POWER Vector Library Manual

1.0.0

Generated by Doxygen 1.8.11

Contents

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

1.1 Notices

1.1.1 Reference Documentation

- Power Instruction Set Architecture, Versions 2.06B, 2.07B and 3.0B, IBM, 2010-2017. http://www.power.org and more recently from https://www-355.ibm.com/systems/power/openpower/
- ALTIVEC PIM: AltiVecTM Technology Programming Interface Manual, Freescale Semiconductor, 1999. http://www.freescale.com/files/32bit/doc/ref_manual/ALTIVECPIM.pdf
- 64-bit PowerPC ELF Application Binary Interface Supplement 1.9. http://refspecs.linuxfoundation.org/ELF/ppc64/PPC-elf64abi.html
- OpenPOWER ELF V2 application binary interface (ABI), OpenPOWER Foundation, 2017. https://openpowerfoundation.org/?resource_lib=64-bit-elf-v2-abi-specification-power-archi
- Using the GNU Compiler Collection (GCC), Free Software Foundation, 1988-2018. https://gcc.gnu.org/onlinedocs/
- POWER8 Processor User's Manual for the Single-Chip Module https://ibm.ent.box.com/s/649rlau0zjcc0yrulqf4cgx5wk3pgbfk
- POWER9 Processor User's Manual https://ibm.ent.box.com/s/8uj02ysel62meji4voujw29wwkhsz6a4
- Warren, Henry S. Jr, Hacker's Delight, 2nd Edition, Upper Saddle River, NJ: Addison Wesley, 2013.

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) did add 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.

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 and sizes.

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). 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. As another example full __int128 implementations of Count Leading Zeros, Population Count, Shift left/right immediate, and integer divide.
- Such implementations should be small enough to inline and allow the compiler opportunity to apply common optimization techniques.

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_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.

Another issue is that generic operations were not uniformly applicable across vector types. For example:

- 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)
- Integer long (64-bit) and __int128 support for POWER8 (PowerISA 2.07B).

But vec_mul / vec_div did not:

- vec_mul applied to float (and later double, with POWER7 VSX).
- 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 all can be very frustrating, at minimum, or even a show stopper, if you are on a tight schedule for your project. Especially if you are not familiar with the evolving history of the PowerISA and supporting compilers.

1.2.2.1 What can we do about this?

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 V↩ SRs, 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.

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 Word (**vmulesw**) and Vector Multiply Odd Signed Word (**vmulosw**) 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. We hope to see this implemented 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 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.

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. 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 32 instructions on a POWER8 (using multiple word even/odd).

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 Binary Coded Decimal (**BCD**) add/subtract for signed 31 digit vector values. The 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 bcd_add / bcd_sub can be generated as a single instruction on POWER8 and later, and 10-11 instructions for Power6/7. This count include the VSR <-> FPRp transfers, BCD <-> DFP conversions, and DFP add/sub. Similarly bcd_mul / bcd_div are implemented in 11 instructions using register transfer and the DFU operations for Power6/7/8.

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.

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 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, subseted) 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 Open↩ POWER 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 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 *numeric* 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 compilers 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

1.2.2.3 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)

1.2.2.3.1 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. This leaves the carry bit in the original (word) column, so use a shift left 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

1.2.2.3.2 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

1.2.2.4 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.

1.2.2.4.1 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 V↩MX 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

1.2.2.4.2 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 (96) of registers 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. GCC adds __ibm128 and _Decimal128 floating point types which are held in Floating-point Registers pairs. GCC recently added the __float128 floating point type which are held in single vector register. Similarly for vector __int128.

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 data types. Such aggregates can be passed as up to eight floating-point or vector registers. This is defined for the Little Endian ELF V2 ABI and is not applicable to Big Endian ELF V1 targets. Also current GCC versions, in common use, do not fully implement this ABI feature.

So we have shown that there are mechanisms for functions to return multiple vector register values. But none are really practical at this time as they not yet available (function or optimal code generation) in current GCC compilers, that are in common use.

Returning pairs of vector __int128 values as _Complex __float128 would be awkward at best. And it is not clear when or if _Complex vector __int128 will be supported. GCC's Generic Vector Extensions are only implemented for vector_size(16). And current GCC compilers can generate some sub-optimal code for passing/returning *homogeneous aggregates* as suggested in the OpenPOWER ABI.

1.2.2.4.3 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.
- So far these functions are small and fully inlined.

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 and let the user decide which works best.

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.

1.2.3 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.

  This is complicated as the PowerISA evolves. The original Altivec (VMX) provided vector multiply (even / odd) operations for byte (char) and halfword (short) integers. Multiple even / odd word (int) instructions were not introduced until PowerISA V2.07 (POWER8). PowerISA 2.07 also introduced vector multiply word modulo which is included under the generic vec_mul.

As the PowerISA evolved adding new vector (VMX) instructions, new facilities (Vector Scalar Extended (VSX)), and specialized vector categories (little endian, AES, SHA2, RAID), these new operators were added to <altivec.h>. This included new specific and generic operations and additional vector element types (long (64-bit) int, __int128, double and quad precision (__Float128) float).

However the 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 ...

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

Also note that 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_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).

The ABI also defines vec_mul as an overloaded operation on integer types, where only the low order half (modulo element size) of the product is returned. The PowerISA does not provide direct multiply modulo instructions for all the integer sizes / types. So this requires a multiple-instruction sequence to implement. Also integer vec_mul is defined in the ABI as "phased in" and is only implemented in the latest GCC versions.

This is a small sample of the complexity we encounter programming at this low level (vector intrinsic) API. Partially this is due to RISC design philosophy where there is a trade-off of software complexity for simpler (hopefully faster) hardware design.

1.2.4 pveclib is not a vector math library

The pveclib does not implement general purpose vector math operations. These should continue to be developed and improved within existing projects (ie LAPACK, OpenBLAS, ATLAS and libmvec).

We believe that pveclib will be helpful to implementors of vector math libraries by providing a higher level, more portable, and more consistent vector interface for the PowerISA. Similarly for implementors of extended arithmetic, cryptographic, compression/decompression, and pattern matching / search libraries.

1.3 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

1.3.1 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

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.

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_128 Union Reference

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

`#include <vec_common_ppc.h>`

Public Attributes

- unsigned __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.

6.1.1 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/vec_common_ppc.h

6.2 __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.

6.2.1 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/vec_f128_ppc.h

Chapter 7

File Documentation

7.1 src/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 <vec_common_ppc.h>
#include <vec_char_ppc.h>
#include <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 PVECLIB 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 G↩ T/LT/EQ/OVF. 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 PowerISA. 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 (*D↩ FP*) 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 ***C**redit* as positive and ***D**edit* 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:

```
  210000000000000000000000000000008
+ 190000000000000000000000000000008
= 400000000000000000000000000000016
```

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:

```
  210000000000000000000000000000008
- 190000000000000000000000000000008
= 020000000000000000000000000000000
```

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.

```
  0000000000000000000000000000002c   1000000000000000000000000000008c
  0000000000000000000000000000001d + 9000000000000000000000000000008d
                                   =
                                0c   8000000000000000000000000000000d
+ 0000000000000000000000000000002c
+ 0000000000000000000000000000001d

= 0000000000000000000000000000001c
```

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);
  }
```

```
  0000000000000000000000000000002c   1000000000000000000000000000008c
  0000000000000000000000000000001d + 9000000000000000000000000000008d
                                   = ?
                                     8000000000000000000000000000000d
                                   + 9999999999999999999999999999999c
                                   + 0000000000000000000000000000001c
                                1d   2000000000000000000000000000000c
+ 0000000000000000000000000000002c
+ 0000000000000000000000000000001d

= 0000000000000000000000000000000c
```

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 POWER9. 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 PO↩ WER9 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 #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 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 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 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 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 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 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 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	a 128-bit vector as an signed __int128 number in the range -9999999999999999999999999999999 to +9999999999999999999999999999999.

Returns

128-bit vector signed BCD value in the range -9999999999999999999999999999999 to +9999999999999999999999999999999

7.1.6.8 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 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 0-99999999999999999999999999999999.

Returns

128-bit vector unsigned BCD value in the range 0-99999999999999999999999999999999.

7.1.6.10 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 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 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 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 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 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 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 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 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 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.

Parameters

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 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 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 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 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 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 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 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 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 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 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 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 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 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 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 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 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 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 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 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 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 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 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.
↩ _↩ *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 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 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 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 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 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 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 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 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 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 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.
$\hookleftarrow$ $\_\hookleftarrow$ *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 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 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 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 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 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 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 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 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 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 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 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 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 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 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 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 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 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 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 0-9999999999999999999999999999999.

7.1.6.70 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 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 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 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 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	Vector of doublewords (long int).
lval	FPR pair containing a _Decimal128.

Returns

FPR pair containing a _Decimal128.

7.1.6.75 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/vec_char_ppc.h File Reference

Header package containing a collection of 128-bit SIMD operations over 8-bit integer (char) elements.

```
#include <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)

  Count leading zeros for a vector 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 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 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 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 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 static vui8_t vec_clzb (vui8_t *vra*) [inline],[static]

Count leading zeros for a vector 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 integer (byte) elements.

Returns

128-bit vector with the Leading Zeros count for each byte element.

7.2.4.3 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.

7.2.4.4 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.

7.2.4.5 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.

7.2.4.6 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.

7.2.4.7 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.

7.2.4.8 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.

7.2.4.9 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.

7.2.4.10 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.

7.2.4.11 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.

7.2.4.12 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.

7.2.4.13 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.

7.2.4.14 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.

7.2.4.15 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.

7.2.4.16 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.

7.2.4.17 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.

7.2.4.18 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.

7.2.4.19 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.

7.2.4.20 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.

7.2.4.21 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/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 int vi128_t
  vector of one 128-bit signed __int128 element.
- typedef __vector unsigned int vui128_t
  vector of one 128-bit unsigned __int128 element.
- typedef __vector __bool unsigned int vb128_t
  vector of 128-bit bool __int128 elements.

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 (VS↩ R0-31) 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_DF↩ P2BCD().

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

7.4 src/vec_f128_ppc.h File Reference

Header package containing a collection of 128-bit SIMD operations over Quad-Precision floating point elements.

```
#include <vec_common_ppc.h>
#include <vec_int128_ppc.h>
#include <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 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_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)

  Clear the sign bit of __float128 input and return the resulting positive __float128 value.
- 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_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 f128y and merge with the magnitude from f128x. 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_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. For 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_iszerof128 (__binary128 f128)

  Return 128-bit vector boolean true value, if the value that is +-0.0.
- 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').

7.4.1 Detailed Description

Header package containing a collection of 128-bit SIMD operations over Quad-Precision floating point elements.

PowerISA 3.0 added Quad-Precision floating point type and operations to the Vector-Scalar Extension (VSX) facility. The first hardware implementation is available in POWER9.

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 implement in a few vector logical instruction 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.

So it is not unreasonable for this header to provide vector forms of the __float128 classification functions (isnormal/subnormal/finite/inf/nan/zero, copysign, and abs). 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.

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 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.
- Providing special 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 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.

7.4.3 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.

7.4.4 Function Documentation

7.4.4.1 static __binary128 vec_absf128 (__binary128 *f128*) [inline],[static]

Clear the sign bit of __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.

7.4.4.2 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.

7.4.4.3 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.

7.4.4.4 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 Power↩ ISA-2.07. The sign bit is ignored.

processor	Latency	Throughput
power8	6-29	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.

7.4.4.5 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.

7.4.4.6 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.

7.4.4.7 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.

7.4.4.8 static __binary128 vec_const_huge_valf128 ( ) [inline],[static]

return a positive infinity.

Returns

const __float128 positive infinity.

7.4.4.9 static __binary128 vec_const_inff128 ( ) [inline],[static]

return a positive infinity.

Returns

a const __float128 positive infinity.

7.4.4.10 static __binary128 vec_const_nanf128 ( ) [inline], [static]

return a quiet NaN.

Returns

a const __float128 quiet NaN.

7.4.4.11 static __binary128 vec_const_nansf128 ( ) [inline], [static]

return a signaling NaN.

Returns

a const __float128 signaling NaN.

7.4.4.12 static __binary128 vec_copysignf128 (__binary128 *f128x,* __binary128 *f128y*) [inline], [static]

Copy the sign bit from f128y and merge with the magnitude from f128x. The merged result is returned as a __↩ float128 value.

processor	Latency	Throughput
power8	2-11	2/cycle
power9	2	4/cycle

Parameters

f128x	a __float128 value containing the magnitude.
f128y	a __float128 value containing the sign bit.

Returns

a __float128 value with magnitude from f128x and the sign of f128y.

7.4.4.13 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.

7.4.4.14 static int vec_isinf_signf128 (__binary128 *f128*) [inline],[static]

Return true (nonzero) value if the __float128 value is infinity. For 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.

7.4.4.15 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..

7.4.4.16 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. This requires a combination of verifying the exponent and that any bit of the significand is nonzero. Using the combined vec_all_eq / vec_any_gt compare conditional verify both conditions before negating the result from zero to all ones.. The sign bit is ignored.

processor	Latency	Throughput
power8	10-19	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.

7.4.4.17 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.

7.4.4.18 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).

7.4.4.19 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).

7.4.4.20 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.

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.

7.4.4.21 static int vec_signbitf128 (__binary128 *f128*) [inline],[static]

Return int boolean true if the __float128 value is negative (sign bit is '1').

Anding with a signmask and then vec_all_eq compare with that mask generates the boolean of the sign bit.

processor	Latency	Throughput
power8	4-20	2/cycle
power9	3	2/cycle

Parameters

f128	a __float128 value in vector.

Returns

a int boolean indicating the sign bit.

7.4.4.22 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.

7.4.4.23 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.

7.4.4.24 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.

7.4.4.25 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.

7.4.4.26 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.

7.4.4.27 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.

7.4.4.28 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.

7.4.4.29 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.

7.4.4.30 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.

7.4.4.31 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.

7.5 src/vec_f32_ppc.h File Reference

Header package containing a collection of 128-bit SIMD operations over 4x32-bit floating point elements.

```
#include <vec_common_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 vf32y merged with magnitude from vf32x 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.

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 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 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 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 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 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 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 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 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 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 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 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 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 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 static vf32_t vec_copysignf32 (vf32_t *vf32x,* vf32_t *vf32y*) [inline], [static]

Copy the sign bit from vf32y merged with magnitude from vf32x and return the resulting vector float values.

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

Parameters

vf32x	vector float values containing the magnitudes.

Parameters

vf32y	vector float values containing the sign bits.

Returns

vector float values with magnitude from vf32x and the sign of vf32y.

7.5.4.15 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 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 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 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 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 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).

7.6 src/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 <vec_common_ppc.h>
#include <vec_int64_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 vf64y merged with magnitude from vf64x 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 vf64_t vec_unpack_longdouble (long double lval)

  Copy the pair of doubles from a IBM long double to a vector double.

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) 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.

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 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 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 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 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 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 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 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 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 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 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 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 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 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 static vf64_t vec_copysignf64 (vf64_t *vf64x,* vf64_t *vf64y*) [inline],[static]

Copy the sign bit from vf64y merged with magnitude from vf64x and return the resulting vector double values.

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

Parameters

vf64x	vector double values containing the magnitudes.

Parameters

vf64y	vector double values containing the sign bits.

Returns

vector double values with magnitude from vf64x and the sign of vf64y.

7.6.4.15 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 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 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 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 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 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 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 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.

7.7 src/vec_int128_ppc.h File Reference

Header package containing a collection of 128-bit computation functions implemented with PowerISA VMX and VSX instructions.

```
#include <vec_common_ppc.h>
#include <vec_int64_ppc.h>
```

Functions

- static vui128_t vec_absduq (vui128_t vra, vui128_t vrb)

  Vector Absolute Difference Unsigned 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.
- 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_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_popcntq (vui128_t vra)

  Vector Population Count Quadword.
- 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 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_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_vmuloud (vui64_t a, vui64_t b)

  Vector Multiply Odd Unsigned Doublewords.

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.

7.7.3 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.

7.7.3.1 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]);
}
```

7.7.3.2 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) 1000000000000000UL };
  // Magic numbers for multiplicative inverse to divide by 10**31
  // are 48049504185897259083631856820830611 67, 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) 1000000000000000UL };
  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.

7.7.3.3 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.

7.7.3.3.1 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;
  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_addcuq (mphl, mc);
  mp = vec_muludq (&mc, m2h, m1l);
  mplh = vec_addcq (&mq, mplh, mp);
  mphl = vec_addcq (&mc, mphl, mq);
  mp = vec_muludq (&mphh, m2h, m1h);
  mplh = vec_addcq (&mc, mplh, mp);
  mphl = vec_addcuq (mphh, 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.

7.7.3.3.2 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) 1000000000000000UL };
  const vui128_t zero = (vui128_t) { (__int128) 0UL };
  // Magic numbers for multiplicative inverse to divide by 10**31
  // are 48049504185897259083631856820830611 67, 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  = +100000000000000000000000000000000Q
  const vui128_t ten31 = (vui128_t)
         { (__int128) 10000000000000000UL * (__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 0-999999999999999999999999999999 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.

7.7.4 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.

7.7.5 Function Documentation

7.7.5.1 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
vrb	vector of unsigned __int128

Returns

vector of the absolute difference.

7.7.5.2 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.

7.7.5.3 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.

7.7.5.4 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.

7.7.5.5 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.

7.7.5.6 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.

7.7.5.7 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.

7.7.5.8 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.

7.7.5.9 static vui128_t vec_clzq (vui128_t *vra*) [inline], [static]

Vector Count Leading Zeros Quadword.

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	19-28	1/cycle
power9	25-36	1/cycle

Parameters

vra	a 128-bit vector treated a __int128.

Returns

a 128-bit vector with bits 121:127 containing the count of leading zeros.

7.7.5.10 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.

7.7.5.11 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.

7.7.5.12 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.

7.7.5.13 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.

7.7.5.14 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.

7.7.5.15 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.

7.7.5.16 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.

7.7.5.17 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.

7.7.5.18 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.

7.7.5.19 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.

7.7.5.20 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.

Parameters

vrb	128-bit vector treated as an signed __int128.

Returns

128-bit vector boolean reflecting vector signed __int128 compare not equal.

7.7.5.21 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.

7.7.5.22 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.

7.7.5.23 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.

7.7.5.24 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.

7.7.5.25 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.

7.7.5.26 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.

7.7.5.27 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.

Parameters

vrb	128-bit vector treated as an vector signed __int128 (qword) element.

Returns

boolean __int128 for all 128-bits, true if equal, false otherwise.

7.7.5.28 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.

7.7.5.29 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.

7.7.5.30 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.

7.7.5.31 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.

7.7.5.32 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.

7.7.5.33 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.

7.7.5.34 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.

7.7.5.35 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.

7.7.5.36 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.

7.7.5.37 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.

7.7.5.38 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.

7.7.5.39 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.

7.7.5.40 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.

7.7.5.41 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.

7.7.5.42 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.

7.7.5.43 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.

7.7.5.44 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.

7.7.5.45 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.

7.7.5.46 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.

7.7.5.47 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.

Parameters

q	128-bit signed __int128 containing the quotient from vec_divuq_10e31().

Returns

the remainder as vector signed __int128.

7.7.5.48 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.

7.7.5.49 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.

7.7.5.50 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.

7.7.5.51 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.

7.7.5.52 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 produ256 bit product of two 128 bit values a, b. Only the low order 128 bits of the product are returned.

processor	Latency	Throughput
power8	30-32	1/cycle
power9	5-7	2/cycle

Parameters

a	128-bit vector treated as __vector unsigned long int.
b	128-bit vector treated as __vector unsigned long int.
c	128-bit vector treated as __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.

7.7.5.53 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.

7.7.5.54 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.

7.7.5.55 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.

7.7.5.56 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.

7.7.5.57 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.

7.7.5.58 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.

7.7.5.59 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.

7.7.5.60 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.

7.7.5.61 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.

7.7.5.62 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 int.
vrb	128-bit vector unsigned long int.

Returns

vector unsigned long int of the low order 64-bits of the unsigned 128-bit product of the doubleword elements from vra and vrb.

7.7.5.63 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	56-64	1/cycle
power9	33-39	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.

7.7.5.64 static vui128_t vec_popcntq (vui128_t *vra*) [inline], [static]

Vector Population Count Quadword.

Count the number of '1' bits within a vector __int128 and return the count (0-128) in a vector __int128.

processor	Latency	Throughput
power8	15	2/2 cycles
power9	16	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.

7.7.5.65 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.

7.7.5.66 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.

7.7.5.67 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.

7.7.5.68 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	2/2 cycles
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.

7.7.5.69 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	2/2 cycles
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.

7.7.5.70 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.

7.7.5.71 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.

7.7.5.72 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.

7.7.5.73 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.

7.7.5.74 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.

7.7.5.75 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.
```

7.7.5.76 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.

7.7.5.77 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.

7.7.5.78 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 _AR↩ CH_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.

7.7.5.79 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.

7.7.5.80 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.

7.7.5.81 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.

7.7.5.82 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.

7.7.5.83 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.

7.7.5.84 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.

7.7.5.85 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.

7.7.5.86 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.

7.7.5.87 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	8-13	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.

7.7.5.88 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.

7.8 src/vec_int16_ppc.h File Reference

Header package containing a collection of 128-bit SIMD operations over 16-bit integer elements.

```
#include <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)

  Count Leading Zeros for a vector unsigned short (halfword) 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 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 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/PO↩ WER9) 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 (POWE↩ R8) added several useful halfword 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_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 high and shift immediate operations.

7.8.2 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).

7.8.2.1 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.

7.8.3 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.

7.8.3.1 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.

7.8.3.2 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.

7.8.4 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 POWER9).

processor	Latency	Throughput
power8	9-13	1/cycle
power9	10-14	1/cycle

7.8.4.1 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.

7.8.5 Function Documentation

7.8.5.1 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.

7.8.5.2 static vui16_t vec_clzh (vui16_t *vra*) [inline],[static]

Count Leading Zeros for a vector unsigned short (halfword) 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 integer (halfword) elements.

Returns

128-bit vector with the Leading Zeros count for each halfword element.

7.8.5.3 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.

7.8.5.4 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.

7.8.5.5 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.

7.8.5.6 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.

7.8.5.7 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.

7.8.5.8 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.

7.8.5.9 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.

7.8.5.10 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.

7.8.5.11 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.

7.8.5.12 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.

7.8.5.13 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.

7.8.5.14 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.

7.8.5.15 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.

7.8.5.16 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/vec_int32_ppc.h File Reference

Header package containing a collection of 128-bit SIMD operations over 32-bit integer elements.

```
#include <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_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 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.

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 VMX. 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 (POWER7 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 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.

7.9.2.1 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.

7.9.2.2 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
```

7.9.3 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.

7.9.3.1 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;
    }
}
```

7.9.4 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.

7.9.5 Function Documentation

7.9.5.1 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.

7.9.5.2 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.

7.9.5.3 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.

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.

7.9.5.4 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).

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.

7.9.5.5 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.

7.9.5.6 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.

7.9.5.7 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.

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.

7.9.5.8 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.

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.

7.9.5.9 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.

7.9.5.10 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.

7.9.5.11 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.

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.

7.9.5.12 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.

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.

7.9.5.13 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.

7.9.5.14 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.

7.9.5.15 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.

7.9.5.16 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.

7.9.5.17 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.

7.9.5.18 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.

7.10 src/vec_int64_ppc.h File Reference

Header package containing a collection of 128-bit SIMD operations over 64-bit integer elements.

```
#include <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)

  Count leading zeros for a vector unsigned long int.
- 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 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 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 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 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 vui32_t vec_vpkudum (vui64_t vra, vui64_t vrb)

  Vector Pack Unsigned Doubleword Unsigned Modulo.
- 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_vmsumuwm (vui32_t vra, vui32_t vrb, vui64_t vrc)

  Vector Multiply-Sum Unsigned Word Modulo.

7.10.1 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, ...) V↩ MX 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 OpenP↩ OWER ABI. Examples include: Absolute Difference Doubleword and Multiply-Sum Unsigned Word Modulo.

7.10.2 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 (POWER7) 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)

7.10.2.1 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.

7.10.2.2 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;
#ifdef _ARCH_PWR9
  __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;
#ifdef _ARCH_PWR9
  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 P↩ OWER8 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.

7.10.3 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().

7.10.4 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.

7.10.4.1 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);
    }
}
```

7.10.5 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.

7.10.6 Function Documentation

7.10.6.1 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.

7.10.6.2 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.

7.10.6.3 static vui64_t vec_clzd (vui64_t *vra*) [inline],[static]

Count leading zeros for a vector unsigned long int.

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	2	2/cycle
power9	2	2/cycle

Parameters

vra	a 128-bit vector treated a __int128.

Returns

a 128-bit vector with bits 121:127 containing the count of leading zeros.

7.10.6.4 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.

7.10.6.5 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.

7.10.6.6 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.

7.10.6.7 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.

7.10.6.8 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.

7.10.6.9 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.

7.10.6.10 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.

7.10.6.11 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.

7.10.6.12 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.

7.10.6.13 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.

7.10.6.14 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.

7.10.6.15 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.

7.10.6.16 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.

7.10.6.17 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.

7.10.6.18 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.

7.10.6.19 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.

7.10.6.20 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.

7.10.6.21 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.

7.10.6.22 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.

Parameters

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.

7.10.6.23 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.

7.10.6.24 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.

7.10.6.25 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.

7.10.6.26 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.

7.10.6.27 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.

7.10.6.28 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.

7.10.6.29 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.

7.10.6.30 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.

7.10.6.31 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.

7.10.6.32 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.

7.10.6.33 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.

7.10.6.34 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.

Parameters

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.

7.10.6.35 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.

7.10.6.36 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.

7.10.6.37 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.

7.10.6.38 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.

7.10.6.39 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.

7.10.6.40 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.

7.10.6.41 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.

7.10.6.42 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.

7.10.6.43 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.

7.10.6.44 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.

7.10.6.45 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.

7.10.6.46 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.

7.10.6.47 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.

7.10.6.48 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.

7.10.6.49 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.

7.10.6.50 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.

7.10.6.51 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.

7.10.6.52 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.

7.10.6.53 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.

7.10.6.54 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.

7.10.6.55 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.

7.10.6.56 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.

7.10.6.57 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.

7.10.6.58 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.

7.10.6.59 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.

7.10.6.60 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.

7.10.6.61 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.

7.10.6.62 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 PowerISA 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 product of the vra and vrb, plus the doubleword elements of vrc.

7.10.6.63 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.

7.10.6.64 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.

7.10.6.65 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.

7.10.6.66 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.

7.10.6.67 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.

7.10.6.68 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