of vec_sums.

Note

This implementation is NOT endian sensitive and the function is stable across BE/LE implementations.

processor	Latency	Throughput
power8	7	2/cycle
power9	7	2/cycle

Parameters

vra	Vector signed int as words within quadword.
vrb	Vector signed int where word element 3 is summed with words from vra.

Returns

Vector signed int with words 0-2 set to 0 and word element 3 containing the word sums.

7.9.6.53 vec_vupkhsw()

```
static vi64_t vec_vupkhsw (
            vi32_t vra )  [inline], [static]
```

Vector Unpack High Signed Word.

From the word source in vra. For each integer word [i] from 0 to 1, sign extend to 64-bit and place in doubleword element [i] of the result vector.

processor	Latency	Throughput
power8	2	2/cycle
power9	2	2/cycle

Note

This operation is the equivalent of the generic vec_unpackh for type vector signed int. However vec_unpackh (for this type) is not available for _ARCH_PWR7 and earlier versions of GCC. This PVECLIB operation is available to both.

Use vec_vupkhsw naming but only if the compiler does not define it in <altivec.h>.

Parameters

vra	a 128-bit vector treated as 4 x signed integers.

Returns

128-bit vector treated as 2 x signed long long integers.

7.9.6.54 vec_vupkhuw()

```
static vui64_t vec_vupkhuw (
             vui32_t vra )  [inline], [static]
```

Vector Unpack High Unsigned Word.

From the word source in vra. For each integer word [i] from 0 to 1, zero extend to 64-bit and place in doubleword element [i] of the result vector.

processor	Latency	Throughput
power8	2-4	2/cycle
power9	2-4	2/cycle

Note

vec_vupkhuw does not exist in <altivec.h> nor as an instruction is the PowerISA. But it is easy to construct using vec_mergeh and a zero vector.

Parameters

vra	a 128-bit vector treated as 4 x unsigned integers.

Returns

128-bit vector treated as 2 x unsigned long long integers.

7.9.6.55 vec_vupklsw()

```
static vi64_t vec_vupklsw (
             vi32_t vra )  [inline], [static]
```

Vector Unpack Low Signed Word.

From the word source in vra. For each integer word [i+2] from 0 to 1 (words 2 and 3), sign extend to 64-bit and place in doubleword element [i] of the result vector.

processor	Latency	Throughput
power8	2	2/cycle
power9	2	2/cycle

Note

Use vec_vupkhsw naming but only if the compiler does not define it in <altivec.h>.

Parameters

vra	a 128-bit vector treated as 4 x signed integers.

Returns

128-bit vector treated as 2 x signed long long integers.

7.9.6.56 vec_vupkluw()

```
static vui64_t vec_vupkluw (
            vui32_t vra )  [inline], [static]
```

Vector Unpack Low Unsigned Word.

From the word source in vra. For each integer word [i+2] from 0 to 1 (words 2 and 3), zero extend to 64-bit and place in doubleword element [i] of the result vector.

processor	Latency	Throughput
power8	2-4	2/cycle
power9	2-4	2/cycle

Note

vec_vupkluw does not exist in <altivec.h> nor as an instruction is the PowerISA. But it is easy to construct using vec_mergeh and a zero vector.

Parameters

vra	a 128-bit vector treated as 4 x unsigned integers.

Returns

128-bit vector treated as 2 x unsigned long long integers.

7.10 src/pveclib/vec_int512_ppc.h File Reference

Header package containing a collection of multiple precision quadword integer computation functions implemented with 128-bit PowerISA VMX and VSX instructions.

```
#include <pveclib/vec_int128_ppc.h>
```

Classes

- struct __VEC_U_256

  A vector representation of a 256-bit unsigned integer.
- struct __VEC_U_512

  A vector representation of a 512-bit unsigned integer.
- struct __VEC_U_640

  A vector representation of a 640-bit unsigned integer.
- union __VEC_U_512x1

  A vector representation of a 512-bit unsigned integer and a 128-bit carry-out.
- struct __VEC_U_1024

  A vector representation of a 1024-bit unsigned integer.
- struct __VEC_U_1152

  A vector representation of a 1152-bit unsigned integer.
- struct __VEC_U_2048

  A vector representation of a 2048-bit unsigned integer.
- union __VEC_U_1024x512

  A vector representation of a 1024-bit unsigned integer as two 512-bit fields.
- union __VEC_U_2048x512

  A vector representation of a 2048-bit unsigned integer as 4 x 512-bit integer fields.
- struct __VEC_U_2176

  A vector representation of a 2176-bit unsigned integer.
- struct __VEC_U_4096

  A vector representation of a 4096-bit unsigned integer.
- union __VEC_U_4096x512

  A vector representation of a 4096-bit unsigned integer as 8 x 512-bit integer fields.

Macros

- #define CONST_VINT512_Q(__q0, __q1, __q2, __q3) {__q3, __q2, __q1, __q0}

  Generate a 512-bit vector unsigned integer constant from 4 x quadword constants.
- #define COMPILE_FENCE __asm (";":::)

  A compiler fence to prevent excessive code motion.
- #define __VEC_PWR_IMP(FNAME) FNAME ## _PWR7

  Macro to add platform suffix for static calls.

Functions

- static __VEC_U_640 vec_add512cu (__VEC_U_512 a, __VEC_U_512 b)

  Vector Add 512-bit Unsigned Integer & Write Carry.
- static __VEC_U_640 vec_add512ecu (__VEC_U_512 a, __VEC_U_512 b, vui128_t c)

  Vector Add Extended 512-bit Unsigned Integer & Write Carry.
- static __VEC_U_512 vec_add512eum (__VEC_U_512 a, __VEC_U_512 b, vui128_t c)

  Vector Add Extended 512-bit Unsigned Integer Modulo.
- static __VEC_U_512 vec_add512um (__VEC_U_512 a, __VEC_U_512 b)

  Vector Add 512-bit Unsigned Integer Modulo.
- static __VEC_U_512 vec_add512ze (__VEC_U_512 a, vui128_t c)

  Vector Add 512-bit to Zero Extended Unsigned Integer Modulo.
- static __VEC_U_512 vec_add512ze2 (__VEC_U_512 a, vui128_t c1, vui128_t c2)

  Vector Add 512-bit to Zero Extended2 Unsigned Integer Modulo.
- static __VEC_U_256 vec_mul128x128_inline (vui128_t a, vui128_t b)

  Vector 128x128bit Unsigned Integer Multiply.
- static __VEC_U_512 vec_mul256x256_inline (__VEC_U_256 m1, __VEC_U_256 m2)

  Vector 256x256-bit Unsigned Integer Multiply.
- static __VEC_U_640 vec_mul512x128_inline (__VEC_U_512 m1, vui128_t m2)

  Vector 512x128-bit Unsigned Integer Multiply.
- static __VEC_U_640 vec_madd512x128a128_inline (__VEC_U_512 m1, vui128_t m2, vui128_t a1)

  Vector 512x128-bit Multiply-Add Unsigned Integer.
- static __VEC_U_640 vec_madd512x128a512_inline (__VEC_U_512 m1, vui128_t m2, __VEC_U_512 a2)

  Vector 512x128-bit Multiply-Add Unsigned Integer.
- static __VEC_U_640 vec_madd512x128a128a512_inline (__VEC_U_512 m1, vui128_t m2, vui128_t a1, __VEC_U_512 a2)

  Vector 512x128-bit Multiply-Add Unsigned Integer.
- static __VEC_U_1024 vec_mul512x512_inline (__VEC_U_512 m1, __VEC_U_512 m2)

  Vector 512x512-bit Unsigned Integer Multiply.
- static __VEC_U_1024 vec_madd512x512a512_inline (__VEC_U_512 m1, __VEC_U_512 m2, __VEC_U_512 a1)

  Vector 512-bit Unsigned Integer Multiply-Add.
- __VEC_U_256 vec_mul128x128 (vui128_t m1, vui128_t m2)

  Vector 128x128bit Unsigned Integer Multiply.
- __VEC_U_512 vec_mul256x256 (__VEC_U_256 m1, __VEC_U_256 m2)

  Vector 256x256-bit Unsigned Integer Multiply.
- __VEC_U_640 vec_mul512x128 (__VEC_U_512 m1, vui128_t m2)

  Vector 512x128-bit Unsigned Integer Multiply.
- __VEC_U_640 vec_madd512x128a512 (__VEC_U_512 m1, vui128_t m2, __VEC_U_512 a2)

  Vector 512x128-bit Multiply-Add Unsigned Integer.
- __VEC_U_1024 vec_mul512x512 (__VEC_U_512 m1, __VEC_U_512 m2)

  Vector 512x512-bit Unsigned Integer Multiply.
- void vec_mul1024x1024 (__VEC_U_2048 ∗p2048, __VEC_U_1024 ∗m1, __VEC_U_1024 ∗m2)

  Vector 1024x1024-bit Unsigned Integer Multiply.
- void vec_mul2048x2048 (__VEC_U_4096 ∗p4096, __VEC_U_2048 ∗m1, __VEC_U_2048 ∗m2)

  Vector 2048x2048-bit Unsigned Integer Multiply.
- void vec_mul128_byMN (vui128_t ∗p, vui128_t ∗m1, vui128_t ∗m2, unsigned long M, unsigned long N)

  Vector Unsigned Integer Quadword MxN Multiply.
- void vec_mul512_byMN (__VEC_U_512 ∗p, __VEC_U_512 ∗m1, __VEC_U_512 ∗m2, unsigned long M, unsigned long N)

  Vector Unsigned Integer Quadword 4xMxN Multiply.

7.10.1 Detailed Description

Header package containing a collection of multiple precision quadword integer computation functions implemented with 128-bit PowerISA VMX and VSX instructions.

PVECLIB vec_int128_ppc.h provides the 128x128-bit multiply and 128-bit add with carry/extend operations. This is most of what we need to implement multiple precision integer computation. This header builds on those operations to build 256x256, 512x128, 512x512, 1024x1024 and 2048x2048 multiplies. We also provide 512-bit add with carry/extend operations as a general aid to construct multiple quadword precision arithmetic.

We provide static inline implementations for up to 512x512 multiplies and 512x512 add with carry/extend. These in-line operations are provided as building blocks for coding implementations of larger multiply and sum operations. Otherwise the in-line code expansion is getting too large for normal coding. So we also provide callable (static and dynamic) library implementations as well (Building libraries for vec_int512_ppc).

7.10.2 Security related implications

The challenge is delivering a 2048x2048 bit multiply, producing a 4096-bit product, while minimizing cache and timing side-channel exploits. The goal is to minimize the memory visibility of intermediate products and sums and internal conditional logic (like early exit optimizations). The working theory is to use vector registers and operations and avoid storing intermediate results. This implies:

- While the final 4096-bit product is so large (32 quadwords), it requires a memory buffer for the result, we should not use any part of this buffer to hold intermediate partial sums.
- The 2048-bit multiplicands are also large (2 x 16 quadwords) and will be passed in memory buffers that are effectively constant.
- All intermediate partial products and sums should be held in vector registers (VSRs) until quadwords of the final product are computed and ready to store into the result buffer.
- Avoid conditional logic that effects function timing based on values of the inputs or results.
- Internally the code can be organized as straight line code or loops, in-line functions or calls to carefully crafted leaf functions, as long as the above goals are met.

Achieving these goals requires some knowledge of the Application Binary Interface (ABI) and foibles of the Instruction Set Architecture (PowerISA) and how they impact what the compiler can generate. The compiler itself has internal strategies (and foibles) that need to be managed as well.

7.10.2.1 Implications of the ABI

The computation requires a number of internal temporary vectors in addition to the inputs and outputs. The Power Architecture, 64-Bit ELF V2 ABI Specification (AKA the ABI) places some generous but important restrictions on how the compiler generates code (and how compliant assembler code is written).

- Up to 20 volatile vector registers v0-v19 (VSRs vs32-vs51) of which 12 can be used for function arguments/return values.
    - Up to 12 vector arguments are passed in vector registers v2-v13 (VSRs vs34-vs45).

  - Longer vector argument lists are forced into the callers parameter save area (Stack pointer +32).
  - Functions can return a 128-bit vector value or a homogeneous aggregate of up to 8 vector values in vector registers v2-v9 (VSRs 34-41).
  - Wider (8 x vectors) function return values are returned in memory via a reference pointer passed as a hidden parameter in GPR 3.
- Up to 12 additional non-volatile vector registers v20-v31 (vs51-vs63). Any non-volatile registers must be saved before use and restored before function return.
- The lower half for the VSRs (vs0-vs31) are prioritized for scalar floating-point operations. If a function is using vectors and but not scalar floating-point then the lower VSRs are available for vector logical and integer operations and temporary spill from vector registers.
  - Up to 14 volatile float double (f0-f13) or vector registers (vs0-vs13).
  - Up to 18 non-volatile float double (f14-f31) or vector registers (vs14-vs31).
- All volatile registers are a considered “clobbered” after a function call.
  - So the calling function must hold any local vector variables in memory or non-volatile registers if the live range extends across the function call.
  - In-lining the called function allows the compiler to manage register allocation across the whole sequence. This can reduce register pressure when the called function does not actually use/modify all the volatile registers.

7.10.2.1.1 Implications for parameter passing and Product size Care is required in selecting the width (256, 512-bit etc) of parameter and return values. Parameters totaling more then 12 vector quadwords or return values totaling more then 8 vector quadwords will be spilled to the callers parameter save area. This may expose intermediate partial products to cache side-channel attacks. A 512x128-bit multiply returning a 640-bit product and a 512x512-bit multiply returning a 1024-bit product meets this criteria (both the parameters and return values fit within the ABI limits). But a 1024x128-bit multiply returning 1152-bits is not OK because the 1152-bit return value requires 9 vector registers, which will be returned in memory.

Also if any of these sub-functions are used without in-lining, the generated code must be inspected to insure it is not spilling any local variables. In my experiments with GCC 8.1 the 128x128, 256x256, and 512x128 multiplies all avoid spilling. However the stand-alone 512x512 implementation does require saving 3 non-volatile registers. This can be eliminated by in-lining the 512x512 multiply into the 2048x2048 multiply function.

Note

> GCC compilers before version 8 have an incomplete design for homogeneous aggregates of vectors and may generate sub-optimal code for these parameters.

7.10.2.2 Implications of the PowerISA

The Power Instruction Set Architecture (PowerISA) also imposes some restriction on the registers vector instructions can access.

- The original VMX (AKA Altivec) facility has 32 vector registers and instruction encoding to access those 32 registers.
  - This original instruction set was incorporated unchanged into the later versions of the PowerISA.

  - When Vector Scalar Extended facility was added, the original VMX instructions where restricted to the upper 32 VSRs (original vector registers).

- VSX was originally focused on vector and scalar floating-point operations. With a handful of vector logical/permute/splat operations added for completeness. These instructions where encoded to access all 64 VSRs.
  - All vector integer arithmetic operations remained restricted to the upper 32 VSRs (the original VRs).
  - Later versions of the PowerISA (POWER8/9) added new vector integer arithmetic operations. This includes word/doubleword multiply and doubleword/quadword add/subtract. But these are also encoded to access only 32 vector registers.
  - The lower VSRs can still be used hold temporaries and local variables for vector integer operations.

7.10.2.3 Implications for the compiler

The compiler has to find a path though the ABI and ISA restriction above while it performs:

- function in-lining
- instruction selection
- instruction scheduling
- register allocation

For operations defined in PVECLIB, most operations are defined in terms of AltiVec/VSX Built-in Functions. So the compiler does not get much choice for instruction selection. The PVECLIB coding style does leverage C language vector extensions to load constants and manage temporary variables. Using compiler Altivec/VSX built-ins and vector extensions allows the compiler visibility to and control of these optimizations.

Internal function calls effectively *clobber* all (34 VSRs) volatile registers. As the compiler marshals parameters into ABI prescribed VRs it needs to preserve previous live content for later computation. Similarly for volatile registers not used for parameter passing as they are assumed to be clobbered by the called function. The compiler preserves local live variables before the call by copying their contents to non-volatile registers or spilling to memory. This may put more *register pressure* on the available non-volatile registers. Small to medium sized functions often require only a fraction of the available volatile registers. In this case, in-lining the function avoids the disruptive volatile register clobber and allows better overall register allocation. So there is a strong incentive to in-line local/static functions.

These compiler optimizations are not independent processes. For example specific VSX instruction can access all 64 VSRs, others are restricted to the 32 VRs (like vector integer instructions). So the compiler prioritizes VRs (the higher 32 VSRs) for allocation to vector integer computation. While the lower 32 VSRs can be used for logical/permute operations and as a *level 1* spill area for VRs. These restrictions combined with code size/complexity can increase *register pressure* to the point the compiler is forced to spill active (or live) vector registers to secondary storage. This secondary storage can be:

- other architected registers that are available for direct transfer but not usable in the computation.
- Local variables allocated on the stack
- Compiler temporaries allocated on the stack.

Instruction scheduling can increase register pressure by moving (reordering) instructions. This is more prevalent when there are large differences in instruction latency in the code stream. For example moving independent / long latency instructions earlier and dependent / short latency instructions later. This tends to increase the distance between the instruction that sets a register result and the next instruction the uses that result in its computation. The distance between a registers set and use is called the *live range*. This also tends to increase the number of concurrently active and overlapping live ranges.

For this specific (multi-precision integer multiply) example, integer multiple and add/carry/extend instructions predominate. For POWER9, vector integer multiply instructions run 7 cycles, while integer add/carry/extend quadword instruction run 3 cycles. The compiler will want to move the independent multiply instructions earlier while the dependent add/carry instructions are moved later until the latency of the (multiply) instruction (on which it depends) is satisfied. Moving dependent instructions apart and moving independent instructions into the scheduling gap increases register pressure.

In extreme cases, this can get out of hand. At high optimization levels, the compiler can push instruction scheduling to the point that it runs out of registers. This forces the compiler to spill live register values, splitting the live range into two smaller live ranges. Any spilled values have to be reloaded later so they can used in computation. This causes the compiler to generate more instructions that need additional register allocation and scheduling.

Note

> A 2048x2048-bit multiply is definitely an extreme case. The implementation requires 256 128x128-bit multiplies, where each 128x128-bit multiply requires 18-30 instructions. The POWER9 implementation requires 1024 vector doublewword multiplies plus 2400+ vector add/carry/extend quadword instructions. When implemented as straight line code and expanded in-line (*attribute (flatten)*) the total runs over 6000 instructions.

Compiler spill code usually needs registers in addition (perhaps of a different class) to the registers being spilled. This can be as simple as moving to a register of the same size but different class. For example, register moves to/from VRs and the lower 32 VSRs. But it gets more complex when spilling vector registers to memory. For example, vector register spill code needs GPRs to compute stack addresses for vector load/store instructions. Normally this OK, unless the the spill code consumes so many GPRs that it needs to spill GPRs. In that case we can see serious performance bottlenecks.

But remember that a primary goal (Security related implications) was to avoid spilling intermediate results to memory. Spilling between high and low VSRs is acceptable (no cache side-channel), but spilling to memory must be avoided. The compiler should have heuristics to back off in-lining and scheduling-driven code motions just enough to avoid negative performance impacts. But this is difficult to model and may not handle all cases with equal grace. Also this may not prevent spilling VRs to memory if the compiler scheduler's cost computation indicates that is an acceptable trade-off.

So we will have to directly override compiler settings and heuristics to guarantee the result we want/need. The P↩ VECLIB implementation already marks most operations as **static inline**. But as we use these inline operations as building blocks to implement larger operations we can push the resulting code size over the compiler's default inline limits (**-finline-limit**). Then compiler will stop in-lining for the duration of compiling the current function.

This may require stronger options/attributes to the compiler like (*attribute (always_inline)*), (*attribute (gnu_inline)*), or (*attribute (flatten)*). The first two are not any help unless you are compiling at lower optimization level (**-O0** or **-O1**). **-O2** defaults to **-finline-small-functions** and **-O3** defaults to the stronger **-finline-functions**. However *attribute (flatten)* seems do exactly what we want. Every call inside this function is in-lined unless explicitly told not to (*attribute (noinline)*). It seems that *attribute (flatten)* ignores the **-finline-limit**.

Note

> You should be compiling PVECLIB applications at **-O3** anyway.

Now we have a large block of code for the compiler's instruction scheduler to work on. In this case the code is very repetitive (multiply, add the column, generate carries, repeat). The instruction will have lots of opportunity for scheduling long vs short latency instructions and create new and longer live ranges.

/note In fact after applying *attribute (flatten)* to vec_mul2048x2048_PWR9 we see a lot of spill code. This expands the code to over 9300 instructions with ∼3300 instructions associated with spill code.

We need a mechanism to limit (set boundaries) on code motion while preserving optimization over smaller blocks of code. This is normally called a *compiler fence* but there are multiple definitions so we need to be careful what we use.

We want something that will prevent the compiler from moving instructions (in either direction) across specified *lines in the code*.

We don't need an atomic memory fence (like **__atomic_thread_fence** or **__sync_synchronize**) that forces the processor to order loads and stores relative to a specific synchronization point.

We don't need a compiler memory fence (like **asm ("" ::: "memory")**). The "memory" clobber forces GCC to assume that any memory may be arbitrarily read or written by the asm block. So any registers holding live local variables will be forced to memory before and need to be reloaded after. This prevents the compiler from reordering loads, stores, and arithmetic operations across it, but does not prevent the processor from reordering them.

Note

> POWER process have an aggressively *Speculative Superscalar* design with out-of-order issue and execution.

Neither of the above are what we want for this case. We specifically want to avoid memory side effects in this computation. We only need the minimal compiler fence (like **asm (";" :::)**) that prevents the compiler from reordering any code across it but does not prevent the processor from reordering them.

By placing this compiler fence between multiply/sum stages of vec_mul512x128_inline(), vec_mul512x512_inline() and vec_mul2048x2048() we limit instruction scheduling and code motion to smaller code blocks. This in turn reduces register pressure to the point where all 64 VSRs are in use, but no spilling to stack memory is required.

7.10.2.4 So what does this all mean?

The 2048x2048 multiplicands and the resulting product are so large (8192-bits, 64 quadwords total) that at the outer most function the inputs and the result must be in memory and passed by reference. The implementation of a 2048x2048-bit multiply requires 256 128x128-bit multiplies. Otherwise the code can be organized into sub-functions generating intermediate partial products and sums.

Coding 256 128x128 products and generating column sums would be tedious. One approach builds up products into larger and larger blocks in stages. For example code a vec_mul512x128_inline() operation then use that in the implementation of vec_mul512x512_inline(). We also provide 512-bit add/carry/extend operations to simplify generating sums of 512-bit partial products. Then load blocks of 512-bits (4 quadwords, 64-bytes) using vec_mul512x512_inline() to produce a 1024-bit partial product (Implications for parameter passing and Product size).

Then multiply the 512-bit blocks across one 2048-bit (4 x 512-bit) multiplicand. The completion of a 2048x512-bit partial product (of 2560-bits) includes the low order 512-bits ready to store to the output operand. Repeat for each 512-bit block of the other 2048-bit multiplicand summing across the 512-bit columns. The final sum, after the final 2048x512 partial product, produces the high order 2048-bits of the 2048x2048 product ready to store to the output operand.

Note

> Security aware implementations could use masking countermeasures associated with these load/store operations. The base PVECLIB implementation does not do this. The source is available in ./src/vec_int512_runtime.c.

It is best if the sub-functions code can be fully in-lined into the 2048x2048-bit multiply or the sub-functions are carefully written. In this case these sub-functions should be leaf-functions (does not call other functions) and can execute without spilling register state or requiring stored (by reference) parameters.

All levels of implementation should avoid conditional logic based on values of inputs or partial products (For example early exits for leading or trailing zero quadwords). Doing so may expose the multiply function to timing side-channel attacks. So the best case would be one large function implemented as straight-line code.

We will need all 64 VSX registers for operations and local variables. So the outer function will need to allocate a stack-frame and save all of the non-volatile floating point registers (allowing the use of vs14-vs31 for local vector variables) and vector registers (v20-v31 AKA vs51-vs63) on entry. This frees up (18+12=) 30 additional quadword registers for local vector variables within the outer multiply function.

These saved registers reflect the state of the calling (or higher) function and may not have any crypto sensitive content. These register save areas will not be updated with internal state from the 2048x2048-bit multiply operation itself.

The 128x128-bit vector multiply is implemented with Vector Multiply-Sum Unsigned Doubleword Modulo for Power9 and Vector Multiply Even/Odd Unsigned Word for Power8. The timing for vector integer multiply operations are fixed at 7 cycles latency for Power8/9. The sums of partial products are implemented with Vector Add Unsigned Quadword Modulo/write-Carry/Extended. The timing of integer add quadword operations are fixed at 4 cycles for Power8 and 3 cycles for Power9. The rest of the 128x128-bit multiply operation is a combination of Vector Doubleword Permute Immediate, Vector Shift Left Double by Octet Immediate, Vector Splats, and Vector Logical Or (used as a vector register move spanning the 64 VSRs). All of these have fixed timings of 2 or 3 cycles.

So the overall timing of the 2048x2048-bit multiply should be consistent independent of input values. The only measurable variations would be as the processor changes Simultaneous Multithreading (SMT) modes (controlled by the virtual machine and kernel). The SMT mode (1,2,4,8) controls each hardware thread's priority to issue instructions to the core and if the instruction stream is dual or single issue (from that thread's perspective).

But the better news is that with some extra function attributes (always_inline and flatten) the entire 2048x2048 multiply function can be flattened into a single function of straight line code (no internal function calls or conditional branches) running ∼6.3K instructions. And no spill code was generated for local variables (no register spill within the function body).

7.10.3 Endian for Multi-quadword precision operations

As described in General Endian Issues and Endian problems with quadword implementations supporting both big and little endian in a single implementation has its challenges. But I think we can leave the details of quadword operations to the vec_int128_ppc.h implementation. The decision needed for these implementations is how the quadwords of a multi-quadword integer are ordered in storage. For example given an array or structure of 16 quadwords representing a single 2048-bit binary number which quadword contains the low order bits and which the high order bits.

This is largely arbitrary and independent from the system endian. But we should be consistent within the API defined by this header and PVECLIB as a whole. Placing the low order bits in the first (lowest address in memory) quadword and the high order bits in last (highest address in memory) quadword would be consistent with little endian. While placing the high order bits in the first (lowest address in memory) quadword and the low order bits in last (highest address in memory) quadfword would be consistent with big endian. Either is valid internal to the implementation where the key issue is accessing the quadwords of the multiplicands is a convenient order to generate the partial products in an order that support efficient generation of column sums and carries.

It is best for the API if the order of quadwords in multi-quadword integers match the endian of the platform. This should be helpful where we want the use the PVECLIB implementations under existing APIs using arrays of smaller integer types.

So on powerpc64le systems the low order quadword is the first quadword. While on older powwerpc64 systems the high order quadword is the first quadword. For example we can represent a 512-bit integer with the following structure.

```
 typedef struct
{
#if __BYTE_ORDER__ == __ORDER_LITTLE_ENDIAN__
  vui128_t vx0;
  vui128_t vx1;
  vui128_t vx2;
  vui128_t vx3;
#else
  vui128_t vx3;
  vui128_t vx2;
  vui128_t vx1;
  vui128_t vx0;
#endif
} __VEC_U_512;
```

In this example the field *vx0* is always the low order quadword and *vx3* is always the high order quadword, independent of endian. We repeat this pattern for the range of multi-quadword integer sizes (from __VEC_U_256 to __VEC_U_4096) supported by this header. In each case the field name *vx0* is consistently the low order quadword. The field name suffix numbering continues from low to high with the highest numbered field name being the high order quadword.

7.10.3.1 Multi-quadword Integer Constants

As we have seen, initializing larger multiple precision constants an be challenging (Quadword Integer Constants). The good news we can continue to to use aggregate initializers for structures and arrays of vector quadwords. For example:

```
const __VEC_U_512 vec512_one =
    {
      (vui128_t) ((unsigned __int128) 0x00000000),
      (vui128_t) ((unsigned __int128) 0x00000000),
      (vui128_t) ((unsigned __int128) 0x00000000),
      (vui128_t) ((unsigned __int128) 0x00000001)
    };
```

This example is in the expected high to low order for the 512-bit constant 1. Unfortunately endian raises it ugly head again and this would a different value on little endian platform.

So PVECLIB provides another helper macro (CONST_VINT512_Q()) to provide a consistent numbericial order for multiple quadword constants. For example:

```
const __VEC_U_512 vec512_one = CONST_VINT512_Q
    (
      (vui128_t) ((unsigned __int128) 0x00000000),
      (vui128_t) ((unsigned __int128) 0x00000000),
      (vui128_t) ((unsigned __int128) 0x00000000),
      (vui128_t) ((unsigned __int128) 0x00000001)
    );
```

and

```
// const for 10**128
const __VEC_U_512  vec512_ten128th = CONST_VINT512_Q
    (
     CONST_VUINT128_QxW (0x00000000, 0x00000000, 0x0000024e, 0xe91f2603),
     CONST_VUINT128_QxW (0xa6337f19, 0xbccdb0da, 0xc404dc08, 0xd3cff5ec),
     CONST_VUINT128_QxW (0x2374e42f, 0x0f1538fd, 0x03df9909, 0x2e953e01),
     CONST_VUINT128_QxW (0x00000000, 0x00000000, 0x00000000, 0x00000000)
    );
```

Unfortunately the compiler can not help with multi-quadword decimal constants. So we must resort to external tools like **bc** to compute large constant values and convert them to hexadecimal which are easier to break into words and doubleword. These can then be used a constants in program source to represent arbitrarily large binary values.

7.10.4 Building libraries for vec_int512_ppc

See also

Putting the Library into PVECLIB

Many of the implementations associated with 512-bit integer operations are uncomfortably large to expand as in-line code (Examples include vec_mul512x512(), vec_mul1024x1024(), and vec_mul2048x2048()). It is better to collect these large implementations in separately compiled run-time libraries. Another consideration is that most of these operations are multiple quadword multiplies and the optimum quadword multiply is processor (and PowerISA version) dependent. This is especially true for Vector integer multiplies across POWER7-POWER9.

This places requirements on the structure of runtime implementation codes and the library build process.

- Building a set of source implementations for multiple compile (-mcpu=) targets.
- Providing unique function names based on the operation and the compile target.
- Providing static (archive) and dynamic (DSO) libraries, while adjusting the the compile options appropriately for each.
  - Objects compiled for inclusion in dynamic libraries should be position independent code (i.e. compiled with -fpic or -fPIC).
  - DSOs supporting operations optimized for multiple compile (-mcpu=) targets need to export matching `IFUNC` symbols and resolver stubs.

For the first requirement we can collect the runtime implementations for vec_int512_ppc in to a single source file (vec_int512_runtime.c). The build system can then collect this and other runtime source files to compile for different targets. This can be as simple as:

```
//  \file  vec_runtime_PWR9.c
#include "vec_int512_runtime.c"
...
```

and similarly for vec_runtime_PWR7.c and vec_runtime_PWR8.c.

As the implementation of vec_int512_ppc.c is already leveraging _ARCH_PWR7/8/9 tuned static inline operations from vec_int512_ppc.h, vec_int128_ppc.h, etc, all we need to do is apply the appropriate -mcpu=power7/8/9 compile option to each (target qualified) runtime source file.

The second requirement is addressed by applying a target qualifying suffix to each runtime function implementation. Here we use the __VEC_PWR_IMP() as function name wrapper macro.

```
#ifdef _ARCH_PWR10
#define __VEC_PWR_IMP(FNAME) FNAME ## _PWR10
#else
#ifdef _ARCH_PWR9
#define __VEC_PWR_IMP(FNAME) FNAME ## _PWR9
#else
#ifdef _ARCH_PWR8
#define __VEC_PWR_IMP(FNAME) FNAME ## _PWR8
#else
#define __VEC_PWR_IMP(FNAME) FNAME ## _PWR7
#endif
#endif
#endif
```

We need to apply the name wrapper to both the functions extern (in vec_int512_ppc.h) and the function implementation (in vec_int512_runtime.c). For example:

```
//  \file  vec_int512_ppc.h
 ...
extern __VEC_U_256
__VEC_PWR_IMP (vec_mul128x128) (vui128_t m1l, vui128_t m2l);
 ...
```

Note

> Doxygen does not tolerate attributes or macros in function prototypes. So these externs are guarded by a @cond INTERNAL ... @endcond" block. The \brief and @param descriptions are provided for the unqualified dynamic function symbol and apply to the corresponding qualified function symbols.

```
//  \file  vec_int512_runtime.c
#include <altivec.h>
#include <pveclib/vec_int128_ppc.h>
#include <pveclib/vec_int512_ppc.h>
 ...
// vec_mul128x128_inline is defined in vec_int512_ppc.h
__VEC_U_256
__VEC_PWR_IMP (vec_mul128x128) (vui128_t m1l, vui128_t m2l)
{
  return vec_mul128x128_inline (m1l, m2l);
}
```

This ensures that target specific runtime implementations have unique function symbols. This is important to avoid linker errors (due to duplicate symbol names).

Note

> Each runtime operation will have 2 or 3 target qualified implementations. This is times 2 with separate builds for static archives and dynamic (DSO) libraries. The big endian powerpc64 platform supports 3 VSX enabled targets -mcpu=[power7|power8|power9]. The little endian powerpc64le platform currently supports 2 VSX enabled targets -mcpu=[power8|power9]. POWER7 is not supported for powerpc64le and the vec_runtime_PWR7.c source files are conditionally nulled out for powerpc64le targets. As new POWER processors are released, additional targets will be added.

7.10.4.1 Static linkage to platform specific functions

For static linkage the application is compiled for a specific platform target (via -mcpu=). So function calls should be bound to the matching platform specific implementations. The application may select the platform specific function directly by defining a *extern* and invoking the platform qualified function.

For applications binding to PVECLIB via static archives it is convenient to apply the __VEC_PWR_IMP() wrapper to the function call:

```
k = __VEC_PWR_IMP (vec_mul128x128)(i, j);
```

The function call symbol picks up the target suffix based on the compile target (-mcpu=) for the application (see Static linkage to platform specific functions). The linker will extract the matching implementations from the PVECLIB archive and (statically) bind them with the application. This simplifies binding the application to the matching target specific implementations.

7.10.4.2 Dynamic linkage to platform specific functions

For applications binding to dynamic libraries, the target qualified naming strategy also simplifies the implementation of IFUNC resolvers for the DSO library (see Building dynamic runtime libraries). Here the target qualified names of the PIC implementations are known to the corresponding resolver function but are not exported from the DSO. Allowing the application to bind to the target qualified names would defeat the automatic selection of target optimized implementations.

Applications using dynamic linkage will call the unqualified function symbol. For example:

```
//  \file  vec_int512_ppc.h
 ...
extern __VEC_U_256
```

```
vec_mul128x128 (vui128_t, vui128_t);
```

This symbol's implementation has a special **STT_GNU_IFUNC** attribute recognized by the dynamic linker. This attribute associates this symbol with the corresponding runtime resolver function. So in addition to any platform specific implementations we need to provide the resolver function referenced by the *IFUNC* symbol. For example:

```
//  \file  vec_runtime_DYN.c
 ...
extern __VEC_U_256
vec_mul128x128_PWR7 (vui128_t, vui128_t);
extern __VEC_U_256
vec_mul128x128_PWR8 (vui128_t, vui128_t);
extern __VEC_U_256
vec_mul128x128_PWR9 (vui128_t, vui128_t);
static __VEC_U_256
(*resolve_vec_mul128x128 (void))(vui128_t, vui128_t)
{
#ifdef  __BUILTIN_CPU_SUPPORTS__
  if (__builtin_cpu_is ("power9"))
    return vec_mul128x128_PWR9;
  else
    {
      if (__builtin_cpu_is ("power8"))
        return vec_mul128x128_PWR8;
      else
        return vec_mul128x128_PWR7;
    }
#else // ! __BUILTIN_CPU_SUPPORTS__
    return vec_mul128x128_PWR7;
#endif
}
__VEC_U_256
vec_mul128x128 (vui128_t, vui128_t)
__attribute__ ((ifunc ("resolve_vec_mul128x128")));
```

On the program's first call to a *IFUNC* symbol, the dynamic linker calls the resolver function associated with that symbol. The resolver function performs a runtime check to determine the platform, selects the (closest) matching platform specific function, then returns that functions address to the dynamic linker.

The dynamic linker stores this function address in the callers Procedure Linkage Tables (PLT) before forwarding the call to the resolved implementation. Any subsequent calls to this function symbol branch (via the PLT) directly to appropriate platform specific implementation.

Note

> The operation vec_mul128x128() has multiple implementations and names. It has a static inline implementation vec_mul128x128_inline(). This uses the static inline vec_muludq() from _vec_int128_ppc.h but returns the 256-bit result as a single struct __VEC_U_256. It has a number (currently 2 or 3) of target qualified extern declarations and static implementations for static linkage. And it has a unqualified extern declaration and IFUNC attributed symbol associated with its resolver for dynamic linkage.

Todo Currently the dynamic resolvers and *IFUNC* symbols for vec_int512_runtime.c are contained within vec_runtime_DYN.c. As the list of runtime operations expands to other element sizes/types, vec_runtime_DYN.c should be refactored into multiple files.

7.10.5 Macro Definition Documentation

7.10.5.1 COMPILE_FENCE

```
#define COMPILE_FENCE __asm (";":::)
```

A compiler fence to prevent excessive code motion.

We use the COMPILER_FENCE to limit instruction scheduling and code motion to smaller code blocks. This in turn reduces register pressure and avoids generating spill code.

7.10.5.2 CONST_VINT512_Q

```
#define CONST_VINT512_Q(
                __q0,
                __q1,
                __q2,
                __q3 ) {__q3, __q2, __q1, __q0}
```

Generate a 512-bit vector unsigned integer constant from 4 x quadword constants.

Combine 4 x quadwords constants into a 512-bit __VEC_U_512 constant. The 4 parameters are quadword integer constant values in high to low order. For example:

```
// 512-bit integer constant for 10**128
const __VEC_U_512  vec512_ten128th = CONST_VINT512_Q
    (
      CONST_VUINT128_QxW (0x00000000, 0x00000000, 0x0000024e, 0xe91f2603),
      CONST_VUINT128_QxW (0xa6337f19, 0xbccdb0da, 0xc404dc08, 0xd3cff5ec),
      CONST_VUINT128_QxW (0x2374e42f, 0x0f1538fd, 0x03df9909, 0x2e953e01),
      CONST_VUINT128_QxW (0x00000000, 0x00000000, 0x00000000, 0x00000000)
    );
```

7.10.6 Function Documentation

7.10.6.1 vec_add512cu()

```
static __VEC_U_640 vec_add512cu (
            __VEC_U_512 a,
            __VEC_U_512 b )  [inline], [static]
```

Vector Add 512-bit Unsigned Integer & Write Carry.

Compute the 512 bit sum of two 512 bit values a, b and produce the carry. The sum (with-carry) is returned as single 640-bit integer in a homogeneous aggregate structure.

processor	Latency	Throughput
power8	16	1/cycle
power9	12	1/cycle

Parameters

a	vector representation of a unsigned 512-bit integer.
b	vector representation of a unsigned 512-bit integer.

Returns

homogeneous aggregate representation of the unsigned 640-bit sum of a + b.

7.10.6.2 vec_add512ecu()

```
static __VEC_U_640 vec_add512ecu (
            __VEC_U_512 a,
            __VEC_U_512 b,
            vui128_t c )  [inline], [static]
```

Vector Add Extended 512-bit Unsigned Integer & Write Carry.

Compute the 512 bit sum of two 512 bit values a, b and 1 bit value carry-in value c. Produce the carry out of the high order bit of the sum. The sum (with-carry) is returned as single 640-bit integer in a homogeneous aggregate structure.

processor	Latency	Throughput
power8	16	1/cycle
power9	12	1/cycle

Parameters

a	vector representation of a unsigned 512-bit integer.
b	vector representation of a unsigned 512-bit integer.
c	vector representation of a unsigned 1-bit carry.

Returns

homogeneous aggregate representation of the unsigned 640-bit sum of a + b + c.

7.10.6.3 vec_add512eum()

```
static __VEC_U_512 vec_add512eum (
            __VEC_U_512 a,
            __VEC_U_512 b,
            vui128_t c )  [inline], [static]
```

Vector Add Extended 512-bit Unsigned Integer Modulo.

Compute the 512 bit sum of two 512 bit values a, b and 1 bit value carry-in value c. The sum is returned as single 512-bit integer in a homogeneous aggregate structure. Any carry-out of the high order bit of the sum is lost.

processor	Latency	Throughput
power8	16	1/cycle
power9	12	1/cycle

Parameters

a	vector representation of a unsigned 512-bit integer.
b	vector representation of a unsigned 512-bit integer.
c	vector representation of a unsigned 1-bit carry.

Returns

homogeneous aggregate representation of the unsigned 512-bit sum of a + b + c.

7.10.6.4 vec_add512um()

```
static __VEC_U_512 vec_add512um (
                __VEC_U_512 a,
                __VEC_U_512 b )  [inline], [static]
```

Vector Add 512-bit Unsigned Integer Modulo.

Compute the 512 bit sum of two 512 bit values a, b. The sum is returned as single 512-bit integer in a homogeneous aggregate structure. Any carry-out of the high order bit of the sum is lost.

processor	Latency	Throughput
power8	16	1/cycle
power9	12	1/cycle

Parameters

a	vector representation of a unsigned 512-bit integer.
b	vector representation of a unsigned 512-bit integer.

Returns

homogeneous aggregate representation of the unsigned 512-bit sum of a + b.

7.10.6.5 vec_add512ze()

```
static __VEC_U_512 vec_add512ze (
             __VEC_U_512 a,
             vui128_t c ) [inline], [static]
```

Vector Add 512-bit to Zero Extended Unsigned Integer Modulo.

The carry-in is zero extended to the left before computing the 512-bit sum a + c. The sum is returned as single 512-bit integer in a homogeneous aggregate structure. Any carry-out of the high order bit of the sum is lost.

processor	Latency	Throughput
power8	16	1/cycle
power9	12	1/cycle

Parameters

a	vector representation of a unsigned 512-bit integer.
c	vector representation of a unsigned 1-bit carry.

Returns

homogeneous aggregate representation of the unsigned 512-bit sum of a + c.

7.10.6.6 vec_add512ze2()

```
static __VEC_U_512 vec_add512ze2 (
             __VEC_U_512 a,
             vui128_t c1,
             vui128_t c2 ) [inline], [static]
```

Vector Add 512-bit to Zero Extended2 Unsigned Integer Modulo.

The two carry-ins are zero extended to the left before Computing the 512 bit sum a + c1 + c2. The sum is returned as single 512-bit integer in a homogeneous aggregate structure. Any carry-out of the high order bit of the sum is lost.

processor	Latency	Throughput
power8	16	1/cycle
power9	12	1/cycle

Parameters

a	vector representation of a unsigned 512-bit integer.
c1	vector representation of a unsigned 1-bit carry.
c2	vector representation of a unsigned 1-bit carry.

Returns

homogeneous aggregate representation of the unsigned 512-bit sum of a + c1 + c2.

7.10.6.7 vec_madd512x128a128_inline()

```
static __VEC_U_640 vec_madd512x128a128_inline (
            __VEC_U_512 m1,
            vui128_t m2,
            vui128_t a1 )  [inline], [static]
```

Vector 512x128-bit Multiply-Add Unsigned Integer.

Compute the 640 bit sum of 512 bit value m1 and 128-bit value m2 plus 128-bit value a1. The product is returned as single 640-bit integer in a homogeneous aggregate structure.

Note

The advantage of this form is that the final 640 bit sum can not overflow and carries between stages are eliminated. Also applying the addend early (1st multiply stage) reduces the live ranges for registers passing partial products for larger multiple precision multiplies.

We use the COMPILER_FENCE to limit instruction scheduling and code motion to smaller code blocks. This in turn reduces register pressure and avoids generating spill code.

processor	Latency	Throughput
power8	224-232	1/cycle
power9	132-135	1/cycle

Parameters

m1	vector representation of a unsigned 512-bit integer.
m2	vector representation of a unsigned 128-bit integer.
a1	vector representation of a unsigned 128-bit integer.

Returns

homogeneous aggregate representation of the unsigned 640-bit sum of (m1 ∗ m2) + c.

7.10.6.8 vec_madd512x128a128a512_inline()

```
static __VEC_U_640 vec_madd512x128a128a512_inline (
             __VEC_U_512 m1,
             vui128_t m2,
             vui128_t a1,
             __VEC_U_512 a2 )  [inline], [static]
```

Vector 512x128-bit Multiply-Add Unsigned Integer.

Compute the 640 bit sum of 512 bit value m1 and 128-bit value m2, plus 128-bit value a1, plus 512-bit value a2. The sum is returned as single 640-bit integer in a homogeneous aggregate structure.

Note

> The advantage of this form is that the final 640 bit sum can not overflow and carries between stages are eliminated. Also applying the addend early (1st multiply stage) reduces the live ranges for registers passing partial products for larger multiple precision multiplies.
>
> We use the COMPILER_FENCE to limit instruction scheduling and code motion to smaller code blocks. This in turn reduces register pressure and avoids generating spill code.

processor	Latency	Throughput
power8	224-232	1/cycle
power9	132-135	1/cycle

Parameters

m1	vector representation of a unsigned 512-bit integer.
m2	vector representation of a unsigned 128-bit integer.
a1	vector representation of a unsigned 128-bit integer.
a2	vector representation of a unsigned 512-bit integer.

Returns

> homogeneous aggregate representation of the unsigned 640-bit sum of (m1 ∗ m2) + a1 + a2.

7.10.6.9 vec_madd512x128a512()

```
__VEC_U_640 vec_madd512x128a512 (
             __VEC_U_512 m1,
             vui128_t m2,
             __VEC_U_512 a2 )
```

Vector 512x128-bit Multiply-Add Unsigned Integer.

Compute the 640 bit sum of the product of the 512 bit value m1 and 128-bit value m2 plus the 512-bit value a2. The sum is returned as single 640-bit integer in a homogeneous aggregate structure.

Note

> The advantage of this form is that the final 640 bit sum can not overflow and carries between stages are eliminated. Also applying the addend early (1st multiply stage) reduces the live ranges for registers passing partial products for larger multiple precision multiplies.

processor	Latency	Throughput
power8	224-232	1/cycle
power9	132-135	1/cycle

Parameters

m1	vector representation of a unsigned 512-bit integer.
m2	vector representation of a unsigned 128-bit integer.
a2	vector representation of a unsigned 512-bit integer.

Returns

> homogeneous aggregate representation of the unsigned 640-bit sum of (m1 ∗ m2) + a2.

7.10.6.10 vec_madd512x128a512_inline()

```
static __VEC_U_640 vec_madd512x128a512_inline (
              __VEC_U_512 m1,
              vui128_t m2,
              __VEC_U_512 a2 ) [inline], [static]
```

Vector 512x128-bit Multiply-Add Unsigned Integer.

Compute the 640 bit sum of 512 bit value m1 and 128-bit value m2 plus 512-bit value a2. The sum is returned as single 640-bit integer in a homogeneous aggregate structure.

Note

> The advantage of this form is that the final 640 bit sum can not overflow and carries between stages are eliminated. Also applying the addend early (1st multiply stage) reduces the live ranges for registers passing partial products for larger multiple precision multiplies.
>
> We use the COMPILER_FENCE to limit instruction scheduling and code motion to smaller code blocks. This in turn reduces register pressure and avoids generating spill code.

processor	Latency	Throughput
power8	224-232	1/cycle
power9	132-135	1/cycle

Parameters

m1	vector representation of a unsigned 512-bit integer.
m2	vector representation of a unsigned 128-bit integer.
a2	vector representation of a unsigned 512-bit integer.

Returns

homogeneous aggregate representation of the unsigned 640-bit sum of (m1 $*$ m2) + a2.

7.10.6.11 vec_madd512x512a512_inline()

```
static __VEC_U_1024 vec_madd512x512a512_inline (
            __VEC_U_512 m1,
            __VEC_U_512 m2,
            __VEC_U_512 a1 )  [inline], [static]
```

Vector 512-bit Unsigned Integer Multiply-Add.

Compute the 1024 bit sum of the product of 512 bit values m1 and m2 and 512 bit addend a1. The sum is returned as single 1024-bit integer in a homogeneous aggregate structure.

Note

The advantage of this form is that the final 1024 bit sum can not overflow and carries between stages are eliminated. Also applying the addend early (1st multiply stage) reduces the live ranges for registers passing partial products for larger multiple precision multiplies.

We use the COMPILER_FENCE to limit instruction scheduling and code motion to smaller code blocks. This in turn reduces register pressure and avoids generating spill code.

processor	Latency	Throughput
power8	∼600	1/cycle
power9	∼210	1/cycle

Parameters

m1	vector representation of a unsigned 512-bit integer.
m2	vector representation of a unsigned 512-bit integer.
a1	vector representation of a unsigned 512-bit integer.

Returns

homogeneous aggregate representation of the unsigned 1028-bit product of a ∗ b.

7.10.6.12 vec_mul1024x1024()

```
void vec_mul1024x1024 (
            __VEC_U_2048 * p2048,
            __VEC_U_1024 * m1,
            __VEC_U_1024 * m2 )
```

Vector 1024x1024-bit Unsigned Integer Multiply.

Compute the 2048 bit product of 1024 bit values m1 and m2. The product is returned as single 2048-bit integer in a homogeneous aggregate structure.

Note

This is the dynamic call ABI for IFUNC selection. The static implementations are vec_mul1024x1024_PWR8 and vec_mul1024x1024_PWR9. For static calls the __VEC_PWR_IMP() macro will add appropriate suffix based on the compile -mcpu= option.

The storage order for quadwords matches the system endian. On Little Endian systems the least significant quadword is quadword element 0. The most significant is quadword elements [M-1], [N-1], and [M+N-1]. On Big Endian systems the least significant quadword is quadword elements [M-1], [N-1], and [M+N-1]. The most significant is quadword element 0.

processor	Latency	Throughput
power8	∼2500	1/cycle
power9	∼810	1/cycle

Parameters

p2048	vector result as a unsigned 2048-bit integer in storage.
m1	vector representation of a unsigned 1024-bit integer.
m2	vector representation of a unsigned 1024-bit integer.

7.10.6.13 vec_mul128_byMN()

```
void vec_mul128_byMN (
            vui128_t * p,
            vui128_t * m1,
            vui128_t * m2,
            unsigned long M,
            unsigned long N )
```

Vector Unsigned Integer Quadword MxN Multiply.

Compute the M+N quadword product of two quadword arrays m1, m2. The product is returned as M+N quadword array p.

Note

> This is the dynamic call ABI for IFUNC selection. The static implementations are vec_mul128_byMN_PWR8 and vec_mul128_byMN_PWR9. For static calls the __VEC_PWR_IMP() macro will add appropriate suffix based on the compile -mcpu= option.
>
> The storage order for quadwords matches the system endian. On Little Endian systems the least significant quadword is quadword element 0. The most significant is quadword elements [M-1], [N-1], and [M+N-1]. On Big Endian systems the least significant quadword is quadword elements [M-1], [N-1], and [M+N-1]. The most significant is quadword element 0.

processor	Latency	Throughput
power8	???	1/cycle
power9	???	1/cycle

Parameters

p	pointer to vector result as a unsigned (M+N)x128-bit integer in storage.
m1	pointer to vector representation of a unsigned Mx128-bit integer.
m2	pointer ro vector representation of a unsigned Nx128-bit integer.
M	long int specifying the number of quadword in m1.
N	long int specifying the number of quadword in m2.

7.10.6.14 vec_mul128x128()

```
__VEC_U_256 vec_mul128x128 (
            vui128_t m1,
            vui128_t m2 )
```

Vector 128x128bit Unsigned Integer Multiply.

Compute the 256 bit product of two 128 bit values a, b. The product is returned as single 256-bit integer in a homogeneous aggregate structure.

Note

> This is the dynamic call ABI for IFUNC selection. The static implementations are vec_mul128x128_PWR8 and vec_mul128x128_PWR9. For static calls the __VEC_PWR_IMP() macro will add appropriate suffix based on the compile -mcpu= option.

processor	Latency	Throughput
power8	48-56	1/cycle
power9	16-24	1/cycle

Parameters

m1	vector representation of a unsigned 128-bit integer.
m2	vector representation of a unsigned 128-bit integer.

Returns

homogeneous aggregate representation of the unsigned 256-bit product of a ∗ b.

7.10.6.15 vec_mul128x128_inline()

```
static __VEC_U_256 vec_mul128x128_inline (
             vui128_t a,
             vui128_t b )  [inline], [static]
```

Vector 128x128bit Unsigned Integer Multiply.

Compute the 256 bit product of two 128 bit values a, b. The product is returned as single 256-bit integer in a homogeneous aggregate structure.

processor	Latency	Throughput
power8	56-64	1/cycle
power9	33-39	1/cycle

Parameters

a	vector representation of a unsigned 128-bit integer.
b	vector representation of a unsigned 128-bit integer.

Returns

homogeneous aggregate representation of the unsigned 256-bit product of a ∗ b.

7.10.6.16 vec_mul2048x2048()

```
void vec_mul2048x2048 (
             __VEC_U_4096 * p4096,
             __VEC_U_2048 * m1,
             __VEC_U_2048 * m2 )
```

Vector 2048x2048-bit Unsigned Integer Multiply.

Compute the 4096 bit product of 2048 bit values m1 and m2. The product is returned as single 4096-bit integer in a homogeneous aggregate structure.

Note

> This is the dynamic call ABI for IFUNC selection. The static implementations are vec_mul2048x2048_PWR8 and vec_mul2048x2048_PWR9. For static calls the __VEC_PWR_IMP() macro will add appropriate suffix based on the compile -mcpu= option.
>
> The storage order for quadwords matches the system endian. On Little Endian systems the least significant quadword is quadword element 0. The most significant is quadword elements [M-1], [N-1], and [M+N-1]. On Big Endian systems the least significant quadword is quadword elements [M-1], [N-1], and [M+N-1]. The most significant is quadword element 0.

processor	Latency	Throughput
power8	∼12000	1/cycle
power9	4770	1/cycle

Parameters

p4096	vector result as a unsigned 4096-bit integer in storage.
m1	vector representation of a unsigned 2048-bit integer.
m2	vector representation of a unsigned 2048-bit integer.

7.10.6.17 vec_mul256x256()

```
__VEC_U_512 vec_mul256x256 (
             __VEC_U_256 m1,
             __VEC_U_256 m2 )
```

Vector 256x256-bit Unsigned Integer Multiply.

Compute the 512 bit product of two 256 bit values a, b. The product is returned as single 512-bit integer in a homogeneous aggregate structure.

Note

> This is the dynamic call ABI for IFUNC selection. The static implementations are vec_mul256x256_PWR8 and vec_mul256x256_PWR9. For static calls the __VEC_PWR_IMP() macro will add appropriate suffix based on the compile -mcpu= option.

processor	Latency	Throughput
power8	140-150	1/cycle
power9	46-58	1/cycle

Parameters

m1	vector representation of a unsigned 256-bit integer.
m2	vector representation of a unsigned 256-bit integer.

Returns

homogeneous aggregate representation of the unsigned 512-bit product of m1 ∗ m2.

7.10.6.18 vec_mul256x256_inline()

```
static __VEC_U_512 vec_mul256x256_inline (
            __VEC_U_256 m1,
            __VEC_U_256 m2 )  [inline], [static]
```

Vector 256x256-bit Unsigned Integer Multiply.

Compute the 512 bit product of two 256 bit values a, b. The product is returned as single 512-bit integer in a homogeneous aggregate structure.

Note

Using the Multiply-Add form which applies the addend early reduces the live ranges for registers passing partial products for larger multiple precision multiplies.

We use the COMPILER_FENCE to limit instruction scheduling and code motion to smaller code blocks. This in turn reduces register pressure and avoids generating spill code.

processor	Latency	Throughput
power8	224-232	1/cycle
power9	132-135	1/cycle

Parameters

m1	vector representation of a unsigned 256-bit integer.
m2	vector representation of a unsigned 256-bit integer.

Returns

homogeneous aggregate representation of the unsigned 512-bit product of m1 ∗ m2.

7.10.6.19 vec_mul512_byMN()

```
void vec_mul512_byMN (
             __VEC_U_512 * p,
             __VEC_U_512 * m1,
             __VEC_U_512 * m2,
             unsigned long M,
             unsigned long N )
```

Vector Unsigned Integer Quadword 4xMxN Multiply.

Compute the 4xM+N quadword product of two quadword arrays m1, m2. The product is returned as 4xM+N quadword array p.

Note

This is the dynamic call ABI for IFUNC selection. The static implementations are vec_mul512_byMN_PWR8 and vec_mul512_byMN_PWR9. For static calls the __VEC_PWR_IMP() macro will add appropriate suffix based on the compile -mcpu= option.

The storage order for quadwords matches the system endian. On Little Endian systems the least significant quadword is quadword element 0. The most significant is quadword elements [M-1], [N-1], and [M+N-1]. On Big Endian systems the least significant quadword is quadword elements [M-1], [N-1], and [M+N-1]. The most significant is quadword element 0.

processor	Latency	Throughput
power8	∼570∗(M∗N)	1/cycle
power9	∼260∗(M∗N)	1/cycle

Parameters

p	pointer to vector result as a unsigned (M+N)x512-bit integer in storage.
m1	pointer to vector representation of a unsigned Mx512-bit integer.
m2	pointer ro vector representation of a unsigned Nx512-bit integer.
M	long int specifying the number of 4x quadwords in m1.
N	long int specifying the number of 4x quadwords in m2.

7.10.6.20 vec_mul512x128()

```
__VEC_U_640 vec_mul512x128 (
                __VEC_U_512 m1,
                vui128_t m2 )
```

Vector 512x128-bit Unsigned Integer Multiply.

Compute the 640 bit product of 512 bit value m1 and 128-bit value m2. The product is returned as single 640-bit integer in a homogeneous aggregate structure.

Note

> This is the dynamic call ABI for IFUNC selection. The static implementations are vec_mul256x256_PWR8 and vec_mul256x256_PWR9. For static calls the __VEC_PWR_IMP() macro will add appropriate suffix based on the compile -mcpu= option.

processor	Latency	Throughput
power8	224-232	1/cycle
power9	132-135	1/cycle

Parameters

m1	vector representation of a unsigned 512-bit integer.
m2	vector representation of a unsigned 128-bit integer.

Returns

> homogeneous aggregate representation of the unsigned 640-bit product of m1 ∗ m2.

7.10.6.21 vec_mul512x128_inline()

```
static __VEC_U_640 vec_mul512x128_inline (
                __VEC_U_512 m1,
                vui128_t m2 )  [inline], [static]
```

Vector 512x128-bit Unsigned Integer Multiply.

Compute the 640 bit product of 512 bit value m1 and 128-bit value m2. The product is returned as single 640-bit integer in a homogeneous aggregate structure.

Note

> Using the Multiply-Add form which applies the addend early reduces the live ranges for registers passing partial products for larger multiple precision multiplies.
>
> We use the COMPILER_FENCE to limit instruction scheduling and code motion to smaller code blocks. This in turn reduces register pressure and avoids generating spill code.

processor	Latency	Throughput
power8	224-232	1/cycle
power9	132-135	1/cycle

Parameters

m1	vector representation of a unsigned 512-bit integer.
m2	vector representation of a unsigned 128-bit integer.

Returns

homogeneous aggregate representation of the unsigned 640-bit product of m1 ∗ m2.

7.10.6.22 vec_mul512x512()

```
__VEC_U_1024 vec_mul512x512 (
            __VEC_U_512 m1,
            __VEC_U_512 m2 )
```

Vector 512x512-bit Unsigned Integer Multiply.

Compute the 1024 bit product of 512 bit values m1 and m2. The product is returned as single 1024-bit integer in a homogeneous aggregate structure.

Note

This is the dynamic call ABI for IFUNC selection. The static implementations are vec_mul512x512_PWR8 and vec_mul512x512_PWR9. For static calls the __VEC_PWR_IMP() macro will add appropriate suffix based on the compile -mcpu= option.

processor	Latency	Throughput
power8	∼600	1/cycle
power9	∼210	1/cycle

Parameters

m1	vector representation of a unsigned 512-bit integer.
m2	vector representation of a unsigned 512-bit integer.

Returns

homogeneous aggregate representation of the unsigned 1028-bit product of a ∗ b.

7.10.6.23 vec_mul512x512_inline()

```
static __VEC_U_1024 vec_mul512x512_inline (
            __VEC_U_512 m1,
            __VEC_U_512 m2 ) [inline], [static]
```

Vector 512x512-bit Unsigned Integer Multiply.

Compute the 1024 bit product of 512 bit values m1 and m2. The product is returned as single 1024-bit integer in a homogeneous aggregate structure.

Note

> We use the COMPILER_FENCE to limit instruction scheduling and code motion to smaller code blocks. This in turn reduces register pressure and avoids generating spill code.
>
> Using the Multiply-Add form which applies the addend early reduces the live ranges for registers passing partial products for larger multiple precision multiplies.

processor	Latency	Throughput
power8	∼600	1/cycle
power9	∼210	1/cycle

Parameters

m1	vector representation of a unsigned 512-bit integer.
m2	vector representation of a unsigned 512-bit integer.

Returns

> homogeneous aggregate representation of the unsigned 1028-bit product of m1 ∗ m2.

7.11 src/pveclib/vec_int64_ppc.h File Reference

Header package containing a collection of 128-bit SIMD operations over 64-bit integer elements.

```
#include <pveclib/vec_int32_ppc.h>
```

Functions

- static vui64_t vec_absdud (vui64_t vra, vui64_t vrb)

Vector Absolute Difference Unsigned Doubleword.

- static vui64_t vec_addudm (vui64_t a, vui64_t b)

  Vector Add Unsigned Doubleword Modulo.
- static vui64_t vec_clzd (vui64_t vra)

  Vector Count Leading Zeros Doubleword for unsigned long long elements.
- static vui64_t vec_ctzd (vui64_t vra)

  Vector Count Trailing Zeros Doubleword for unsigned long long elements.
- static vb64_t vec_cmpeqsd (vi64_t a, vi64_t b)

  Vector Compare Equal Signed Doubleword.
- static vb64_t vec_cmpequd (vui64_t a, vui64_t b)

  Vector Compare Equal Unsigned Doubleword.
- static vb64_t vec_cmpgesd (vi64_t a, vi64_t b)

  Vector Compare Greater Than or Equal Signed Doubleword.
- static vb64_t vec_cmpgeud (vui64_t a, vui64_t b)

  Vector Compare Greater Than or Equal Unsigned Doubleword.
- static vb64_t vec_cmpgtsd (vi64_t a, vi64_t b)

  Vector Compare Greater Than Signed Doubleword.
- static vb64_t vec_cmpgtud (vui64_t a, vui64_t b)

  Vector Compare Greater Than Unsigned Doubleword.
- static vb64_t vec_cmplesd (vi64_t a, vi64_t b)

  Vector Compare Less Than Equal Signed Doubleword.
- static vb64_t vec_cmpleud (vui64_t a, vui64_t b)

  Vector Compare Less Than Equal Unsigned Doubleword.
- static vb64_t vec_cmpltsd (vi64_t a, vi64_t b)

  Vector Compare less Than Signed Doubleword.
- static vb64_t vec_cmpltud (vui64_t a, vui64_t b)

  Vector Compare less Than Unsigned Doubleword.
- static vb64_t vec_cmpnesd (vi64_t a, vi64_t b)

  Vector Compare Not Equal Signed Doubleword.
- static vb64_t vec_cmpneud (vui64_t a, vui64_t b)

  Vector Compare Not Equal Unsigned Doubleword.
- static int vec_cmpsd_all_eq (vi64_t a, vi64_t b)

  Vector Compare all Equal Signed Doubleword.
- static int vec_cmpsd_all_ge (vi64_t a, vi64_t b)

  Vector Compare all Greater Than or Equal Signed Doubleword.
- static int vec_cmpsd_all_gt (vi64_t a, vi64_t b)

  Vector Compare all Greater Than Signed Doubleword.
- static int vec_cmpsd_all_le (vi64_t a, vi64_t b)

  Vector Compare all Less than equal Signed Doubleword.
- static int vec_cmpsd_all_lt (vi64_t a, vi64_t b)

  Vector Compare all Less than Signed Doubleword.
- static int vec_cmpsd_all_ne (vi64_t a, vi64_t b)

  Vector Compare all Not Equal Signed Doubleword.
- static int vec_cmpsd_any_eq (vi64_t a, vi64_t b)

  Vector Compare any Equal Signed Doubleword.
- static int vec_cmpsd_any_ge (vi64_t a, vi64_t b)

  Vector Compare any Greater Than or Equal Signed Doubleword.

- static int vec_cmpsd_any_gt (vi64_t a, vi64_t b)

  Vector Compare any Greater Than Signed Doubleword.
- static int vec_cmpsd_any_le (vi64_t a, vi64_t b)

  Vector Compare any Less than equal Signed Doubleword.
- static int vec_cmpsd_any_lt (vi64_t a, vi64_t b)

  Vector Compare any Less than Signed Doubleword.
- static int vec_cmpsd_any_ne (vi64_t a, vi64_t b)

  Vector Compare any Not Equal Signed Doubleword.
- static int vec_cmpud_all_eq (vui64_t a, vui64_t b)

  Vector Compare all Equal Unsigned Doubleword.
- static int vec_cmpud_all_ge (vui64_t a, vui64_t b)

  Vector Compare all Greater Than or Equal Unsigned Doubleword.
- static int vec_cmpud_all_gt (vui64_t a, vui64_t b)

  Vector Compare all Greater Than Unsigned Doubleword.
- static int vec_cmpud_all_le (vui64_t a, vui64_t b)

  Vector Compare all Less than equal Unsigned Doubleword.
- static int vec_cmpud_all_lt (vui64_t a, vui64_t b)

  Vector Compare all Less than Unsigned Doubleword.
- static int vec_cmpud_all_ne (vui64_t a, vui64_t b)

  Vector Compare all Not Equal Unsigned Doubleword.
- static int vec_cmpud_any_eq (vui64_t a, vui64_t b)

  Vector Compare any Equal Unsigned Doubleword.
- static int vec_cmpud_any_ge (vui64_t a, vui64_t b)

  Vector Compare any Greater Than or Equal Unsigned Doubleword.
- static int vec_cmpud_any_gt (vui64_t a, vui64_t b)

  Vector Compare any Greater Than Unsigned Doubleword.
- static int vec_cmpud_any_le (vui64_t a, vui64_t b)

  Vector Compare any Less than equal Unsigned Doubleword.
- static int vec_cmpud_any_lt (vui64_t a, vui64_t b)

  Vector Compare any Less than Unsigned Doubleword.
- static int vec_cmpud_any_ne (vui64_t a, vui64_t b)

  Vector Compare any Not Equal Unsigned Doubleword.
- static vi64_t vec_maxsd (vi64_t vra, vi64_t vrb)

  Vector Maximum Signed Doubleword.
- static vui64_t vec_maxud (vui64_t vra, vui64_t vrb)

  Vector Maximum Unsigned Doubleword.
- static vi64_t vec_minsd (vi64_t vra, vi64_t vrb)

  Vector Minimum Signed Doubleword.
- static vui64_t vec_minud (vui64_t vra, vui64_t vrb)

  Vector Minimum Unsigned Doubleword.
- static vui64_t vec_mrgahd (vui128_t vra, vui128_t vrb)

  Vector Merge Algebraic High Doublewords.
- static vui64_t vec_mrgald (vui128_t vra, vui128_t vrb)

  Vector Merge Algebraic Low Doublewords.
- static vui64_t vec_mrged (vui64_t __VA, vui64_t __VB)

  Vector Merge Even Doubleword. Merge the even doubleword elements from two vectors into the high and low doubleword elements of the result. This is effectively the VSX Permute Doubleword Immediate operation modified for endian.

- static vui64_t vec_mrghd (vui64_t __VA, vui64_t __VB)

  Vector Merge High Doubleword. Merge the high doubleword elements from two vectors into the high and low doubleword elements of the result. This is effectively the VSX Permute Doubleword Immediate operation modified for endian.
- static vui64_t vec_mrgld (vui64_t __VA, vui64_t __VB)

  Vector Merge Low Doubleword. Merge the low doubleword elements from two vectors into the high and low doubleword elements of the result. This is effectively the VSX Permute Doubleword Immediate operation modified for endian.
- static vui64_t vec_mrgod (vui64_t __VA, vui64_t __VB)

  Vector Merge Odd Doubleword. Merge the odd doubleword elements from two vectors into the high and low doubleword elements of the result. This is effectively the VSX Permute Doubleword Immediate operation modified for endian.
- static vui128_t vec_msumudm (vui64_t a, vui64_t b, vui128_t c)

  Vector Multiply-Sum Unsigned Doubleword Modulo.
- static vui128_t vec_muleud (vui64_t a, vui64_t b)

  Vector Multiply Even Unsigned Doublewords.
- static vui64_t vec_mulhud (vui64_t vra, vui64_t vrb)

  Vector Multiply High Unsigned Doubleword.
- static vui128_t vec_muloud (vui64_t a, vui64_t b)

  Vector Multiply Odd Unsigned Doublewords.
- static vui64_t vec_muludm (vui64_t vra, vui64_t vrb)

  Vector Multiply Unsigned Doubleword Modulo.
- static vui64_t vec_pasted (vui64_t __VH, vui64_t __VL)

  Vector doubleword paste. Concatenate the high doubleword of the 1st vector with the low double word of the 2nd vector.
- static vui64_t vec_permdi (vui64_t vra, vui64_t vrb, const int ctl)

  Vector Permute Doubleword Immediate. Combine a doubleword selected from the 1st (vra) vector with a doubleword selected from the 2nd (vrb) vector.
- static vui64_t vec_popcntd (vui64_t vra)

  Vector Population Count doubleword.
- static vui64_t vec_revbd (vui64_t vra)

  byte reverse each doubleword for a vector unsigned long int.
- static vui64_t vec_vrld (vui64_t vra, vui64_t vrb)

  Vector Rotate Left Doubleword.
- static vui64_t vec_vsld (vui64_t vra, vui64_t vrb)

  Vector Shift Left Doubleword.
- static vui64_t vec_vsrd (vui64_t vra, vui64_t vrb)

  Vector Shift Right Doubleword.
- static vi64_t vec_vsrad (vi64_t vra, vui64_t vrb)

  Vector Shift Right Algebraic Doubleword.
- static vb64_t vec_setb_sd (vi64_t vra)

  Vector Set Bool from Signed Doubleword.
- static vui64_t vec_rldi (vui64_t vra, const unsigned int shb)

  Vector Rotate left Doubleword Immediate.
- static vui64_t vec_sldi (vui64_t vra, const unsigned int shb)

  Vector Shift left Doubleword Immediate.
- static vi64_t vec_selsd (vi64_t vra, vi64_t vrb, vb64_t vrc)

  Vector Select Signed Doubleword.
- static vui64_t vec_selud (vui64_t vra, vui64_t vrb, vb64_t vrc)

  Vector Select Unsigned Doubleword.
- static vui64_t vec_splatd (vui64_t vra, const int ctl)

Vector splat doubleword. Duplicate the selected doubleword element across the doubleword elements of the result. This is effectively the VSX Merge doubleword operation modified for endian.

- static vi64_t vec_splat_s64 (const int sim)

  Vector Splat Immediate Signed Doubleword. Duplicate the signed integer constant across doubleword elements of the result. This is the doubleword equivalent Vector Splat Immediate Signed (Byte | Halfword |Word).
- static vui64_t vec_splat_u64 (const int sim)

  Vector Splat Immediate Unsigned Doubleword. Duplicate the unsigned integer constant across doubleword elements of the result. This is the doubleword equivalent Vector Splat Immediate Unsigned (Byte | Halfword |Word).
- static vui64_t vec_spltd (vui64_t vra, const int ctl)
- static vui64_t vec_srdi (vui64_t vra, const unsigned int shb)

  Vector Shift Right Doubleword Immediate.
- static vi64_t vec_sradi (vi64_t vra, const unsigned int shb)

  Vector Shift Right Algebraic Doubleword Immediate.
- static vui64_t vec_subudm (vui64_t a, vui64_t b)

  Vector Subtract Unsigned Doubleword Modulo.
- static vui64_t vec_swapd (vui64_t vra)

  Vector doubleword swap. Exchange the high and low doubleword elements of a vector.
- static vui64_t vec_vgluddo (unsigned long long ∗array, vi64_t vra)

  Vector Gather-Load Integer Doublewords from Vector Doubleword Offsets.
- static vui64_t vec_vgluddsx (unsigned long long ∗array, vi64_t vra, const unsigned char scale)

  Vector Gather-Load Integer Doublewords from Vector Doubleword Scaled Indexes.
- static vui64_t vec_vgluddx (unsigned long long ∗array, vi64_t vra)

  Vector Gather-Load Integer Doublewords from Vector Doubleword Indexes.
- static vui64_t vec_vgludso (unsigned long long ∗array, const long long offset0, const long long offset1)

  Vector Gather-Load Integer Doublewords from Scalar Offsets.
- static vui64_t vec_vlsidx (const signed long long ra, const unsigned long long ∗rb)

  Vector Load Scalar Integer Doubleword Indexed.
- static vui128_t vec_vmadd2eud (vui64_t a, vui64_t b, vui64_t c, vui64_t d)

  Vector Multiply-Add2 Even Unsigned Doublewords.
- static vui128_t vec_vmaddeud (vui64_t a, vui64_t b, vui64_t c)

  Vector Multiply-Add Even Unsigned Doublewords.
- static vui128_t vec_vmadd2oud (vui64_t a, vui64_t b, vui64_t c, vui64_t d)

  Vector Multiply-Add2 Odd Unsigned Doublewords.
- static vui128_t vec_vmaddoud (vui64_t a, vui64_t b, vui64_t c)

  Vector Multiply-Add Odd Unsigned Doublewords.
- static vui128_t vec_vmuleud (vui64_t a, vui64_t b)

  Vector Multiply Even Unsigned Doublewords.
- static vui128_t vec_vmuloud (vui64_t a, vui64_t b)

  Vector Multiply Odd Unsigned Doublewords.
- static vui128_t vec_vmsumeud (vui64_t a, vui64_t b, vui128_t c)

  Vector Multiply-Sum Even Unsigned Doublewords.
- static vui128_t vec_vmsumoud (vui64_t a, vui64_t b, vui128_t c)

  Vector Multiply-Sum Odd Unsigned Doublewords.
- static vui32_t vec_vpkudum (vui64_t vra, vui64_t vrb)

  Vector Pack Unsigned Doubleword Unsigned Modulo.
- static void vec_vsstuddo (vui64_t xs, unsigned long long ∗array, vi64_t vra)

  Vector Scatter-Store Integer Doublewords to Vector Doublewords Offsets.
- static void vec_vsstuddsx (vui64_t xs, unsigned long long ∗array, vi64_t vra, const unsigned char scale)

Vector Scatter-Store Integer Doublewords to Vector Doubleword Scaled Indexes.

- static void vec_vsstuddx (vui64_t xs, unsigned long long ∗array, vi64_t vra)

  Vector Scatter-Store Integer Doublewords to Vector Doubleword Indexes.
- static void vec_vsstudso (vui64_t xs, unsigned long long ∗array, const long long offset0, const long long offset1)

  Vector Scatter-Store Integer Doublewords to Scalar Offsets.
- static void vec_vstsidx (vui64_t xs, const signed long long ra, unsigned long long ∗rb)

  Vector Store Scalar Integer Doubleword Indexed.
- static vui64_t vec_xxspltd (vui64_t vra, const int ctl)

  Vector splat doubleword. Duplicate the selected doubleword element across the doubleword elements of the result.
- static vui64_t vec_vmaddeuw (vui32_t a, vui32_t b, vui32_t c)

  Vector Multiply-Add Even Unsigned Words.
- static vui64_t vec_vmadd2euw (vui32_t a, vui32_t b, vui32_t c, vui32_t d)

  Vector Multiply-Add2 Even Unsigned Words.
- static vui64_t vec_vmaddouw (vui32_t a, vui32_t b, vui32_t c)

  Vector Multiply-Add Odd Unsigned Words.
- static vui64_t vec_vmadd2ouw (vui32_t a, vui32_t b, vui32_t c, vui32_t d)

  Vector Multiply-Add2 Odd Unsigned Words.
- static vui64_t vec_vmsumuwm (vui32_t vra, vui32_t vrb, vui64_t vrc)

  Vector Multiply-Sum Unsigned Word Modulo.

7.11.1 Detailed Description

Header package containing a collection of 128-bit SIMD operations over 64-bit integer elements.

Most of these operations are implemented in a single instruction on newer (POWER8/POWER9) processors. This header serves to fill in functional gaps for older (POWER7, POWER8) processors and provides a in-line assembler implementation for older compilers that do not provide the built-ins.

The original VMX (AKA Altivec) did not define any doubleword element (long long integer or double float) operations. The VSX facility (introduced with POWER7) added vector double float but did not add any integer doubleword (64-bit) operations. However it did add a useful doubleword permute immediate and word wise; merge, shift, and splat immediate operations. Otherwise vector long int (64-bit elements) operations have to be implemented using VMX word and halfword element integer operations for POWER7.

POWER8 (PowerISA 2.07B) adds important doubleword integer (add, subtract, compare, shift, rotate, ...) VMX operations. POWER8 also added multiply word operations that produce the full doubleword product and full quadword add / subtract (with carry extend).

POWER9 (PowerISA 3.0B) adds the **Vector Multiply-Sum Unsigned Doubleword Modulo** instruction. This is not the expected multiply even/odd/modulo doubleword nor a full multiply modulo quadword. But with a few extra (permutes and splat zero) instructions you can get equivalent function.

Note

> The doubleword integer multiply implementations are included in vec_int128_ppc.h. This resolves a circular dependency as 64-bit by 64-bit integer multiplies require 128-bit integer addition (vec_adduqm()) to produce the full product.

See also

vec_msumudm, vec_muleud, vec_mulhud, vec_muloud, vec_muludm, vec_vmuleud, and vec_vmuloud

Most of these intrinsic (compiler built-in) operations are defined in <altivec.h> and described in the compiler documentation. However it took several compiler releases for all the new POWER8 64-bit integer vector intrinsics to be added to **altivec.h**. This support started with the GCC 4.9 but was not complete across function/type and bug free until GCC 6.0.

Note

The compiler disables associated <altivec.h> built-ins if the **mcpu** target does not enable the specific instruction. For example, if you compile with **-mcpu=power7**, vec_vclz and vec_vclzd will not be defined. But vec_clzd is always defined in this header, will generate the minimum code, appropriate for the target, and produce correct results.

64-bit integer operations are commonly used in the implementation of optimized double float math library functions and this applies to the vector equivalents of math functions. So missing, incomplete or buggy support for vector long integer intrinsics can be a impediment to the implementation of optimized and portable vector double math libraries. This header is a prerequisite for vec_f64_ppc.h which together are intended to support the implementation of vector math libraries.

Most of these operations are implemented in a single instruction on newer (POWER8/POWER9) processors. So this header serves to fill in functional gaps for older (POWER7, POWER8) processors and provides a in-line assembler implementation for older compilers that do not provide the built-ins.

This header covers operations that are any of the following:

- Implemented in hardware instructions for later processors and useful to programmers, on slightly older processors, even if the equivalent function requires more instructions. Examples include the doubleword operations: Add, Compare, Maximum, Minimum and Subtract.
- Defined in the OpenPOWER ABI but *not* yet defined in <altivec.n> provided by available compilers in common use. Examples include doubleword forms of: Multiply Even/Odd/Modulo, Count Leading Zeros, Population Count, and Byte Reverse operations.
- Commonly used operations, not covered by the ABI or <altivec.h>, and require multiple instructions or are not obvious. Examples include doubleword forms of: Merge Algebraic High/Low, Paste, and Rotate/Shift Immediate operations.
- Commonly used operations that are useful for doubleword, but are missing from the PowerISA and OpenPOWER ABI. Examples include: Absolute Difference Doubleword and Multiply-Sum Unsigned Word Modulo.

7.11.2 Some missing doubleword operations

The original VMX instruction set extension was limited to byte, halfword, and word size element operations. This limited vector arithmetic operations to char, short, int and float elements. This limitation persisted until PowerISA 2.06 (POW←ER7) added the Vector Scalar Extensions (VSX) facility. VSX combined/extended the FPRs and VRs into 64 by 128-bit Vector/Scalar Registers (VSRs).

VSX added a large number of scalar double-precision and vector single / double-precision floating-point operations. The double-precision scalar (**xs** prefix) instructions where largely duplicates of the existing Floating-Point Facility operations, extended to access the whole (64) VSX register set. Similarly the VSX vector single precision floating-point (**xv** prefix, **sp** suffix) instructions were added to give vectorized float code access to 64 VSX registers.

The addition of VSX vector double-precision (**xv** prefix) instructions was the most significant addition. This added vector doubleword floating-point operations and provided access to all 64 VSX registers. Alas, there are no doubleword (64-bit long) integer operations in the initial VSX. A few logical and permute class (**xx** prefix) operations on word/doubleword elements where tacked on. These apply equally to float and integer elements. But nothing for 64-bit integer arithmetic.

Note

The full title in PowerISA 2.06 is **Vector-Scalar Floating-Point Operations [Category: VSX]**.

PowerISA 2.07 (POWER8) did add a significant number of doubleword (64-bit) integer operations. Including;

- Add and subtract modulo
- Signed and unsigned compare, maximum, minimum,
- Shift and rotate
- Count leading zeros and population count

Also a number of new word (32-bit) integer operations;

- Multiply even/odd/modulo.
- Pack signed/unsigned/saturate and Unpack signed.
- Merge even/odd words

And some new quadword (128-bit) integer operations;

- Add and Subtract modulo/extend/write-carry
- Decimal Add and Subtract modulo

And some specialized operations;

- Crypto, Raid, Polynomial multiply-sum

Note

The operations above are all Vector Category and can only access the 32 original vector registers (VSRs 32-63).

The new VSX operations (with access to all 64 VSRs) were not directly applicable to 64-bit integer arithmetic:

- Scalar single precision floating-point
- Direct move between GPRs and VSRs
- Logical operations; equivalence, not and, or compliment

PowerISA 3.0 (POWER9) adds a few more doubleword (64-bit) integer operations. Including;

- Compare not equal
- Count trailing zeros and parity
- Extract and Insert

- Multiply-sum modulo
- Negate
- Rotate Left under mask

Also a number of new word (32-bit) integer operations;

- Absolute Difference word
- Extend Sign word to doubleword

And some new quadword (128-bit) integer operations;

- Multiply-by-10 extend/write-carry
- Decimal convert from/to signed (binary) quadword
- Decimal convert from/to zoned (ASCII char)
- Decimal shift/round/truncate

The new VSX operations (with access to all 64 VSRs) were not directly applicable to 64-bit integer arithmetic:

- Scalar quad-precision floating-point
- Scalar and Vector convert with rounding
- Scalar and Vector extract/insert exponent/significand
- Scalar and Vector test data class
- Permute and Permute right index

An impressive list of operations that can be used for;

- Vectorizing long integer loops
- Implementing useful quadword integer operations which do not have corresponding PowerISA instructions
- implementing extended precision multiply and multiplicative inverse operations

The challenge is that useful operations available for POWER9 will need equivalent implementations for POWER8 and POWER7. Similarly for operations introduced for POWER8 will need POWER7 implementations. Also there are some obvious missing operations;

- Absolute Difference Doubleword (we have byte, halfword, and word)
- Average Doubleword (we have byte, halfword, and word)
- Extend Sign Doubleword to quadword (we have byte, halfword, and word)
- Multiply-sum Word (we have byte, halfword, and doubleword)
- Multiply Even/Odd Doublewords (we have byte, halfword, and word)

7.11.2.1 Challenges and opportunities

The stated goals for pveclib are:

- Provide equivalent functions across versions of the compiler.
- Provide equivalent functions across versions of the PowerISA.
- Provide complete arithmetic operations across supported C types.

So the first step is to provide implementations for the key POWER8 doubleword integer operations for older compilers. For example, some of the generic doubleword integer operations were not defined until GCC 6.0. Here we define the specific Compare Equal Unsigned Doubleword implementation:

```
static inline
vb64_t
vec_cmpequd (vui64_t a, vui64_t b)
{
  vb64_t result;
#ifdef _ARCH_PWR8
#if __GNUC__ >= 6
  result = vec_cmpeq(a, b);
#else
  __asm__(
      "vcmpequd %0,%1,%2;\n"
      : "=v" (result)
      : "v" (a),
      "v" (b)
      : );
#endif
#else
  // _ARCH_PWR7 implementation ...
#endif
  return (result);
}
```

The implementation checks if the compile target is POWER8 then checks of the compiler is new enough to use the generic vector compare built-in. If the generic built-in is not defined in <altivec.h> then we provide the equivalent inline assembler.

For POWER7 targets we don't have any vector compare doubleword operations and we need to define the equivalent operation using PowerISA 2.06B (and earlier) instructions. For example:

```
#else
  // _ARCH_PWR7 implementation ...
  vui8_t permute =
    { 0x04,0x05,0x06,0x07, 0x00,0x01,0x02,0x03,
      0x0C,0x0D,0x0E,0x0F, 0x08,0x09,0x0A,0x0B};
  vui32_t r, rr;
  r = (vui32_t) vec_cmpeq ((vui32_t) a, (vui32_t) b);
  if (vec_any_ne ((vui32_t) a, (vui32_t) b))
    {
       rr = vec_perm (r, r, permute);
       r= vec_and (r, rr);
    }
  result = (vb64_t)r;
#endif
```

Here we use Compare Equal Unsigned Word. If all words are equal, use the result as is. Otherwise, if any word elements are not equal, we do some extra work. For each doubleword, rotate the word compare result by 32-bits (here we use permute as we don't have rotate doubleword either). Then logical and the original word compare and rotated results to get the final doubleword compare results.

Similarly for all the doubleword compare variants. Similarly for doubleword; add, subtract, maximum, minimum, shift, rotate, count leading zeros, population count, and Byte reverse.

7.11.2.2 More Challenges

Now we can look at the case where vector doubleword operations of interest don't have an equivalent instruction. Here interesting operations include those that are supported for other element sizes and types.

The simplest example is absolute difference which was introduced in PowerISA 3.0 for byte, halfword and word elements. From the implementation of vec_absduw() we see how to implement the operation for POWER8 using subtract, maximum, and minimum. For example:

```
static inline vui64_t
vec_absdud (vui64_t vra, vui64_t vrb)
{
  return vec_subudm (vec_maxud (vra, vrb), vec_minud (vra, vrb));
}
```

This works because pveclib provides implementations for min, max, and sub operations that work across GCC versions and provide processor specific implementations for POWER8/9 and POWER7.

Now we need to look at the multiply doubleword situation. We need implementations for vec_msumudm(), vec_muleud(), vec_mulhud(), vec_muloud(), and vec_muludm(). We saw in the implementations of vec_int32_ppc.h that multiply high and low/modulo can implemented using multiply and merge even/odd of that element size. Multiply low can also be implemented using the multiply sum and multiply odd of the next smaller element size. Also multiply-sum can be implemented using multiply even/odd and a couple of adds. And multiply even/odd can be implemented using multiply sum by supplying zeros to appropriate inputs/elements.

The above discussion has many circular dependencies. Eventually we need to get down to an implementation on each processor using actual hardware instructions. So what multiply doubleword operations does the PowerISA actually have from the list above:

- POWER9 added multiply-sum unsigned doubleword modulo but no multiply doubleword even/odd/modulo instructions.
- POWER8 added multiply even/odd/modulo word but no multiply-sum word instructions
- POWER7 and earlier we have the original VMX multiply even/odd halfword, and multiply-sum unsigned halfword modulo, but no multiply modulo halfword.

It seems the best implementation strategy uses;

- Multiply-sum doubleword for POWER9
- Multiply even/odd word for POWER8
- Multiply even/odd halfword for POWER7

We really care about performance and latency for POWER9/8. We need POWER7 to work correctly so we can test on and support *legacy* hardware. The rest is grade school math.

First we need to make sure we have implementations across the GCC versions 6, 7, and 8 for the instructions we need. For example:

```
static inline vui128_t
vec_msumudm (vui64_t a, vui64_t b, vui128_t c)
{
  vui128_t res;
#if defined (_ARCH_PWR9) && ((__GNUC__ >= 6) || (__clang_major__ >= 11))
  __asm__(
      "vmsumudm %0,%1,%2,%3;\n"
      : "=v" (res)
      : "v" (a), "v" (b), "v" (c)
      : );
#else
  vui128_t p_even, p_odd, p_sum;
  p_even = vec_muleud (a, b);
  p_odd  = vec_muloud (a, b);
  p_sum  = vec_adduqm (p_even, p_odd);
  res    = vec_adduqm (p_sum, c);
#endif
  return (res);
}
```

Note

> The _ARCH_PWR8 implementation above depends on vec_muleud() and vec_muloud() for which there are no hardware instructions. Hold that thought.

While we are it we can implement multiply-sum unsigned word modulo.

```
static inline vui64_t
vec_vmsumuwm (vui32_t vra, vui32_t vrb, vui64_t vrc)
{
  vui64_t peven, podd, psum;
  peven = vec_muleuw (vra, vrb);
  podd  = vec_mulouw (vra, vrb);
  psum  = vec_addudm (peven, podd);
  return vec_addudm (psum, vrc);
}
```

We will need this later.

Now we need to provide implementations of vec_muleud() and vec_muloud(). For example:

```
static inline vui128_t
vec_muleud (vui64_t a, vui64_t b)
{
#if __BYTE_ORDER__ == __ORDER_LITTLE_ENDIAN__
  return vec_vmuloud (a, b);
#else
  return vec_vmuleud (a, b);
#endif
}
```

The implementation above is just handling the pesky little endian transforms. The real implementations are in vec_vmuleud() and vec_vmuloud() which implement the operation as if the PowerISA included such an instruction. These implementation is NOT endian sensitive and the function is stable across BE/LE implementations. For example:

```
static inline vui128_t
vec_vmuleud (vui64_t a, vui64_t b)
{
  vui64_t res;
#if defined (_ARCH_PWR9) && ((__GNUC__ >= 6) || (__clang_major__ >= 11))
  const vui64_t zero = { 0, 0 };
  vui64_t b_eud = vec_mrgahd ((vui128_t) b, (vui128_t) zero);
  __asm__(
      "vmsumudm %0,%1,%2,%3;\n"
      : "=v" (res)
      : "v" (a), "v" (b_eud), "v" (zero)
      : );
#else
#ifdef _ARCH_PWR8
  const vui64_t zero = { 0, 0 };
  vui64_t p0, p1, pp10, pp01;
  vui32_t m0, m1;
  // Need the endian invariant merge word high here
#if __BYTE_ORDER__ == __ORDER_LITTLE_ENDIAN__
  // Nullify the little endian transform
  m0 = vec_mergel ((vui32_t) b, (vui32_t) b);
#else
  m0 = vec_mergeh ((vui32_t) b, (vui32_t) b);
#endif
  m1 = (vui32_t) vec_xxspltd ((vui64_t) a, 0);
  // Need the endian invariant multiply even/odd word here
#if __BYTE_ORDER__ == __ORDER_LITTLE_ENDIAN__
  // Nullify the little endian transform
  p1 = vec_muleuw (m1, m0);
  p0 = vec_mulouw (m1, m0);
#else
  p1 = vec_mulouw (m1, m0);
  p0 = vec_muleuw (m1, m0);
#endif
  // res[1] = p1[1];  res[0] = p0[0];
  res = vec_pasted (p0, p1);
  // pp10[1] = p1[0]; pp10[0] = 0;
  // pp01[1] = p0[1]; pp01[0] = 0;
  // Need the endian invariant merge algebraic high/low here
  pp10 = (vui64_t) vec_mrgahd ((vui128_t) zero, (vui128_t) p1);
  pp01 = (vui64_t) vec_mrgald ((vui128_t) zero, (vui128_t) p0);
  // pp01 = pp01 + pp10.
  pp01 = (vui64_t) vec_adduqm ((vui128_t) pp01, (vui128_t) pp10);
  // res = res + (pp01 « 32)
  pp01 = (vui64_t) vec_sld ((vi32_t) pp01, (vi32_t) pp01, 4);
  res = (vui64_t) vec_adduqm ((vui128_t) pp01, (vui128_t) res);
#else
  // _ARCH_PWR7 implementation ...
#endif
#endif
  return ((vui128_t) res);
}
```

The _ARCH_PWR9 implementation uses the multiply-sum doubleword operation but implements the multiply even behavior by forcing the contents of doubleword element 1 of [VRB] and the contents of [VRC] to 0.

The _ARCH_PWR8 implementation looks ugly but it works. It starts with some merges and splats to get inputs columns lined up for the multiply. Then we use (POWER8 instructions) Multiply Even/Odd Unsigned Word to generate doubleword partial products. Then more merges and a rotate to line up the partial products for summation as the final quadword product.

Individually vec_vmuleud() and vec_vmuloud() execute with a latency of 21-23 cycles on POWER8. Normally these operations are used and scheduled together as in the POWER8 implementation of vec_msumudm() or vec_mulhud(). Good scheduling by the compiler and pipelining keeps the POWER8 latency in the 28-32 cycle range. For example, the vec_mulhud() implementation:

```
static inline vui64_t
vec_mulhud (vui64_t vra, vui64_t vrb)
{
  return vec_mrgahd (vec_vmuleud (vra, vrb), vec_vmuloud (vra, vrb));
}
```

Generates the following code for POWER8:

```
vspltisw v0,0
xxmrghw vs33,vs35,vs35
xxspltd vs45,vs34,0
xxmrglw vs35,vs35,vs35
vmulouw v11,v13,v1
xxspltd vs34,vs34,1
xxmrghd vs41,vs32,vs43
vmulouw v12,v2,v3
vmuleuw v13,v13,v1
vmuleuw v2,v2,v3
xxmrghd vs42,vs32,vs44
xxmrgld vs33,vs32,vs45
xxmrgld vs32,vs32,vs34
xxpermdi vs44,vs34,vs44,1
vadduqm v1,v1,v9
xxpermdi vs45,vs45,vs43,1
vadduqm v0,v0,v10
vsldoi  v1,v1,v1,4
vsldoi  v0,v0,v0,4
vadduqm v2,v1,v13
vadduqm v0,v0,v12
xxmrghd vs34,vs34,vs32
```

The POWER9 latencies for this operation range from 5-7 (for vmsumudm itself) to 11-16 (for vec_mulhud()). The additional latency reflects zero constant vector generation and merges required to condition the inputs and output. For these operations the vec_msumudm(), *vrc* operand is always zero. Selecting the even/odd doubleword for input requires a merge low/high. And selecting the high doubleword for multiply high require a final merge high.

vec_mulhud() generates the following code for POWER9:

```
xxspltib vs32,0
xxmrghd vs33,vs35,vs32
xxmrgld vs35,vs32,vs35
vmsumudm v1,v2,v1,v0
vmsumudm v2,v2,v3,v0
xxmrghd vs34,vs33,vs34
```

Wrapping up the doubleword multiplies we should look at the multiply low (AKA Multiply Unsigned Doubleword Modulo). The POWER9 implementation is similar to vec_mulhud () and the generated code is similar to the example above.

Multiply low doubleword is a special case, as we are discarding the highest partial doubleword product. For POWER8 we can optimize for that case using multiply odd and multiply-sum word operations. Also as we are only generating doubleword partial products we only need add doubleword modulo operations to sum the results. This avoids the more expensive add quadword operation required for the general case. The fact that vec_vmsumuwm() is only a software construct is not an issue. It expands into hardware multiple even/odd word and add doubleword instructions that the compiler can schedule and optimize.

Here vec_mulouw() generates low order partial product. Then vec_vrld () and vec_vmsumuwm() generate doubleword sums of the two middle order partial products. Then vec_vsld() shifts the middle order partial sum left 32-bits (discarding the unneeded high order 32-bits). Finally sum the low and middle order partial doubleword products to produce the multiply-low doubleword result. For example, this POWER8 only implementation:

```
static inline vui64_t
vec_muludm (vui64_t vra, vui64_t vrb)
{
  vui64_t s32 = { 32, 32 }; // shift / rotate amount.
  vui64_t z = { 0, 0 };
  vui64_t t2, t3, t4;
  vui32_t t1;
  t1 = (vui32_t) vec_vrld (vrb, s32);
#if __BYTE_ORDER__ == __ORDER_LITTLE_ENDIAN__
  // Nullify the little endian transform, really want mulouw here.
  t2 = vec_muleuw ((vui32_t) vra, (vui32_t) vrb);
#else
  t2 = vec_mulouw ((vui32_t) vra, (vui32_t) vrb);
#endif
  t3 = vec_vmsumuwm ((vui32_t) vra, t1, z);
  t4 = vec_vsld (t3, s32);
  return (vui64_t) vec_vaddudm (t4, t2);
}
```

Which generates the following for POWER8:

```
addis   r9,r2,.rodata.cst16+0x60@ha
addi    r9,r9,.rodata.cst16+0x60@l
lxv     vs33,0,r9
vmulouw v13,v2,v3
vrld    v0,v3,v1
vmulouw v3,v2,v0
vmuleuw v2,v2,v0
vaddudm v2,v3,v2
vsld    v2,v2,v1
vaddudm v2,v13,v2
```

Note

> The addition of *zeros* to the final sum of vec_vmsumuwm() (*vec_addudm (psum, vrc)*)has been optimized away by the compiler. This eliminates the xxspltib and one vaddudm instruction from the final code sequence.

And we can assume that the constant load of *{ 32, 32 }* will be common-ed with other operations or hoisted out of loops. So the shift constant can be loaded early and vrld is not delayed. This keeps the POWER8 latency in the 19-28 cycle range.

7.11.2.3 Loading small Doubleword constants

Programming with vector doubleword integers will need doubleword constants for masking and arithmetic operations. Doubleword *splat* constants are common in vectorized long integer code for arithmetic, comparison, and mask operations. For example:

```
vui64_t
__test_incud_V0 (vui64_t vra)
{
  // increament unsigned doubleword elements
  return vra + 1;
}
```

The endian sensitive macros from vec_common_ppc.h can be used to construct doubleword integer constants. For example:

```
const vui64_t dw_one = CONST_VINT64_DW(1, 1);
const vui64_t dw_ten = CONST_VINT64_DW(10, 10);
const vui64_t dw_sign_mask = (vui64_t) CONST_VINT128_W(0x80000000, 0x0,
                                                        0x80000000, 0x0);
```

In most cases this compiler will allocate these constant values to the read-only data (.rodata) section. When these constants are referenced in programming operations the compiler generates the appropriate vector loads. For example the GCC V11 generates the following for the **-mcpu=power8** target:

```
addis   r9,r2,.rodata.cst16+0x30@toc@ha
addi    r9,r9,.rodata.cst16+0x30@toc@l
lvx     v0,0,r9    # Load { 1, 1 }
vaddudm v2,v2,v0   # vra + 1
```

The addis/addi/lvx pattern is common to loading most vector constants for POWER8 and earlier.

For some odd reason the compiler might generate the sequence:

```
addis   r9,r2,.rodata.cst16+0x30@toc@ha
addi    r9,r9,.rodata.cst16+0x30@toc@l
rldicr  r9,r9,0,59
lxvd2x  vs0,0,r9
xxswapd vs0,vs0
```

for **-mcpu=power8** ppc64le targets.

The *Load VSX Vector Dword∗2 Indexed (**lxvd2x**)* would be required if the compiler could not know that the data was quadword aligned. The **lxvd2x** instruction handles unaligned access but requires the *little endian adjustment* (xxswapd). However the compiler controls the allocation and alignment of vector constants in .rodata and already insures quadword alignment.

Note

> This is has the look of a compiler phase error bug where important information is lost between compiler phases.

For the **-mcpu=power9** (and later) target GCC uses the *Load VXS Vector (**lxv**)* instruction:

```
addis   r9,r2,.rodata.cst16+0x30@toc@ha
addi    r9,r9,.rodata.cst16+0x30@toc@l
lxv     v2,0(r9)
```

The first sequence is expected for POWER8 as PowerISA 2.07B does not have any displacement form (D-Form) vector (VSX) load/store instructions. The compiler allocates constants to the .rodata sections and the linker collects .rodata from object files into a combined executable .rodata section. This section is placed near the *Table of Contents (**TOC**)* section. The ABI dedicates R2 as the base address **.TOC.** for the TOC and adjacent sections.

The *Add Immediate Shifted (addis) Add Immediate (addi)* sequence above computes a signed 32-bit **.TOC.** relative offset to a specific .rodata quadword. Two instructions are required as; *addis* provides the *high adjusted (**@ha**)* 16-bits shifted left 16-bits, while *addi* provides the *low (**@l**)* 16-bits. The sum of R2 and these immediate values is the 64-bit effective address of a .rodata constant value. A signed 32-bit offset is large enough to support most (*-mcmodel=medium*) program and library executables.

The load itself has a 5-cycle latency assuming a L1 cache hit. The three instruction sequence is sequentially dependent and requires 9-cycles latency (minimum) to execute. A L1 cache miss will increase the latency by 7-28 cycles, assuming the data resides in the L2/L3 caches.

7.11.2.3.1 Optimizing loads from .rodata

However the compiler is not following the recommendations of Power ISA 2.07B, *Book II, Chapter 2.1 Performance-Optimized Instruction Sequences*. This chapter recommends a specific pattern for the addi/lvx sequence. For example:

```
addis   rA,r2,.rodata.cst16+0x30@toc@ha
addi    rx,0,.rodata.cst16+0x30@toc@l
lvx     v2,rA,rx
```

In this case rx can be any GPR (including r0) while RA must be a valid base (r1 <-> r31) register.

The POWER8 implementation allows for *Instruction Funsion* combining information from two *adjacent*t instructions into one (internal) instruction so that it executes faster than the non-fused case. Effectively the addi/lvx combination above becomes a D-Form load vector instruction.

There are additional restrictions on the definition of *adjacent*:

- The instruction must be in the same dispatch group.
  - In single-threaded mode, up to six non-branch and up to two branch instructions (6/2 groups).
  - In multi-threaded mode, up to three non-branch and up to one branch instructions (3/1 groups).
- Without any intervening branch instructions.
- Instructions may span an I-cache line, with both lines fetched and residing in the i-buffer.

This can reduce the latency from 9 to 7-cycles. This would be true even without *Instruction Funsion* as the addis/addi instructions are now independent and can execute in parallel.

The sequence generated for POWER9 is even more disappointing. The lxv is a D-Form (DQ) instruction and the displacement operand could be used to replace the addi instruction. For example: **-mcpu=power9** target:

```
addis   r9,r2,.rodata.cst16+0x30@toc@ha
lxv     v2,.rodata.cst16+0x30@toc@l(r9)
```

This provides the equivalent 32-bit TOC relative displacement with one less instructions and reduced latency of 7-cycles.

7.11.2.3.2 Alternatives to loading from .rodata

This is all a little cumbersome and it seems like there should be a better/faster way. Any instruction sequence that loads quadword integer constants in:

- three instruction or less,
- latency of 6 cycles or less,
- and avoids cache misses

is a good deal.

The base (Altivec) vector ISA included Vector Splat Immediate Signed Byte/Halfword/Word instructions. These are fast (2-cycle latency) and convenient for small integer constants in the range -16 to 15. So far the ISA has not added doubleword or quadword forms of *splat immediate*.

POWER9 added a VSX Vector Splat Immediate Byte (xxspltib) instruction. This expands the immediate range to -128 to 127 but does not include larger element sizes. POWER9 does provide Vector Extend Sign Byte To Word/Doubleword (vextsb2w/vextsb2d) instructions. For example the two instruction sequence:

```
xxspltib vs34,127
vextsb2d v2,v2
```

can generate a doubleword splat immediate for integers in the range -128 to 127 with a cycle latency of 5-cycles.

Note

> POWER10 does add the interesting *VSX Vector Splat Immediate Double-Precision* instruction. This is a 64-bit instruction with a 32-bit single precision immediate operand. Interesting but not helpful for doubleword integer.

7.11.2.3.3 Some special quadword constants

The GCC compiler does recognize some vector constants as special case. For example:

```
vi128_t
__test_splatisq_n1_V0 (void)
{
  const vui32_t q_ones = {-1, -1, -1, -1};
  return (vi128_t) q_ones;
}
vi128_t
__test_splatisq_0_V0 (void)
{
  const vui32_t q_zero = {0, 0, 0, 0};
  return (vi128_t) q_zero;
}
```

will generate:

```
0000000000000080 <__test_splatisq_n1_V0>:
     vspltisw v2,-1
     blr
00000000000000a0 <__test_splatisq_0_V0>:
     vspltisw v2,0
     blr
```

As we will see the all zero/ones constants are common building blocks. So the compiler should treat these as common sub expressions across all operations using those constants.

7.11.2.3.4 Defining our own vec_splat_s64

So the compiler can do clever things with vector constants. But so far these are the only examples I have found. Other cases that you might expect to be a special case are not. For example:

```
vui64_t
__test_splatudi_15_V1 (void)
{
  return vec_splats ((unsigned long long) 12);
}
vui64_t
__test_splatudi_15_V0 (void)
{
  const vui64_t dw_15 = CONST_VINT64_DW(15, 15);
  return dw_15;
}
```

both generate the 3 instruction (9-cycle) load from .rodata sequence. Also constants using the vector long long or __int128 types may fail to compile on older versions of the compiler.

We can generate small constants in the range -16 <-> 15 with using the following pattern:

```
vi64_t
__test_splatsdi_15_V1 (void)
{
  vi32_t vwi = vec_splat_s32 (15);
  return vec_unpackl (vwi);
}
```

Which should generate:

```
0000000000000040 <__test_splatisd_15_v2>:
      vspltisw v2,15
      vupklsw v2,v2
      blr
```

Here we use the vec_splat_s32(15) intrinsic to generate *Vector Splat Immediate Signed Word (**vspltisw**)* to splat the value 15 across word elements of *vwi*. Then vec_unpackl (vwi) to generate *Vector Unpack Low Signed Word **vupklsw*** which sign extends the 2 low words of *vwi* to signed doubleword elements. This sequence is only 2 instructions and will execute with 4-cycle latency.

Note

> unfortunately GCC compilers after GCC-8 will recognize this sequence and convert it back to the three instruction .rodata load sequence. See: GCC PR 104124 Until PR 104124 is fixed the following work-around is used for the PVECLIB implementation.

Putting this all together we can create a static inline function to generate small doubleword constants (in the range -16 to 15). For example:

```
static inline vi64_t
vec_splat_s64_PWR8 (const int sim)
{
  vi64_t result;
  if (__builtin_constant_p (sim) && ((sim >= -16) && (sim < 16)))
    {
      vi32_t vwi = vec_splat_s32 (sim);
      if (__builtin_constant_p (sim) && ((sim == 0) || (sim == -1)))
        {
          // Special case for -1 and 0. Skip vec_unpackl().
          result = (vi64_t) vwi;
        } else {
          // For P8 can use either vupklsh or vupklsw but P7 only has
          // vupklsh. Given the reduced range, Either works here.
          // Unpack signed HW works here because immediate value fits
          // into the low HW and sign extends to high HW of each word.
          // Unpack will expand the low HW to low word and high HW
          // (sign extend) into the high word of each DW.
          // Unpack low/high (or endian) will not change the result.
#if defined (__GNUC__) && (__GNUC__ == 8)
          // GCC 8 (AT12) handle this correctly.
          result = (vi64_t) vec_vupklsh ((vi16_t) vwi);
#else
          // But GCC 9+ optimized the above to be load from .rodata.
          // With a little register pressure it adds some gratuitous store/reloads.
          // So the following work-around is required.
          __asm__(
              "vupklsh %0,%1;"
              : "=v" (result)
              : "v" (vwi)
              : );
#endif
        }
    }
  else
    result = vec_splats ((signed long long) sim);
  return (result);
}
```

This version uses only <altivec.h> intrinsics supported by POWER8 and earlier. For constants in the range (-16 to 15) the range is divided into two groups:

- Special values -1 and 0 that can be generated in a single instruction.
- Values -16 to 15 that require the vwi constant to sign extend.

Values outside this range use the vec_splats() intrinsic which will generate the appropriate quadword constant in .rodata and the load sequence to retrieve that value.

For POWER9 and later we can use the vec_splats() intrinsic which (so far) generates the xxspltib/vextsb2d sequence for the constant range -128 to 127.

```
static inline vi64_t
vec_splat_s64_PWR9 (const int sim)
{
  return vec_splats ((signed long long) sim);
}
```

7.11.3 Endian problems with doubleword operations

From the examples above we see that the construction of higher precision multiplies requires significant massaging of input and output elements. Here merge even/odd, merge high/low, swap, and splat doubleword element operations are commonly used.

PowerISA 2.06 VSX (POWER7) added the general purpose Vector Permute Doubleword Immediate (xxpermdi). The compiler generates some form of xxpermdi for the doubleword (double float, long int, bool long) merge/splat/swap operations. As xxpermdi's element selection is an immediate field, most operations require only a single instruction. All the merge/splat/swap doubleword variant are just a specific select mask value and the inputs to xxpermdi.

Which is very useful indeed for assembling, disassembling, merging, splatting, swapping, and pasting doubleword elements.

Of course it took several compiler releases to implement all the generic merge/splat/swap operations for the supported types. GCC 4.8 as the first to support vec_xxpermdi as a built-in. GCC 4.8 also supported the generic built-ins vec↩_mergeh, vec_mergel, and vec_splat for the vector signed/unsigned/bool long type. But endian sensitive vec_mergeh, vec_mergel, and vec_splat were not supported until GCC 7. And the generic vec_mergee, vec_mergeo built-ins where not supported until GCC 8.

But as we have explained in General Endian Issues and Endian problems with word operations the little endian transforms applied by the compiler can cause problems for developers of multi-precision libraries. The doubleword forms of the generic merge/splat operations etc. are no exception. This is especially annoying when the endian sensitive transforms are applied between releases of the compiler.

So we need a strategy to provide endian invariant merge/splat/swap operations to be used in multi-precision arithmetic. And another set of endian sensitive operations that are mandated by the OpenPOWER ABI.

First we need a safely endian invariant version of xxpermdi to use in building other variants:

- vec_permdi() provides the basic xxpermdi operation but nullifies the little endian transforms.

Then build the core set of endian invariant permute doubleword operations using vec_permdi():

- Merge algebraic high/low doubleword operations vec_mrgahd() and vec_mrgald().
- Merge the left and right most doublewords from a double quadword operation vec_pasted().
- Splat from the high/even or low/odd doubleword operation vec_xxspltd().
- Swap high and low doublewords operation vec_swapd().

We use the merge algebraic high/low doubleword operations in the implementation of vec_mulhud(), vec_mulhud(), vec_vmuleud(), and vec_vmuloud(). We use the vec_xxspltd operation in the implementation of vec_vrld(), vec_vmuleud(), and vec_vmuloud(). We use the paste doubleword (vec_pasted()) operation in the implementation of vec_vsrad(), vec_vmuleud(), and vec_vmuloud(). We use the swap doubleword operation in the implementation of vec_cmpequq(), vec_cmpneuq(), vec_muludq(), and vec_mulluq().

Then use the compilers **__BYTE_ORDER__ == ORDER_LITTLE_ENDIAN** conditional to invert the vec_permdi() select control for endian sensitive merge/splat doubleword operations:

- Merge even/odd doubleword operations vec_mrged() and vec_mrgod().
- Merge high/low doubleword operations vec_mrghd() and vec_mrgld().
- Splat even/odd doubleword operation vec_splatd().

7.11.4 Vector Doubleword Examples

Suppose we have a requirement to convert an array of 64-bit time-interval values that need to convert to timespec format. For simplicity we will also assume that the array is nicely (Quadword) aligned and an integer multiple of 2 doublewords or 4 words.

The PowerISA provides a 64-bit TimeBase register that clocks at a constant 512MHz. The TimeBase can be read directly as either the full 64-bit value or as 32-bit upper and lower halves. For this example we assume are dealing with longer intervals (greater than ∼8.38 seconds) so the full 64-bit TimeBase is required. TimeBase values of adjacent events are subtracted to generate the intervals stored in the array.

The timespec format is a struct of unsigned int fields for seconds and nanoseconds. So the task is to convert the 512MHz 64-bit TimeBase intervals to seconds and remaining clock ticks. Then convert the remaining (subsecond) clock ticks from 512MHz to nanoseconds. The separate seconds and nanoseconds are combined in the timespec structure.

First we need to separate the raw TimeBase into the integer seconds and (subsecond) clock-ticks. Normally scalar codes would use integer divide/modulo by 512000000. Did I mention that the PowerISA vector unit does not have a integer divide operation?

Instead we can use the multiplicative inverse which is a scaled fixed point fraction calculated from the original divisor. This works nicely if the fixed radix point is just before the 64-bit fraction and we have a multiply high (vec_mulhud()) operation. Multiplying a 64-bit unsigned integer by a 64-bit unsigned fraction generates a 128-bit product with 64-bits above (integer) and below (fraction) the radix point. The high 64-bits of the product is the integer quotient.

It turns out that generating the multiplicative inverse can be tricky. To produce correct results over the full range requires, possible pre-scaling and post-shifting, and sometimes a corrective addition is necessary. Fortunately the mathematics are well understood and are commonly used in optimizing compilers. Even better, Henry Warren's book has a whole chapter on this topic.

See also

"Hacker's Delight, 2nd Edition," Henry S. Warren, Jr, Addison Wesley, 2013. Chapter 10, Integer Division by Constants.

In the chapter above;

Figure 10-2 Computing the magic number for unsigned division.

provides a sample C function for generating the magic number (actually a struct containing; the magic multiplicative inverse, "add" indicator, and the shift amount.).

For the divisor 512000000 this is { 4835703278458516699, 0 , 27 }:

- the multiplier is 4835703278458516699.
- no corrective add of the dividend is required.
- the final shift is 27-bits right.

```
// Magic numbers for multiplicative inverse to divide by 512,000,000
// are 4835703278458516699 and shift right 27 bits.
const vui64_t mul_invs_clock =
  { 4835703278458516699UL, 4835703278458516699UL };
const int shift_clock = 27;
// Need const for TB clocks/second to extract remainder.
const vui64_t tb_clock_sec =
  { 512000000, 512000000};
vui64_t tb_v, tmp, tb_clocks, seconds, nseconds;
vui64_t timespec1, timespec2;
// extract integer seconds from timebase vector.
tmp = vec_mulhud (tb_v, mul_invs_clock);
seconds = vec_srdi (tmp, shift_clock);
// Extract the remainder in tb clock ticks.
tmp = vec_muludm (seconds, tb_clock_sec);
tb_clocks = vec_sub (tb_v, tmp);
```

Next we need to convert the subseconds from TimeBase clock-ticks to nanoseconds. The subsecond remainder is now small enough (compared to a doubleword) that we can perform the conversion *in place*. The nanosecond conversion is ((tb_clocks ∗ 1000000000) / 512000000). And we can reduce this to ((tb_clocks ∗ 1000) / 512). We still have a small multiply but the divide can be converted to shift right of 9-bits.

```
const int shift_512 = 9;
const vui64_t nano_512 =
  { 1000, 1000};
// Convert 512MHz timebase to nanoseconds.
// nseconds = tb_clocks * 1000000000 / 512000000
// reduces to (tb_clocks * 1000) » 9
tmp = vec_muludm (tb_clocks, nano_512);
nseconds = vec_srdi (tmp, shift_512);
```

Finally we need to merge the vectors of seconds and nanoseconds into vectors of timespec. So far we have been working with 64-bit integers but the timespec is a struct of 32-bit (word) integers. Here 32-bit seconds and nanosecond provided sufficient range and precision. So the final step *packs* a pair of 64-bit timespec values into a vector of two 32-bit timespec values, each containing 2 32-bit (second, nanosecond) values.

```
timespec1 = vec_mergeh (seconds, nseconds);
timespec2 = vec_mergel (seconds, nseconds);
// seconds and nanoseconds fit int 32-bits after conversion.
// So pack the results and store the timespec.
*timespec++ = vec_vpkudum (timespec1, timespec2);
```

Note

vec_sub(), vec_mergeh(), and vec_mergel() are existing altivec.h generic built-ins.

vec_vpkudum() is an existing altivec.h built-in that is only defined for **_ARCH_PWR8** and later. This header insures that vec_vpkudum is defined for older compilers and provides an functional equivalent implementation for POWER7.

7.11.4.1 Vectorized 64-bit TimeBase conversion example

Here is the complete vectorized 64-bit TimeBase to timespec conversion example:

```
void
example_dw_convert_timebase (vui64_t *tb, vui32_t *timespec, int n)
{
  // Magic numbers for multiplicative inverse to divide by 512,000,000
  // are 4835703278458516699 and shift right 27 bits.
  const vui64_t mul_invs_clock =
    { 4835703278458516699UL, 4835703278458516699UL };
  const int shift_clock = 27;
  // Need const for TB clocks/second to extract remainder.
  const vui64_t tb_clock_sec =
    { 512000000, 512000000};
  const int shift_512 = 9;
  const vui64_t nano_512 =
    { 1000, 1000};
  vui64_t tb_v, tmp, tb_clocks, seconds, nseconds;
  vui64_t timespec1, timespec2;
  int i;
  for (i = 0; i < n; i++)
    {
      tb_v = *tb++;
      // extract integer seconds from timebase vector.
      tmp = vec_mulhud (tb_v, mul_invs_clock);
      seconds = vec_srdi (tmp, shift_clock);
      // Extract remainder in tb clock ticks.
      tmp = vec_muludm (seconds, tb_clock_sec);
      tb_clocks = vec_sub (tb_v, tmp);
      // Convert 512MHz timebase to nanoseconds.
      // nseconds = tb_clocks * 1000000000 / 512000000
      // reduces to (tb_clocks * 1000) » 9
      tmp = vec_muludm (tb_clocks, nano_512);
      nseconds = vec_srdi (tmp, shift_512);
      // Use merge high/low to interleave seconds and nseconds
      // into timespec.
      timespec1 = vec_mergeh (seconds, nseconds);
      timespec2 = vec_mergel (seconds, nseconds);
      // seconds and nanoseconds fit int 32-bits after conversion.
      // So pack the results and store the timespec.
      *timespec++ = vec_vpkudum (timespec1, timespec2);
    }
}
```

7.11.5 Performance data.

High level performance estimates are provided as an aid to function selection when evaluating algorithms. For background on how *Latency* and *Throughput* are derived see: Performance data.

7.11.6 Function Documentation

7.11.6.1 vec_absdud()

```
static vui64_t vec_absdud (
              vui64_t vra,
              vui64_t vrb )  [inline], [static]
```

Vector Absolute Difference Unsigned Doubleword.

Compute the absolute difference for each doubleword. For each unsigned doubleword, subtract VRB[i] from VRA[i] and return the absolute value of the difference.

processor	Latency	Throughput
power8	4	1/cycle
power9	5	1/cycle

Parameters

vra	vector of 2 x unsigned doublewords
vrb	vector of 2 x unsigned doublewords

Returns

vector of the absolute differences.

7.11.6.2 vec_addudm()

```
static vui64_t vec_addudm (
             vui64_t a,
             vui64_t b )  [inline], [static]
```

Vector Add Unsigned Doubleword Modulo.

Add two vector long int values and return modulo 64-bits result.

processor	Latency	Throughput
power8	2	2/cycle
power9	2	2/cycle

Parameters

a	128-bit vector long int.
b	128-bit vector long int.

Returns

vector long int sums of a and b.

7.11.6.3 vec_clzd()

```
static vui64_t vec_clzd (
             vui64_t vra )  [inline], [static]
```

Vector Count Leading Zeros Doubleword for unsigned long long elements.

Count the number of leading '0' bits (0-64) within each doubleword element of a 128-bit vector.

For POWER8 (PowerISA 2.07B) or later use the Vector Count Leading Zeros Doubleword instruction **vclzd**. Otherwise use sequence of pre 2.07 VMX instructions.

processor	Latency	Throughput
power8	2	2/cycle
power9	2	2/cycle

Parameters

vra	a 128-bit vector treated as 2 x 64-bit unsigned long long (doubleword) elements.

Returns

128-bit vector with the leading zeros count for each doubleword element.

7.11.6.4 vec_cmpeqsd()

```
static vb64_t vec_cmpeqsd (
             vi64_t a,
             vi64_t b )  [inline], [static]
```

Vector Compare Equal Signed Doubleword.

Compare each signed long (64-bit) integer and return all '1's, if a[i] == b[i], otherwise all '0's.

For POWER8 (PowerISA 2.07B) or later, use the Vector Compare Equal Unsigned DoubleWord (**vcmpequd**) instruction. Otherwise use boolean logic using word compares.

processor	Latency	Throughput
power8	2	2/cycle
power9	3	2/cycle

Parameters

a	128-bit vector treated as 2 x 64-bit signed long integer (dword) elements.
b	128-bit vector treated as 2 x 64-bit signed long integer (dword) elements.

Returns

128-bit vector with each dword boolean reflecting compare equal result for each element.

7.11.6.5 vec_cmpequd()

```
static vb64_t vec_cmpequd (
             vui64_t a,
             vui64_t b )  [inline], [static]
```

Vector Compare Equal Unsigned Doubleword.

Compare each unsigned long (64-bit) integer and return all '1's, if a[i] == b[i], otherwise all '0's.

For POWER8 (PowerISA 2.07B) or later, use the Vector Compare Equal Unsigned DoubleWord (**vcmpequd**) instruction. Otherwise use boolean logic using word compares.

processor	Latency	Throughput
power8	2	2/cycle
power9	3	2/cycle

Parameters

a	128-bit vector treated as 2 x 64-bit unsigned long integer (dword) elements.
b	128-bit vector treated as 2 x 64-bit unsigned long integer (dword) elements.

Returns

128-bit vector with each dword boolean reflecting compare equal result for each element.

7.11.6.6 vec_cmpgesd()

```
static vb64_t vec_cmpgesd (
            vi64_t a,
            vi64_t b )  [inline], [static]
```

Vector Compare Greater Than or Equal Signed Doubleword.

Compare each signed long (64-bit) integer and return all '1's, if a[i] >= b[i], otherwise all '0's. Use vec_cmpgtsd with parameters reversed to implement vec_cmpltud, then return the logical inverse.

processor	Latency	Throughput
power8	4	2/cycle
power9	5	2/cycle

Parameters

a	128-bit vector treated as 2 x 64-bit signed long integer (dword) elements.
b	128-bit vector treated as 2 x 64-bit signed long integer (dword) elements.

Returns

128-bit vector with each dword boolean reflecting compare greater then or equal result for each element.

7.11.6.7 vec_cmpgeud()

```
static vb64_t vec_cmpgeud (
            vui64_t a,
            vui64_t b )  [inline], [static]
```

Vector Compare Greater Than or Equal Unsigned Doubleword.

Compare each unsigned long (64-bit) integer and return all '1's, if a[i] >= b[i], otherwise all '0's. Use vec_cmpgtud with parameters reversed to implement vec_cmpltud, then return the logical inverse.

processor	Latency	Throughput
power8	4	2/cycle
power9	5	2/cycle

Parameters

a	128-bit vector treated as 2 x 64-bit unsigned long integer (dword) elements.
b	128-bit vector treated as 2 x 64-bit unsigned long integer (dword) elements.

Returns

128-bit vector with each dword boolean reflecting compare greater then or equal result for each element.

7.11.6.8 vec_cmpgtsd()

```
static vb64_t vec_cmpgtsd (
            vi64_t a,
            vi64_t b )  [inline], [static]
```

Vector Compare Greater Than Signed Doubleword.

Compare each signed long (64-bit) integer and return all '1's, if a[i] > b[i], otherwise all '0's.

For POWER8 (PowerISA 2.07B) or later use the Vector Compare Greater Than Unsigned DoubleWord (**vcmpgtsd**) instruction. Otherwise use boolean logic using word compares.

processor	Latency	Throughput
power8	2	2/cycle
power9	3	2/cycle

Parameters

a	128-bit vector treated as 2 x 64-bit unsigned long integer (dword) elements.
b	128-bit vector treated as 2 x 64-bit unsigned long integer (dword) elements.

Returns

128-bit vector with each dword boolean reflecting compare greater result for each element.

7.11.6.9 vec_cmpgtud()

```
static vb64_t vec_cmpgtud (
              vui64_t a,
              vui64_t b )  [inline], [static]
```

Vector Compare Greater Than Unsigned Doubleword.

Compare each unsigned long (64-bit) integer and return all '1's, if a[i] > b[i], otherwise all '0's.

For POWER8 (PowerISA 2.07B) or later use the Vector Compare Greater Than Unsigned DoubleWord (**vcmpgtud**) instruction. Otherwise use boolean logic using word compares.

processor	Latency	Throughput
power8	2	2/cycle
power9	3	2/cycle

Parameters

a	128-bit vector treated as 2 x 64-bit unsigned long integer (dword) elements.
b	128-bit vector treated as 2 x 64-bit unsigned long integer (dword) elements.

Returns

128-bit vector with each dword boolean reflecting compare greater result for each element.

7.11.6.10 vec_cmplesd()

```
static vb64_t vec_cmplesd (
              vi64_t a,
              vi64_t b )  [inline], [static]
```

Vector Compare Less Than Equal Signed Doubleword.

Compare each signed long (64-bit) integer and return all '1's, if a[i] > b[i], otherwise all '0's. Use vec_cmpgtsd with parameters reversed to implement vec_cmpltsd then return the logical inverse.

processor	Latency	Throughput
power8	4	2/cycle
power9	5	2/cycle

Parameters

a	128-bit vector treated as 2 x 64-bit signed long integer (dword) elements.
b	128-bit vector treated as 2 x 64-bit signed long integer (dword) elements.

Returns

128-bit vector with each dword boolean reflecting compare greater result for each element.

7.11.6.11 vec_cmpleud()

```
static vb64_t vec_cmpleud (
             vui64_t a,
             vui64_t b )  [inline], [static]
```

Vector Compare Less Than Equal Unsigned Doubleword.

Compare each unsigned long (64-bit) integer and return all '1's, if a[i] > b[i], otherwise all '0's. Use vec_cmpgtud with parameters reversed to implement vec_cmpltud. Use vec_cmpgtud then return the logical inverse.

processor	Latency	Throughput
power8	4	2/cycle
power9	5	2/cycle

Parameters

a	128-bit vector treated as 2 x 64-bit unsigned long integer (dword) elements.
b	128-bit vector treated as 2 x 64-bit unsigned long integer (dword) elements.

Returns

128-bit vector with each dword boolean reflecting compare greater result for each element.

7.11.6.12 vec_cmpltsd()

```
static vb64_t vec_cmpltsd (
              vi64_t a,
              vi64_t b )  [inline], [static]
```

Vector Compare less Than Signed Doubleword.

Compare each signed long (64-bit) integer and return all '1's, if a[i] < b[i], otherwise all '0's. Use vec_cmpgtsd with parameters reversed to implement vec_cmpltsd.

processor	Latency	Throughput
power8	2	2/cycle
power9	3	2/cycle

Parameters

a	128-bit vector treated as 2 x 64-bit signed long integer (dword) elements.
b	128-bit vector treated as 2 x 64-bit signed long integer (dword) elements.

Returns

128-bit vector with each dword boolean reflecting compare less result for each element.

7.11.6.13 vec_cmpltud()

```
static vb64_t vec_cmpltud (
              vui64_t a,
              vui64_t b )  [inline], [static]
```

Vector Compare less Than Unsigned Doubleword.

Compare each unsigned long (64-bit) integer and return all '1's, if a[i] < b[i], otherwise all '0's. Use vec_cmpgtud with parameters reversed to implement vec_cmpltud.

processor	Latency	Throughput
power8	2	2/cycle
power9	3	2/cycle

Parameters

a	128-bit vector treated as 2 x 64-bit unsigned long integer (dword) elements.
b	128-bit vector treated as 2 x 64-bit unsigned long integer (dword) elements.

Returns

128-bit vector with each dword boolean reflecting compare less result for each element.

7.11.6.14 vec_cmpnesd()

```
static vb64_t vec_cmpnesd (
            vi64_t a,
            vi64_t b )  [inline], [static]
```

Vector Compare Not Equal Signed Doubleword.

Compare each signed long (64-bit) integer and return all '1's, if a[i] != b[i], otherwise all '0's. Use vec_cmpequd then return the logical inverse.

processor	Latency	Throughput
power8	4	2/cycle
power9	5	2/cycle

Parameters

a	128-bit vector treated as 2 x 64-bit signed long integer (dword) elements.
b	128-bit vector treated as 2 x 64-bit signed long integer (dword) elements.

Returns

128-bit vector with each dword boolean reflecting compare not equal result for each element.

7.11.6.15 vec_cmpneud()

```
static vb64_t vec_cmpneud (
            vui64_t a,
            vui64_t b )  [inline], [static]
```

Vector Compare Not Equal Unsigned Doubleword.

Compare each unsigned long (64-bit) integer and return all '1's, if a[i] != b[i], otherwise all '0's. Use vec_cmpequd then return the logical inverse.

processor	Latency	Throughput
power8	4	2/cycle
power9	5	2/cycle

Parameters

a	128-bit vector treated as 2 x 64-bit unsigned long integer (dword) elements.
b	128-bit vector treated as 2 x 64-bit unsigned long integer (dword) elements.

Returns

128-bit vector with each dword boolean reflecting compare not equal result for each element.

7.11.6.16 vec_cmpsd_all_eq()

```
static int vec_cmpsd_all_eq (
             vi64_t a,
             vi64_t b )  [inline], [static]
```

Vector Compare all Equal Signed Doubleword.

Compare each signed long (64-bit) integer and return true if all elements of a and b are equal.

processor	Latency	Throughput
power8	4-9	2/cycle
power9	3	2/cycle

Parameters

a	128-bit vector treated as 2 x 64-bit signed long integer (dword) elements.
b	128-bit vector treated as 2 x 64-bit signed long integer (dword) elements.

Returns

boolean int for all 128-bits, true if equal, false otherwise.

7.11.6.17 vec_cmpsd_all_ge()

```
static int vec_cmpsd_all_ge (
             vi64_t a,
             vi64_t b )  [inline], [static]
```

Vector Compare all Greater Than or Equal Signed Doubleword.

Compare each signed long (64-bit) integer and return true if all elements of a >= b.

processor	Latency	Throughput
power8	4-9	2/cycle
power9	3	2/cycle

Parameters

a	128-bit vector treated as 2 x 64-bit signed long integer (dword) elements.
b	128-bit vector treated as 2 x 64-bit signed long integer (dword) elements.

Returns

boolean int for all 128-bits, true if all Greater Than, false otherwise.

7.11.6.18 vec_cmpsd_all_gt()

```
static int vec_cmpsd_all_gt (
            vi64_t a,
            vi64_t b )  [inline], [static]
```

Vector Compare all Greater Than Signed Doubleword.

Compare each signed long (64-bit) integer and return true if all elements of a > b.

processor	Latency	Throughput
power8	4-9	2/cycle
power9	3	2/cycle

Parameters

a	128-bit vector treated as 2 x 64-bit signed long integer (dword) elements.
b	128-bit vector treated as 2 x 64-bit signed long integer (dword) elements.

Returns

boolean int for all 128-bits, true if all Greater Than, false otherwise.

7.11.6.19 vec_cmpsd_all_le()

```
static int vec_cmpsd_all_le (
            vi64_t a,
            vi64_t b )  [inline], [static]
```

Vector Compare all Less than equal Signed Doubleword.

Compare each signed long (64-bit) integer and return true if all elements of a <= b.

processor	Latency	Throughput
power8	4-9	2/cycle
power9	3	2/cycle

Parameters

a	128-bit vector treated as 2 x 64-bit signed long integer (dword) elements.
b	128-bit vector treated as 2 x 64-bit signed long integer (dword) elements.

Returns

boolean int for all 128-bits, true if all Greater Than, false otherwise.

7.11.6.20 vec_cmpsd_all_lt()

```
static int vec_cmpsd_all_lt (
            vi64_t a,
            vi64_t b )  [inline], [static]
```

Vector Compare all Less than Signed Doubleword.

Compare each signed long (64-bit) integer and return true if all elements of a < b.

processor	Latency	Throughput
power8	4-9	2/cycle
power9	3	2/cycle

Parameters

a	128-bit vector treated as 2 x 64-bit signed long integer (dword) elements.
b	128-bit vector treated as 2 x 64-bit signed long integer (dword) elements.

Returns

boolean int for all 128-bits, true if all Greater Than, false otherwise.

7.11.6.21 vec_cmpsd_all_ne()

```
static int vec_cmpsd_all_ne (
            vi64_t a,
            vi64_t b ) [inline], [static]
```

Vector Compare all Not Equal Signed Doubleword.

Compare each signed long (64-bit) integer and return true if all elements of a and b are not equal.

processor	Latency	Throughput
power8	4-9	2/cycle
power9	3	2/cycle

Parameters

a	128-bit vector treated as 2 x 64-bit signed long integer (dword) elements.
b	128-bit vector treated as 2 x 64-bit signed long integer (dword) elements.

Returns

boolean int for all 128-bits, true if equal, false otherwise.

7.11.6.22 vec_cmpsd_any_eq()

```
static int vec_cmpsd_any_eq (
            vi64_t a,
            vi64_t b ) [inline], [static]
```

Vector Compare any Equal Signed Doubleword.

Compare each signed long (64-bit) integer and return true if any elements of a and b are equal.

processor	Latency	Throughput
power8	4-9	2/cycle
power9	3	2/cycle

Parameters

a	128-bit vector treated as 2 x 64-bit signed long integer (dword) elements.
b	128-bit vector treated as 2 x 64-bit signed long integer (dword) elements.

Returns

boolean int for all 128-bits, true if equal, false otherwise.

7.11.6.23 vec_cmpsd_any_ge()

```
static int vec_cmpsd_any_ge (
            vi64_t a,
            vi64_t b ) [inline], [static]
```

Vector Compare any Greater Than or Equal Signed Doubleword.

Compare each signed long (64-bit) integer and return true if any elements of a >= b.

processor	Latency	Throughput
power8	4-9	2/cycle
power9	3	2/cycle

Parameters

a	128-bit vector treated as 2 x 64-bit signed long integer (dword) elements.
b	128-bit vector treated as 2 x 64-bit signed long integer (dword) elements.

Returns

boolean int for all 128-bits, true if all Greater Than, false otherwise.

7.11.6.24 vec_cmpsd_any_gt()

```
static int vec_cmpsd_any_gt (
            vi64_t a,
            vi64_t b ) [inline], [static]
```

Vector Compare any Greater Than Signed Doubleword.

Compare each signed long (64-bit) integer and return true if all elements of a > b.

processor	Latency	Throughput
power8	4-9	2/cycle
power9	3	2/cycle

Parameters

a	128-bit vector treated as 2 x 64-bit signed long integer (dword) elements.
b	128-bit vector treated as 2 x 64-bit signed long integer (dword) elements.

Returns

boolean int for all 128-bits, true if all Greater Than, false otherwise.

7.11.6.25 vec_cmpsd_any_le()

```
static int vec_cmpsd_any_le (
            vi64_t a,
            vi64_t b )  [inline], [static]
```

Vector Compare any Less than equal Signed Doubleword.

Compare each signed long (64-bit) integer and return true if any elements of a <= b.

processor	Latency	Throughput
power8	4-9	2/cycle
power9	3	2/cycle

Parameters

a	128-bit vector treated as 2 x 64-bit signed long integer (dword) elements.
b	128-bit vector treated as 2 x 64-bit signed long integer (dword) elements.

Returns

boolean int for any 128-bits, true if any Greater Than, false otherwise.

7.11.6.26 vec_cmpsd_any_lt()

```
static int vec_cmpsd_any_lt (
            vi64_t a,
            vi64_t b )  [inline], [static]
```

Vector Compare any Less than Signed Doubleword.

Compare each signed long (64-bit) integer and return true if any elements of a < b.

processor	Latency	Throughput
power8	4-9	2/cycle
power9	3	2/cycle

Parameters

a	128-bit vector treated as 2 x 64-bit signed long integer (dword) elements.
b	128-bit vector treated as 2 x 64-bit signed long integer (dword) elements.

Returns

boolean int for any 128-bits, true if any Greater Than, false otherwise.

7.11.6.27 vec_cmpsd_any_ne()

```
static int vec_cmpsd_any_ne (
             vi64_t a,
             vi64_t b )  [inline], [static]
```

Vector Compare any Not Equal Signed Doubleword.

Compare each signed long (64-bit) integer and return true if any elements of a and b are not equal.

processor	Latency	Throughput
power8	4-9	2/cycle
power9	3	2/cycle

Parameters

a	128-bit vector treated as 2 x 64-bit signed long integer (dword) elements.
b	128-bit vector treated as 2 x 64-bit signed long integer (dword) elements.

Returns

boolean int for any 128-bits, true if equal, false otherwise.

7.11.6.28 vec_cmpud_all_eq()

```
static int vec_cmpud_all_eq (
             vui64_t a,
             vui64_t b )  [inline], [static]
```

Vector Compare all Equal Unsigned Doubleword.

Compare each unsigned long (64-bit) integer and return true if all elements of a and b are equal.

processor	Latency	Throughput
power8	4-9	2/cycle
power9	3	2/cycle

Parameters

a	128-bit vector treated as 2 x 64-bit unsigned long integer (dword) elements.
b	128-bit vector treated as 2 x 64-bit unsigned long integer (dword) elements.

Returns

boolean int for all 128-bits, true if equal, false otherwise.

7.11.6.29 vec_cmpud_all_ge()

```
static int vec_cmpud_all_ge (
            vui64_t a,
            vui64_t b )  [inline], [static]
```

Vector Compare all Greater Than or Equal Unsigned Doubleword.

Compare each unsigned long (64-bit) integer and return true if all elements of a >= b.

processor	Latency	Throughput
power8	4-9	2/cycle
power9	3	2/cycle

Parameters

a	128-bit vector treated as 2 x 64-bit unsigned long integer (dword) elements.
b	128-bit vector treated as 2 x 64-bit unsigned long integer (dword) elements.

Returns

boolean int for all 128-bits, true if all Greater Than, false otherwise.

7.11.6.30 vec_cmpud_all_gt()

```
static int vec_cmpud_all_gt (
            vui64_t a,
            vui64_t b )  [inline], [static]
```

Vector Compare all Greater Than Unsigned Doubleword.

Compare each unsigned long (64-bit) integer and return true if all elements of a > b.

processor	Latency	Throughput
power8	4-9	2/cycle
power9	3	2/cycle

Parameters

a	128-bit vector treated as 2 x 64-bit unsigned long integer (dword) elements.
b	128-bit vector treated as 2 x 64-bit unsigned long integer (dword) elements.

Returns

boolean int for all 128-bits, true if all Greater Than, false otherwise.

7.11.6.31 vec_cmpud_all_le()

```
static int vec_cmpud_all_le (
            vui64_t a,
            vui64_t b )  [inline], [static]
```

Vector Compare all Less than equal Unsigned Doubleword.

Compare each unsigned long (64-bit) integer and return true if all elements of a <= b.

processor	Latency	Throughput
power8	4-9	2/cycle
power9	3	2/cycle

Parameters

a	128-bit vector treated as 2 x 64-bit unsigned long integer (dword) elements.
b	128-bit vector treated as 2 x 64-bit unsigned long integer (dword) elements.

Returns

boolean int for all 128-bits, true if all Greater Than, false otherwise.

7.11.6.32 vec_cmpud_all_lt()

```
static int vec_cmpud_all_lt (
            vui64_t a,
            vui64_t b )  [inline], [static]
```

Vector Compare all Less than Unsigned Doubleword.

Compare each unsigned long (64-bit) integer and return true if all elements of a $<$ b.

processor	Latency	Throughput
power8	4-9	2/cycle
power9	3	2/cycle

Parameters

a	128-bit vector treated as 2 x 64-bit unsigned long integer (dword) elements.
b	128-bit vector treated as 2 x 64-bit unsigned long integer (dword) elements.

Returns

boolean int for all 128-bits, true if all Greater Than, false otherwise.

7.11.6.33 vec_cmpud_all_ne()

```
static int vec_cmpud_all_ne (
            vui64_t a,
            vui64_t b )  [inline], [static]
```

Vector Compare all Not Equal Unsigned Doubleword.

Compare each unsigned long (64-bit) integer and return true if all elements of a and b are not equal.

processor	Latency	Throughput
power8	4-9	2/cycle
power9	3	2/cycle

Parameters

a	128-bit vector treated as 2 x 64-bit unsigned long integer (dword) elements.
b	128-bit vector treated as 2 x 64-bit unsigned long integer (dword) elements.

Returns

boolean int for all 128-bits, true if equal, false otherwise.

7.11.6.34 vec_cmpud_any_eq()

```
static int vec_cmpud_any_eq (
            vui64_t a,
            vui64_t b )  [inline], [static]
```

Vector Compare any Equal Unsigned Doubleword.

Compare each unsigned long (64-bit) integer and return true if any elements of a and b are equal.

processor	Latency	Throughput
power8	4-9	2/cycle
power9	3	2/cycle

Parameters

a	128-bit vector treated as 2 x 64-bit unsigned long integer (dword) elements.
b	128-bit vector treated as 2 x 64-bit unsigned long integer (dword) elements.

Returns

boolean int for all 128-bits, true if equal, false otherwise.

7.11.6.35 vec_cmpud_any_ge()

```
static int vec_cmpud_any_ge (
            vui64_t a,
            vui64_t b )  [inline], [static]
```

Vector Compare any Greater Than or Equal Unsigned Doubleword.

Compare each unsigned long (64-bit) integer and return true if any elements of a $>=$ b.

processor	Latency	Throughput
power8	4-9	2/cycle
power9	3	2/cycle

Parameters

a	128-bit vector treated as 2 x 64-bit unsigned long integer (dword) elements.
b	128-bit vector treated as 2 x 64-bit unsigned long integer (dword) elements.

Returns

boolean int for all 128-bits, true if all Greater Than, false otherwise.

7.11.6.36 vec_cmpud_any_gt()

```
static int vec_cmpud_any_gt (
             vui64_t a,
             vui64_t b )  [inline], [static]
```

Vector Compare any Greater Than Unsigned Doubleword.

Compare each unsigned long (64-bit) integer and return true if all elements of a $>$ b.

processor	Latency	Throughput
power8	4-9	2/cycle
power9	3	2/cycle

Parameters

a	128-bit vector treated as 2 x 64-bit unsigned long integer (dword) elements.
b	128-bit vector treated as 2 x 64-bit unsigned long integer (dword) elements.

Returns

boolean int for all 128-bits, true if all Greater Than, false otherwise.

7.11.6.37 vec_cmpud_any_le()

```
static int vec_cmpud_any_le (
             vui64_t a,
             vui64_t b )  [inline], [static]
```

Vector Compare any Less than equal Unsigned Doubleword.

Compare each unsigned long (64-bit) integer and return true if any elements of a <= b.

processor	Latency	Throughput
power8	4-9	2/cycle
power9	3	2/cycle

Parameters

a	128-bit vector treated as 2 x 64-bit unsigned long integer (dword) elements.
b	128-bit vector treated as 2 x 64-bit unsigned long integer (dword) elements.

Returns

boolean int for any 128-bits, true if any Greater Than, false otherwise.

7.11.6.38 vec_cmpud_any_lt()

```
static int vec_cmpud_any_lt (
            vui64_t a,
            vui64_t b )  [inline], [static]
```

Vector Compare any Less than Unsigned Doubleword.

Compare each unsigned long (64-bit) integer and return true if any elements of a < b.

processor	Latency	Throughput
power8	4-9	2/cycle
power9	3	2/cycle

Parameters

a	128-bit vector treated as 2 x 64-bit unsigned long integer (dword) elements.
b	128-bit vector treated as 2 x 64-bit unsigned long integer (dword) elements.

Returns

boolean int for any 128-bits, true if any Greater Than, false otherwise.

7.11.6.39 vec_cmpud_any_ne()

```
static int vec_cmpud_any_ne (
             vui64_t a,
             vui64_t b )  [inline], [static]
```

Vector Compare any Not Equal Unsigned Doubleword.

Compare each unsigned long (64-bit) integer and return true if any elements of a and b are not equal.

processor	Latency	Throughput
power8	4-9	2/cycle
power9	3	2/cycle

Parameters

a	128-bit vector treated as 2 x 64-bit unsigned long integer (dword) elements.
b	128-bit vector treated as 2 x 64-bit unsigned long integer (dword) elements.

Returns

boolean int for any 128-bits, true if equal, false otherwise.

7.11.6.40 vec_ctzd()

```
static vui64_t vec_ctzd (
             vui64_t vra )  [inline], [static]
```

Vector Count Trailing Zeros Doubleword for unsigned long long elements.

Count the number of trailing '0' bits (0-64) within each doubleword element of a 128-bit vector.

For POWER9 (PowerISA 3.0B) or later use the Vector Count Trailing Zeros Doubleword instruction **vctzd**. Otherwise use a sequence of pre ISA 3.0 VMX instructions leveraging the PVECLIB popcntd operation. SIMDized count Trailing zeros inspired by:

Warren, Henry S. Jr and *Hacker's Delight*, 2nd Edition, Addison Wesley, 2013. Chapter 5 Counting Bits, Section 5-4.

processor	Latency	Throughput
power8	8-10	2/2 cycles
power9	3	2/cycle

Parameters

vra	128-bit vector treated as 2 x 64-bit integer (doublewords) elements.

Returns

128-bit vector with the trailng zeros count for each doubleword element.

7.11.6.41 vec_maxsd()

```
static vi64_t vec_maxsd (
             vi64_t vra,
             vi64_t vrb )  [inline], [static]
```

Vector Maximum Signed Doubleword.

For each doubleword element [0|1] of vra and vrb compare as signed integers and return the larger value in the result.

processor	Latency	Throughput
power8	2	2/cycle
power9	3	2/cycle

Parameters

vra	128-bit vector long int.
vrb	128-bit vector long int.

Returns

vector long maximum of a and b.

7.11.6.42 vec_maxud()

```
static vui64_t vec_maxud (
             vui64_t vra,
             vui64_t vrb )  [inline], [static]
```

Vector Maximum Unsigned Doubleword.

For each doubleword element [0|1] of vra and vrb compare as unsigned integers and return the larger value in the result.

processor	Latency	Throughput
power8	2	2/cycle
power9	3	2/cycle

Parameters

vra	128-bit vector long int.
vrb	128-bit vector long int.

Returns

vector unsigned long maximum of a and b.

7.11.6.43 vec_minsd()

```
static vi64_t vec_minsd (
             vi64_t vra,
             vi64_t vrb )  [inline], [static]
```

Vector Minimum Signed Doubleword.

For each doubleword element [0|1] of vra and vrb compare as signed integers and return the smaller value in the result.

processor	Latency	Throughput
power8	2	2/cycle
power9	3	2/cycle

Parameters

vra	128-bit vector long int.
vrb	128-bit vector long int.

Returns

vector long minimum of a and b.

7.11.6.44 vec_minud()

```
static vui64_t vec_minud (
             vui64_t vra,
             vui64_t vrb )  [inline], [static]
```

Vector Minimum Unsigned Doubleword.

For each doubleword element [0|1] of vra and vrb compare as unsigned integers and return the smaller value in the result.

processor	Latency	Throughput
power8	2	2/cycle
power9	3	2/cycle

Parameters

vra	128-bit vector unsigned long int.
vrb	128-bit vector unsignedlong int.

Returns

vector unsigned long minimum of a and b.

7.11.6.45 vec_mrgahd()

```
static vui64_t vec_mrgahd (
            vui128_t vra,
            vui128_t vrb )  [inline], [static]
```

Vector Merge Algebraic High Doublewords.

Merge only the high doublewords from 2 x Algebraic quadwords across vectors vra and vrb. This is effectively the Vector Merge Even Doubleword operation that is not modified for endian.

For example, merge the high 64-bits from 2 x 128-bit products as generated by vec_muleud/vec_muloud. This result is effectively a vector multiply high unsigned doubleword.

processor	Latency	Throughput
power8	2	2/cycle
power9	2	2/cycle

Parameters

vra	128-bit vector unsigned __int128.
vrb	128-bit vector unsigned __int128.

Returns

A vector merge from only the high doublewords of the 2 x algebraic quadwords across vra and vrb.

7.11.6.46 vec_mrgald()

```
static vui64_t vec_mrgald (
            vui128_t vra,
            vui128_t vrb ) [inline], [static]
```

Vector Merge Algebraic Low Doublewords.

Merge only the low doublewords from 2 x Algebraic quadwords across vectors vra and vrb. This effectively the Vector Merge Odd doubleword operation that is not modified for endian.

For example, merge the low 64-bits from 2 x 128-bit products as generated by vec_muleud/vec_muloud. This result is effectively a vector multiply low unsigned doubleword.

processor	Latency	Throughput
power8	2	2/cycle
power9	2	2/cycle

Parameters

vra	128-bit vector unsigned __int128.
vrb	128-bit vector unsigned __int128.

Returns

A vector merge from only the low doublewords of the 2 x algebraic quadwords across vra and vrb.

7.11.6.47 vec_mrged()

```
static vui64_t vec_mrged (
            vui64_t __VA,
            vui64_t __VB ) [inline], [static]
```

Vector Merge Even Doubleword. Merge the even doubleword elements from two vectors into the high and low doubleword elements of the result. This is effectively the VSX Permute Doubleword Immediate operation modified for endian.

processor	Latency	Throughput
power8	2	2/cycle
power9	3	2/cycle

Parameters

__VA	a 128-bit vector as the source of the results even doubleword.
__VB	a 128-bit vector as the source of the results odd doubleword.

Returns

A vector merge from only the even doublewords of the 2 x quadwords across __VA and __VB.

7.11.6.48 vec_mrghd()

```
static vui64_t vec_mrghd (
            vui64_t __VA,
            vui64_t __VB )  [inline], [static]
```

Vector Merge High Doubleword. Merge the high doubleword elements from two vectors into the high and low doubleword elements of the result. This is effectively the VSX Permute Doubleword Immediate operation modified for endian.

processor	Latency	Throughput
power8	2	2/cycle
power9	3	2/cycle

Parameters

__VA	a 128-bit vector as the source of the results even doubleword.
__VB	a 128-bit vector as the source of the results odd doubleword.

Returns

A vector merge from only the high doublewords of the 2 x quadwords across __VA and __VB.

7.11.6.49 vec_mrgld()

```
static vui64_t vec_mrgld (
            vui64_t __VA,
            vui64_t __VB )  [inline], [static]
```

Vector Merge Low Doubleword. Merge the low doubleword elements from two vectors into the high and low doubleword elements of the result. This is effectively the VSX Permute Doubleword Immediate operation modified for endian.

processor	Latency	Throughput
power8	2	2/cycle
power9	3	2/cycle

Parameters

__VA	a 128-bit vector as the source of the results even doubleword.
__VB	a 128-bit vector as the source of the results odd doubleword.

Returns

A vector merge from only the low doublewords of the 2 x quadwords across __VA and __VB.

7.11.6.50 vec_mrgod()

```
static vui64_t vec_mrgod (
             vui64_t __VA,
             vui64_t __VB ) [inline], [static]
```

Vector Merge Odd Doubleword. Merge the odd doubleword elements from two vectors into the high and low doubleword elements of the result. This is effectively the VSX Permute Doubleword Immediate operation modified for endian.

processor	Latency	Throughput
power8	2	2/cycle
power9	3	2/cycle

Parameters

__VA	a 128-bit vector as the source of the results even doubleword.
__VB	a 128-bit vector as the source of the results odd doubleword.

Returns

A vector merge from only the odd doublewords of the 2 x quadwords across __VA and __VB.

7.11.6.51 vec_msumudm()

```
static vui128_t vec_msumudm (
             vui64_t a,
             vui64_t b,
             vui128_t c ) [inline], [static]
```

Vector Multiply-Sum Unsigned Doubleword Modulo.

Note

this implementation exists in vec_int128_ppc::h::vec_msumudm() as it requires vec_adduqm().

7.11.6.52 vec_muleud()

```
static vui128_t vec_muleud (
             vui64_t a,
             vui64_t b )  [inline], [static]
```

Vector Multiply Even Unsigned Doublewords.

Note

this implementation exists in vec_int128_ppc::h::vec_muleud() as it requires vec_vmuleud and vec_adduqm().

7.11.6.53 vec_mulhud()

```
static vui64_t vec_mulhud (
             vui64_t vra,
             vui64_t vrb )  [inline], [static]
```

Vector Multiply High Unsigned Doubleword.

Note

this implementation exists in vec_int128_ppc::h::vec_mulhud() as it requires vec_vmuleud() and vec_vmuloud().

7.11.6.54 vec_muloud()

```
static vui128_t vec_muloud (
             vui64_t a,
             vui64_t b )  [inline], [static]
```

Vector Multiply Odd Unsigned Doublewords.

Note

this implementation exists in vec_int128_ppc::h::vec_muloud() as it requires vec_vmuloud() and vec_adduqm().

7.11.6.55 vec_muludm()

```
static vui64_t vec_muludm (
            vui64_t vra,
            vui64_t vrb )  [inline], [static]
```

Vector Multiply Unsigned Doubleword Modulo.

Note

this implementation exists in vec_int128_ppc::h::vec_muludm() as it requires vec_vmuleud() and vec_vmuloud().

7.11.6.56 vec_pasted()

```
static vui64_t vec_pasted (
            vui64_t __VH,
            vui64_t __VL )  [inline], [static]
```

Vector doubleword paste. Concatenate the high doubleword of the 1st vector with the low double word of the 2nd vector.

processor	Latency	Throughput
power8	2	2/cycle
power9	3	2/cycle

Parameters

__VH	a 128-bit vector as the source of the high order doubleword.
__VL	a 128-bit vector as the source of the low order doubleword.

Returns

The combined 128-bit vector composed of the high order doubleword of __VH and the low order doubleword of __VL.

7.11.6.57 vec_permdi()

```
static vui64_t vec_permdi (
            vui64_t vra,
            vui64_t vrb,
            const int ctl )  [inline], [static]
```

Vector Permute Doubleword Immediate. Combine a doubleword selected from the 1st (vra) vector with a doubleword selected from the 2nd (vrb) vector.

Note

> This function implements the operation of a VSX Permute Doubleword Immediate instruction. This implementation is NOT Endian sensitive and the function is stable across BE/LE implementations.

The 2-bit control operand (ctl) selects which doubleword from the 1st and 2nd vector operands are transfered to the result vector. Control table:

ctl	vrt[0:63]	vrt[64:127]
0	vra[0:63]	vrb[0:63]
1	vra[0:63]	vrb[64:127]
2	vra[64:127]	vrb[0:63]
3	vra[64:127]	vrb[64:127]

processor	Latency	Throughput
power8	2	2/cycle
power9	3	2/cycle

Parameters

vra	a 128-bit vector as the source of the high order doubleword of the result.
vrb	a 128-bit vector as the source of the low order doubleword of the result.
ctl	const integer where the low order 2 bits control the selection of doublewords from input vector vra and vrb.

Returns

> The combined 128-bit vector composed of the high order doubleword of vra and the low order doubleword of vrb.

7.11.6.58 vec_popcntd()

```
static vui64_t vec_popcntd (
            vui64_t vra )  [inline], [static]
```

Vector Population Count doubleword.

Count the number of '1' bits (0-64) within each doubleword element of a 128-bit vector.

processor	Latency	Throughput
power8	4	2/2 cycles
power9	3	2/cycle

For POWER8 (PowerISA 2.07B) or later use the Vector Population Count DoubleWord (**vpopcntd**) instruction. Otherwise use the pveclib vec_popcntw to count each word then sum across with Vector Sum across Half Signed Word

Saturate (**vsum2sws**).

Parameters

vra	128-bit vector treated as 2 x 64-bit integer (dwords) elements.

Returns

128-bit vector with the population count for each dword element.

7.11.6.59 vec_revbd()

```
static vui64_t vec_revbd (
            vui64_t vra )  [inline], [static]
```

byte reverse each doubleword for a vector unsigned long int.

For each doubleword of the input vector, reverse the order of bytes / octets within the doubleword.

processor	Latency	Throughput
power8	2-11	2/cycle
power9	3	2/cycle

Parameters

vra	a 128-bit vector unsigned long int.

Returns

a 128-bit vector with the bytes of each doubleword reversed.

7.11.6.60 vec_rldi()

```
static vui64_t vec_rldi (
            vui64_t vra,
            const unsigned int shb )  [inline], [static]
```

Vector Rotate left Doubleword Immediate.

Rotate left each doubleword element [0-1], 0-63 bits, as specified by an immediate value. The rotate amount is a const unsigned int in the range 0-63. A rotate count of 0 returns the original value of vra. Shift counts greater then 63 bits handled modulo 64.

processor	Latency	Throughput
power8	2-4	2/cycle
power9	2-5	2/cycle

Parameters

vra	a 128-bit vector treated as a vector unsigned long int.
shb	rotate amount in the range 0-63.

Returns

128-bit vector unsigned long int, shifted left shb bits.

7.11.6.61 vec_selsd()

```
static vi64_t vec_selsd (
            vi64_t vra,
            vi64_t vrb,
            vb64_t vrc )  [inline], [static]
```

Vector Select Signed Doubleword.

Return the value, (vra & ∼vrc) | (vrb & vrc).

processor	Latency	Throughput
power8	2	2/cycle
power9	3	2/cycle

Parameters

vra	a 128-bit vector treated as a vector signed long long int.
vrb	a 128-bit vector treated as a vector signed long long int.
vrc	a 128-bit vector treated as vector bool long long int.

Returns

The selected bits from vra and vrb

7.11.6.62 vec_selud()

```
static vui64_t vec_selud (
             vui64_t vra,
             vui64_t vrb,
             vb64_t vrc )  [inline], [static]
```

Vector Select Unsigned Doubleword.

Return the value, (vra & ∼vrc) | (vrb & vrc).

processor	Latency	Throughput
power8	2	2/cycle
power9	3	2/cycle

Parameters

vra	a 128-bit vector treated as a vector unsigned long long int.
vrb	a 128-bit vector treated as a vector unsigned long long int.
vrc	a 128-bit vector treated as vector bool long long int.

Returns

The selected bits from vra and vrb

7.11.6.63 vec_setb_sd()

```
static vb64_t vec_setb_sd (
             vi64_t vra )  [inline], [static]
```

Vector Set Bool from Signed Doubleword.

For each doubleword, propagate the sign bit to all 64-bits of that doubleword. The result is vector bool long long reflecting the sign bit of each 64-bit doubleword.

processor	Latency	Throughput
power8	2-4	2/cycle
power9	2-5	2/cycle

Parameters

vra	Vector signed long long.

Returns

vector bool long long reflecting the sign bits of each doubleword.

7.11.6.64 vec_sldi()

```
static vui64_t vec_sldi (
            vui64_t vra,
            const unsigned int shb )  [inline], [static]
```

Vector Shift left Doubleword Immediate.

Shift left each doubleword element [0-1], 0-63 bits, as specified by an immediate value. The shift amount is a const unsigned long int in the range 0-63. A shift count of 0 returns the original value of vra. Shift counts greater then 63 bits return zero.

processor	Latency	Throughput
power8	2-4	2/cycle
power9	2-5	2/cycle

Parameters

vra	a 128-bit vector treated as a vector unsigned long int.
shb	shift amount in the range 0-63.

Returns

128-bit vector unsigned long int, shifted left shb bits.

7.11.6.65 vec_splat_s64()

```
static vi64_t vec_splat_s64 (
            const int sim )  [inline], [static]
```

Vector Splat Immediate Signed Doubleword. Duplicate the signed integer constant across doubleword elements of the result. This is the doubleword equivalent Vector Splat Immediate Signed (Byte | Halfword |Word).

Note

POWER9/10 will generate the 2 instruction sequence xxspltib/vextsb2d for values -128 to 128. Larger values will be loaded as a quadword constant from the read-only data (.rodata) section. POWER8 (and earlier) does not have vextsb2d instructions. For a smaller range (-16 -> 15) POWER8 can use the sequence vec_splat_s32/vec↩ _unpackl but the latest compilers are too clever for this and generate a load from .rodata anyway.

processor	Latency	Throughput
power8	4 - 9	2/cycle
power9	5	2/cycle

Parameters

sim	a small signed integer const.

Returns

Vector with sim value splatted to doublewords.

7.11.6.66 vec_splat_u64()

```
static vui64_t vec_splat_u64 (
            const int sim )  [inline], [static]
```

Vector Splat Immediate Unsigned Doubleword. Duplicate the unsigned integer constant across doubleword elements of the result. This is the doubleword equivalent Vector Splat Immediate Unsigned (Byte | Halfword |Word).

Note

POWER9/10 will generate the 2 instruction sequence xxspltib/vextsb2d for values -128 to 128. Larger values will be loaded as a quadword constant from the read-only data (.rodata) section. POWER8 (and earlier) does not have vextsb2d instructions. For a smaller range (-16 -> 15) POWER8 can use the sequence vec_splat_s32/vec_unpackl but the latest compilers are too clever for this and generate a load from .rodata anyway.

processor	Latency	Throughput
power8	4 - 9	2/cycle
power9	5	2/cycle

Parameters

sim	a small signed integer const.

Returns

Vector with sim value splatted to doublewords.

7.11.6.67 vec_splatd()

```
static vui64_t vec_splatd (
             vui64_t vra,
             const int ctl )  [inline], [static]
```

Vector splat doubleword. Duplicate the selected doubleword element across the doubleword elements of the result. This is effectively the VSX Merge doubleword operation modified for endian.

processor	Latency	Throughput
power8	2	2/cycle
power9	3	2/cycle

The 1-bit control operand (ctl) selects which (0:1) doubleword element, from the vector operand, is replicated to both doublewords of the result vector. Control table:

ctl	vrt[0]	vrt[1]
0	vra[0]	vra[0]
1	vra[1]	vra[1]

Parameters

vra	a 128-bit vector.
ctl	a const integer encoding the source doubleword.

Returns

The original vector with the doubleword elements swapped.

7.11.6.68 vec_spltd()

```
static vui64_t vec_spltd (
             vui64_t vra,
             const int ctl )  [inline], [static]
```

Deprecated Vector splat doubleword. Duplicate the selected doubleword element across the doubleword elements of the result.

processor	Latency	Throughput
power8	2	2/cycle
power9	3	2/cycle

The 1-bit control operand (ctl) selects which (0:1) doubleword element, from the vector operand, is replicated to both doublewords of the result vector. Control table:

ctl	vrt[0:63]	vrt[64:127]
0	vra[0:63]	vra[0:63]
1	vra[64:127]	vra[64:127]

Parameters

vra	a 128-bit vector.
ctl	a const integer encoding the source doubleword.

Returns

The original vector with the doubleword elements swapped.

7.11.6.69 vec_sradi()

```
static vi64_t vec_sradi (
             vi64_t vra,
             const unsigned int shb )  [inline], [static]
```

Vector Shift Right Algebraic Doubleword Immediate.

Shift Right Algebraic each doubleword element [0-1], 0-63 bits, as specified by an immediate value. The shift amount is a const unsigned int in the range 0-63. A shift count of 0 returns the original value of vra. Shift counts greater then 63 bits return the sign bit propagated to each bit of each element.

processor	Latency	Throughput
power8	2-4	2/cycle
power9	2-5	2/cycle

Parameters

vra	a 128-bit vector treated as a vector signed long int.
shb	shift amount in the range 0-63.

Returns

128-bit vector signed long int, shifted right shb bits.

7.11.6.70 vec_srdi()

```
static vui64_t vec_srdi (
            vui64_t vra,
            const unsigned int shb )  [inline], [static]
```

Vector Shift Right Doubleword Immediate.

Shift Right each doubleword element [0-1], 0-63 bits, as specified by an immediate value. The shift amount is a const unsigned int in the range 0-63. A shift count of 0 returns the original value of vra. Shift counts greater then 63 bits return zero.

processor	Latency	Throughput
power8	2-4	2/cycle
power9	2-5	2/cycle

Parameters

vra	a 128-bit vector treated as a vector unsigned long int.
shb	shift amount in the range 0-63.

Returns

128-bit vector unsigned long int, shifted right shb bits.

7.11.6.71 vec_subudm()

```
static vui64_t vec_subudm (
            vui64_t a,
            vui64_t b )  [inline], [static]
```

Vector Subtract Unsigned Doubleword Modulo.

For each unsigned long (64-bit) integer element c[i] = a[i] + NOT(b[i]) + 1.

processor	Latency	Throughput
power8	2	2/cycle
power9	2	2/cycle

For POWER8 (PowerISA 2.07B) or later use the Vector Subtract Unsigned Doubleword Modulo (**vsubudm**) instruction. Otherwise use vector add word modulo forms and propagate the carry bits.

Parameters

a	128-bit vector treated as 2 X unsigned long int.
b	128-bit vector treated as 2 X unsigned long int.

Returns

vector unsigned long int sum of a[0] + NOT(b[0]) + 1 and a[1] + NOT(b[1]) + 1.

7.11.6.72 vec_swapd()

```
static vui64_t vec_swapd (
             vui64_t vra )  [inline], [static]
```

Vector doubleword swap. Exchange the high and low doubleword elements of a vector.

processor	Latency	Throughput
power8	2	2/cycle
power9	3	2/cycle

Parameters

vra	a 128-bit vector.

Returns

The original vector with the doubleword elements swapped.

7.11.6.73 vec_vgluddo()

```
static vui64_t vec_vgluddo (
             unsigned long long * array,
             vi64_t vra )  [inline], [static]
```

Vector Gather-Load Integer Doublewords from Vector Doubleword Offsets.

For each doubleword element [i] of vra, load the doubleword element at ∗(char∗)array+vra[i]. Merge those doubleword elements and return the resulting vector. For best performance **&array** and doubleword offsets **vra** should be doubleword aligned (integer multiple of 8).

Note

As effective address calculation is modulo 64-bits, signed or unsigned doubleword offsets are equivalent.

processor	Latency	Throughput
power8	12	1/cycle
power9	11	1/cycle

Parameters

array	Pointer to array of integer doublewords.
vra	Vector of doubleword (64-bit) byte offsets from &array.

Returns

vector doubleword containing elements loaded from *(char*)array+vra[0] and *(char*)array+vra[1].

7.11.6.74 vec_vgluddsx()

```
static vui64_t vec_vgluddsx (
            unsigned long long * array,
            vi64_t vra,
            const unsigned char scale )  [inline], [static]
```

Vector Gather-Load Integer Doublewords from Vector Doubleword Scaled Indexes.

For each doubleword element [i] of vra, load the doubleword element array[vra[i] * (1 << scale)]. Merge those doubleword elements and return the resulting vector. Array element indices are converted to byte offsets from (array) by multiplying each index by (sizeof (array element) * scale), which is effected by shifting left (3+scale) bits.

Note

As effective address calculation is modulo 64-bits, signed or unsigned doubleword indexes are equivalent.

processor	Latency	Throughput
power8	14-23	1/cycle
power9	13-22	1/cycle

Parameters

array	Pointer to array of integer doublewords.
vra	Vector of signed doubleword indexes.
scale	8-bit integer. Indexes are multiplying by 2^{scale}.

Returns

vector containing doublewords from array[(vra[0,1]<<scale)].

7.11.6.75 vec_vgluddx()

```
static vui64_t vec_vgluddx (
            unsigned long long * array,
            vi64_t vra ) [inline], [static]
```

Vector Gather-Load Integer Doublewords from Vector Doubleword Indexes.

For each doubleword element [i] of vra, load the doubleword element from array[vra[i]]. Merge those doubleword elements and return the resulting vector. Array element indices are converted to byte offsets from (array) by multiplying each index by (sizeof (array element) ∗ scale), which is effected by shifting left 3 bits.

Note

As effective address calculation is modulo 64-bits, signed or unsigned doubleword indexes are equivalent.

processor	Latency	Throughput
power8	14-23	1/cycle
power9	13-22	1/cycle

Parameters

array	Pointer to array of integer doublewords.
vra	Vector of signed doubleword indexes.

Returns

vector containing doublewords array[vra[0,1]].

7.11.6.76 vec_vgludso()

```
static vui64_t vec_vgludso (
            unsigned long long * array,
            const long long offset0,
            const long long offset1 ) [inline], [static]
```

Vector Gather-Load Integer Doublewords from Scalar Offsets.

For each scalar offset[0|1], load the doubleword element at ∗(char∗)array+offset[0|1]. Merge those doubleword elements and return the resulting vector. For best performance **&array** and doubleword offsets should be doubleword aligned (integer multiple of 8).

processor	Latency	Throughput
power8	7	1/cycle
power9	8	1/cycle

Parameters

array	Pointer to array of integer doublewords.
offset0	Scalar (64-bit) byte offsets from &array.
offset1	Scalar (64-bit) byte offsets from &array.

Returns

vector doubleword containing elements loaded from ∗(char∗)array+offset0 and ∗(char∗)array+offset1.

7.11.6.77 vec_vlsidx()

```
static vui64_t vec_vlsidx (
            const signed long long ra,
            const unsigned long long * rb )  [inline], [static]
```

Vector Load Scalar Integer Doubleword Indexed.

Load the left most doubleword of vector **xt** as a scalar doubleword from the effective address formed by **rb+ra**. The operand **rb** is a pointer to an array of doublewords. The operand **ra** is a doubleword integer byte offset from **rb**. The result **xt** is returned as a vui64_t vector. For best performance **rb** and **ra** should be doubleword aligned (integer multiple of 8).

Note

the right most doubleword of vector **xt** is left *undefined* by this operation.

This operation is an alternate form of Vector Load Element (vec_lde), with the added simplification that data is always left justified in the vector. This simplifies merging elements for gather operations.

Note

This instruction was introduced in PowerISA 2.06 (POWER7). For POWER8/9 there are additional optimizations by effectively converting small constant index values into displacements. For POWER8 a specific pattern of addi/lsxdx instruction is *fused* into a single load displacement internal operation. For POWER9 we can use the lxsd (DS-form) instruction directly.

processor	Latency	Throughput
power8	5	2/cycle
power9	5	2/cycle

Parameters

ra	const signed doubleword index (offset/displacement).
rb	const doubleword pointer to an array of doubles.

Returns

The data stored at (ra + rb) is loaded into vector doubleword element 0. Element 1 is undefined.

7.11.6.78 vec_vmadd2eud()

```
static vui128_t vec_vmadd2eud (
             vui64_t a,
             vui64_t b,
             vui64_t c,
             vui64_t d )  [inline], [static]
```

Vector Multiply-Add2 Even Unsigned Doublewords.

Note

this implementation exists in vec_int128_ppc::h::vec_vmadd2eud() as it requires vec_msumudm() and vec_adduqm().

7.11.6.79 vec_vmadd2euw()

```
static vui64_t vec_vmadd2euw (
             vui32_t a,
             vui32_t b,
             vui32_t c,
             vui32_t d )  [inline], [static]
```

Vector Multiply-Add2 Even Unsigned Words.

Multiply the even 32-bit Words of vector unsigned int values $(a * b)$ and return sums of the unsigned 64-bit product and the even 32-bit words of c and d $(a_{even} * b_{even}) + EXTZ(c_{even} + EXTZ(d_{even})$.

Note

The advantage of this form (versus Multiply-Sum) is that the final 64 bit sums can not overflow.

This implementation is NOT endian sensitive and the function is stable across BE/LE implementations.

processor	Latency	Throughput
power8	9	1/cycle
power9	9	1/cycle

Parameters

a	128-bit vector unsigned int.
b	128-bit vector unsigned int.
c	128-bit vector unsigned int.
d	128-bit vector unsigned int.

Returns

vector unsigned long int sum $(a_{\text{even}} * b_{\text{even}}) + \text{EXTZ}(c_{\text{even}}) + \text{EXTZ}(d_{\text{even}})$.

7.11.6.80 vec_vmadd2oud()

```
static vui128_t vec_vmadd2oud (
            vui64_t a,
            vui64_t b,
            vui64_t c,
            vui64_t d )  [inline], [static]
```

Vector Multiply-Add2 Odd Unsigned Doublewords.

Note

this implementation exists in vec_int128_ppc::h::vec_vmadd2oud() as it requires vec_msumudm() and vec_adduqm().

7.11.6.81 vec_vmadd2ouw()

```
static vui64_t vec_vmadd2ouw (
            vui32_t a,
            vui32_t b,
            vui32_t c,
            vui32_t d )  [inline], [static]
```

Vector Multiply-Add2 Odd Unsigned Words.

Multiply the odd 32-bit Words of vector unsigned int values $(a * b)$ and return sums of the unsigned 64-bit product and the odd 32-bit words of c and d $(a_{\text{odd}} * b_{\text{odd}}) + \text{EXTZ}(c_{\text{odd}} + \text{EXTZ}(d_{\text{odd}})$.

Note

The advantage of this form (versus Multiply-Sum) is that the final 64 bit sums can not overflow.

This implementation is NOT endian sensitive and the function is stable across BE/LE implementations.

processor	Latency	Throughput
power8	9	1/cycle
power9	9	1/cycle

Parameters

a	128-bit vector unsigned int.
b	128-bit vector unsigned int.
c	128-bit vector unsigned int.
d	128-bit vector unsigned int.

Returns

vector unsigned long int sum ($a_{odd} * b_{odd}$) + EXTZ(c_{odd} + EXTZ(d_{odd}).

7.11.6.82 vec_vmaddeud()

```
static vui128_t vec_vmaddeud (
            vui64_t a,
            vui64_t b,
            vui64_t c ) [inline], [static]
```

Vector Multiply-Add Even Unsigned Doublewords.

Note

this implementation exists in vec_int128_ppc::h::vec_vmaddeud() as it requires vec_msumudm() and vec_adduqm().

7.11.6.83 vec_vmaddeuw()

```
static vui64_t vec_vmaddeuw (
            vui32_t a,
            vui32_t b,
            vui32_t c ) [inline], [static]
```

Vector Multiply-Add Even Unsigned Words.

Multiply the even 32-bit Words of vector unsigned int values ($a * b$) and return sums of the unsigned 64-bit product and the even 32-bit words of c ($a_{even} * b_{even}$) + EXTZ(c_{even}).

Note

The advantage of this form (versus Multiply-Sum) is that the final 64 bit sums can not overflow.

This implementation is NOT endian sensitive and the function is stable across BE/LE implementations.

processor	Latency	Throughput
power8	9	2/cycle
power9	9	2/cycle

Parameters

a	128-bit vector unsigned int.
b	128-bit vector unsigned int.
c	128-bit vector unsigned int.

Returns

vector unsigned long int sum ($a_{even} * b_{even}$) + EXTZ(c_{even}).

7.11.6.84 vec_vmaddoud()

```
static vui128_t vec_vmaddoud (
             vui64_t a,
             vui64_t b,
             vui64_t c )  [inline], [static]
```

Vector Multiply-Add Odd Unsigned Doublewords.

Note

this implementation exists in vec_int128_ppc::h::vec_vmaddoud() as it requires vec_msumudm() and vec_adduqm().

7.11.6.85 vec_vmaddouw()

```
static vui64_t vec_vmaddouw (
             vui32_t a,
             vui32_t b,
             vui32_t c )  [inline], [static]
```

Vector Multiply-Add Odd Unsigned Words.

Multiply the odd 32-bit Words of vector unsigned int values ($a * b$) and return sums of the unsigned 64-bit product and the odd 32-bit words of c ($a_{odd} * b_{odd}$) + EXTZ(c_{odd}).

Note

The advantage of this form (versus Multiply-Sum) is that the final 64 bit sums can not overflow.

This implementation is NOT endian sensitive and the function is stable across BE/LE implementations.

processor	Latency	Throughput
power8	9	2/cycle
power9	9	2/cycle

Parameters

a	128-bit vector unsigned int.
b	128-bit vector unsigned int.
c	128-bit vector unsigned int.

Returns

vector unsigned long int sum ($a_{odd} * b_{odd}$) + EXTZ(c_{odd}).

7.11.6.86 vec_vmsumeud()

```
static vui128_t vec_vmsumeud (
            vui64_t a,
            vui64_t b,
            vui128_t c )  [inline], [static]
```

Vector Multiply-Sum Even Unsigned Doublewords.

Note

this implementation exists in vec_int128_ppc::h::vec_vmsumeud() as it requires vec_msumudm() and vec_adduqm().

7.11.6.87 vec_vmsumoud()

```
static vui128_t vec_vmsumoud (
            vui64_t a,
            vui64_t b,
            vui128_t c )  [inline], [static]
```

Vector Multiply-Sum Odd Unsigned Doublewords.

Note

this implementation exists in vec_int128_ppc::h::vec_vmsumoud() as it requires vec_msumudm() and vec_adduqm().

7.11.6.88 vec_vmsumuwm()

```
static vui64_t vec_vmsumuwm (
             vui32_t vra,
             vui32_t vrb,
             vui64_t vrc )  [inline], [static]
```

Vector Multiply-Sum Unsigned Word Modulo.

Multiply the unsigned word elements of vra and vrb, internally generating doubleword products. Then generate three-way sum of adjacent doubleword product pairs, plus the doubleword elements from vrc. The final summation is modulo 64-bits.

Note

This function implements the operation of a Vector Multiply-Sum Unsigned Word Modulo instruction, if the Power↩ ISA included such an instruction. This implementation is NOT endian sensitive and the function is stable across BE/LE implementations.

processor	Latency	Throughput
power8	11	1/cycle
power9	11	1/cycle

Parameters

vra	128-bit vector unsigned int.
vrb	128-bit vector unsigned int.
vrc	128-bit vector unsigned long.

Returns

vector of doubleword elements where each is the sum of the even and odd adjacent products of the vra and vrb, plus the corresponding doubleword element of vrc.

7.11.6.89 vec_vmuleud()

```
static vui128_t vec_vmuleud (
             vui64_t a,
             vui64_t b )  [inline], [static]
```

Vector Multiply Even Unsigned Doublewords.

Note

this implementation exists in vec_int128_ppc::h::vec_vmuleud() as it requires vec_msumudm() and vec_adduqm().

7.11.6.90 vec_vmuloud()

```
static vui128_t vec_vmuloud (
            vui64_t a,
            vui64_t b )  [inline], [static]
```

Vector Multiply Odd Unsigned Doublewords.

Note

this implementation exists in vec_int128_ppc::h::vec_vmuloud() as it requires vec_msumudm() and vec_adduqm().

7.11.6.91 vec_vpkudum()

```
static vui32_t vec_vpkudum (
            vui64_t vra,
            vui64_t vrb )  [inline], [static]
```

Vector Pack Unsigned Doubleword Unsigned Modulo.

The doubleword source is the concatination of vra and vrb. For each integer word from 0 to 3, of the result vector, do the following: place the contents of bits 32:63 of the corresponding doubleword source element [i] into word element [i] of the result.

processor	Latency	Throughput
power8	2	2/cycle
power9	2	2/cycle

Note

Use vec_vpkudum naming but only if the compiler does not define it in <altivec.h>.

Parameters

vra	a 128-bit vector treated as 2 x unsigned long integers.
vrb	a 128-bit vector treated as 2 x unsigned long integers.

Returns

128-bit vector treated as 4 x unsigned integers.

7.11.6.92 vec_vrld()

```
static vui64_t vec_vrld (
            vui64_t vra,
            vui64_t vrb )  [inline], [static]
```

Vector Rotate Left Doubleword.

Vector Rotate Left Doubleword 0-63 bits. The shift amount is from bits 58-63 and 122-127 of vrb.

processor	Latency	Throughput
power8	2	2/cycle
power9	2	2/cycle

Note

Use vec_vrld naming but only if the compiler does not define it in <altivec.h>.

Parameters

vra	a 128-bit vector treated as 2 x unsigned long integers.
vrb	shift amount in bits 58:63 and 122:127.

Returns

Left shifted vector unsigned long.

7.11.6.93 vec_vsld()

```
static vui64_t vec_vsld (
            vui64_t vra,
            vui64_t vrb )  [inline], [static]
```

Vector Shift Left Doubleword.

Vector Shift Left Doubleword 0-63 bits. The shift amount is from bits 58-63 and 122-127 of vrb.

processor	Latency	Throughput
power8	2	2/cycle
power9	2	2/cycle

Note

Can not use vec_sld naming here as that would conflict with the generic Shift Left Double Vector. Use vec_vsld but only if the compiler does not define it in <altivec.h>.

Parameters

vra	a 128-bit vector treated as 2 x unsigned long integers.
vrb	shift amount in bits 58:63 and 122:127.

Returns

Left shifted vector unsigned long.

7.11.6.94 vec_vsrad()

```
static vi64_t vec_vsrad (
            vi64_t vra,
            vui64_t vrb )  [inline], [static]
```

Vector Shift Right Algebraic Doubleword.

Vector Shift Right Algebraic Doubleword 0-63 bits. The shift amount is from bits 58-63 and 122-127 of vrb.

processor	Latency	Throughput
power8	2	2/cycle
power9	2	2/cycle

Note

Use the vec_vsrad for consistency with vec_vsld above. Define vec_vsrad only if the compiler does not define it in <altivec.h>.

Parameters

vra	a 128-bit vector treated as 2 x unsigned long integers.
vrb	shift amount in bits 58:63 and 122:127.

Returns

Right shifted vector unsigned long.

7.11.6.95 vec_vsrd()

```
static vui64_t vec_vsrd (
            vui64_t vra,
            vui64_t vrb )  [inline], [static]
```

Vector Shift Right Doubleword.

Vector Shift Right Doubleword 0-63 bits. The shift amount is from bits 58-63 and 122-127 of vrb.

processor	Latency	Throughput
power8	2	2/cycle
power9	2	2/cycle

Note

Use the vec_vsrd for consistency with vec_vsld above. Define vec_vsrd only if the compiler does not define it in <altivec.h>.

Parameters

vra	a 128-bit vector treated as 2 x unsigned long integers.
vrb	shift amount in bits 58:63 and 122:127.

Returns

Right shifted vector unsigned long.

7.11.6.96 vec_vsstuddo()

```
static void vec_vsstuddo (
            vui64_t xs,
            unsigned long long * array,
            vi64_t vra )  [inline], [static]
```

Vector Scatter-Store Integer Doublewords to Vector Doublewords Offsets.

For each doubleword element [i] of vra, Store the doubleword element xs[i] at the address ∗(char∗)array+vra[i] For best performance **&array** and doubleword offsets **vra** should be doubleword aligned (integer multiple of 8).

processor	Latency	Throughput
power8	12	1/cycle
power9	8	1/cycle

Parameters

xs	Vector of integer doubleword elements to scatter store.
array	Pointer to array of integer doublewords.
vra	Vector of doubleword (64-bit) byte offsets from &array.

7.11.6.97 vec_vsstuddsx()

```
static void vec_vsstuddsx (
            vui64_t xs,
            unsigned long long * array,
            vi64_t vra,
            const unsigned char scale )  [inline], [static]
```

Vector Scatter-Store Integer Doublewords to Vector Doubleword Scaled Indexes.

For each doubleword element [i] of vra, store the doubleword element xs[i] at array[(vra[i] << scale)]. Array element indices are converted to byte offsets from (array) by multiplying each index by (sizeof (array element) ∗ scale), which is effected by shifting left (3+scale) bits.

processor	Latency	Throughput
power8	14-23	1/cycle
power9	10-19	1/cycle

Parameters

xs	Vector of integer doubleword elements to scatter store.
array	Pointer to array of integer doublewords.
vra	Vector of signed doubleword indexes.
scale	8-bit integer. Indexes are multiplying by 2^{scale}.

7.11.6.98 vec_vsstuddx()

```
static void vec_vsstuddx (
            vui64_t xs,
            unsigned long long * array,
            vi64_t vra )  [inline], [static]
```

Vector Scatter-Store Integer Doublewords to Vector Doubleword Indexes.

For each doubleword element [i] of vra, store the doubleword element xs[i] at array[vra[i]]. Indexes are converted to offsets from ∗array by shifting each doubleword of vra left (3+scale) bits.

processor	Latency	Throughput
power8	14-23	1/cycle
power9	10-19	1/cycle

Parameters

xs	Vector of integer doubleword elements to scatter store.
array	Pointer to array of integer doublewords.
vra	Vector of signed doubleword indexes.

7.11.6.99 vec_vsstudso()

```
static void vec_vsstudso (
             vui64_t xs,
             unsigned long long * array,
             const long long offset0,
             const long long offset1 )  [inline], [static]
```

Vector Scatter-Store Integer Doublewords to Scalar Offsets.

For each doubleword element [i] of vra, Store the doubleword element xs[i] at ∗(char∗)array+offset[0|1]. For best performance, **&array** and doubleword offsets should be doubleword aligned (integer multiple of 8).

processor	Latency	Throughput
power8	12	1/cycle
power9	8	1/cycle

Parameters

xs	Vector of integer doubleword elements to scatter store.
array	Pointer to array of integer doublewords.
offset0	Scalar (64-bit) byte offset from &array.
offset1	Scalar (64-bit) byte offset from &array.

7.11.6.100 vec_vstsidx()

```
static void vec_vstsidx (
             vui64_t xs,
             const signed long long ra,
             unsigned long long * rb )  [inline], [static]
```

Vector Store Scalar Integer Doubleword Indexed.

Stores the left most doubleword of vector **xs** as a scalar doubleword at the effective address formed by **rb+ra**. The operand **rb** is a pointer to an array of doublewords. The operand **ra** is a doubleword integer byte offset from **rb**. For best performance **rb** and **ra** should be doubleword aligned (integer multiple of 8).

This operation is an alternate form of vector store element, with the added simplification that data is always left justified in the vector. This simplifies scatter operations.

Note

> This is instruction was introduced in PowerISA 2.06 (POWER7). For POWER9 there are additional optimizations by effectively converting small constant index values into displacements. For POWER9 we can use the stxsd (DS-form) instruction directly.

processor	Latency	Throughput
power8	0 - 2	2/cycle
power9	0 - 2	4/cycle

Parameters

xs	vector doubleword element 0 to be stored.
ra	const signed long long index (offset/displacement).
rb	const doubleword pointer to an array of doubles.

7.11.6.101 vec_xxspltd()

```
static vui64_t vec_xxspltd (
            vui64_t vra,
            const int ctl )  [inline], [static]
```

Vector splat doubleword. Duplicate the selected doubleword element across the doubleword elements of the result.

Note

> This function implements the operation of a VSX Splat Doubleword Immediate instruction. This implementation is NOT Endian sensitive and the function is stable across BE/LE implementations.

The 1-bit control operand (ctl) selects which (0:1) doubleword element, from the vector operand, is replicated to both doublewords of the result vector. Control table:

ctl	vrt[0:63]	vrt[64:127]
0	vra[0:63]	vra[0:63]
1	vra[64:127]	vra[64:127]

processor	Latency	Throughput
power8	2	2/cycle
power9	3	2/cycle

Parameters

vra	a 128-bit vector.
ctl	a const integer encoding the source doubleword.

Returns

The original vector with the doubleword elements swapped.

Index