dss dss

Data Stream Stop

dss	STRM	(A=0)	Form X
dssall	STRM	(A=1)	

31	A	0_0	STRM	0_0000	0000_0	822	0
0–5	6	7–8	9–10	11–15	16–20	21–30	31

```
    DataStreamPrefetchControl ← "stop" || STRM
```

Note that A does not represent **r**A in this instruction.

If A=0 and a data stream associated with the stream ID specified by **STRM** exists, this instruction terminates prefetching of that data stream. It has no effect if the specified stream does not exist.

If A=1, this instruction terminates prefetching of all existing data streams (the STRM field is ignored.)

In addition, executing a **dss** instruction ensures that all accesses associated with data stream prefetching caused by preceding dst and dstst instructions that specified the same stream ID as that specified by the **dss** instruction (A=0), or by all preceding **dst** and **dstst** instructions (A=1), will be in group G1 with respect to the memory barrier created by a subsequent **sync** instruction, refer to Section 5.1, "PowerPC Shared Memory," for more information.

See Section 5.2.1, "Software-Directed Prefetch" for more information on using the **dss** instruction.

Other registers altered:

- None

Simplified mnemonics:

dss **STRM**	equivalent to	**dss**	**STRM**, **0**
dssall	equivalent to	**dss**	**0**, **1**

For more information on the **dss** instruction, refer to Chapter 5, "Cache, Exceptions, and Memory Management."

dst dst

Data Stream Touch

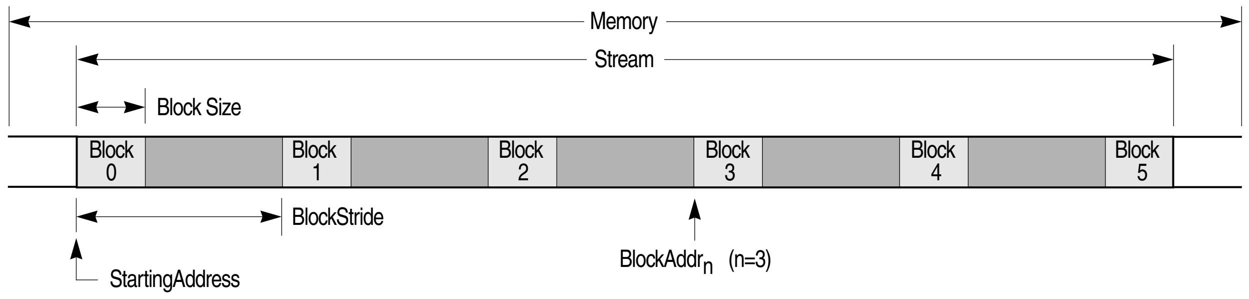
dst	**r**A,**r**B,STRM	(T=0)	Form X
dstt	**r**A,**r**B,STRM	(T=1)	

31	T	0_0	STRM	A	B	342	0
0 5	6	7 8	9 10	11 15	16 20	21 30	31

```
addr$_{0:63}$ ← (rA)
DataStreamPrefetchControl ← “start” || STRM || T || (rB) || addr
```

This instruction initiates a software directed cache prefetch. The instruction is a hint to hardware that performance will probably be improved if the cache blocks containing the specified data stream are fetched into the data cache because the program will probably soon load from the stream.

The instruction associates the data stream specified by the contents of **r**A and **r**B with the stream ID specified by **STRM**. The instruction defines a data stream **STRM** as starting at an effective address (**r**A) and having count units of size quad words separated by stride bytes (as specified in **r**B). The **T** bit of the instruction indicates whether the data stream is likely to be loaded from fairly frequently in the near future (**T** = 0) or to be transient and referenced very few times (**T** = 1).



The **dst** instruction does the following:

- Defines the characteristics of a data stream **STRM** by the contents of **r**A and **r**B
- Associates the stream with a specified stream ID, **STRM** (Range for STRM is 0-3)
- Indicates that the data in the specified stream **STRM** starting at the address in **r**A may soon be loaded
- Indicates whether memory locations within the stream are likely to be needed over a longer period of time (**T**=0) or be treated as transient data (**T**=1)
- Terminates prefetching from any stream that was previously associated with the specified stream ID, **STRM**.

The specified data stream is encoded for 32-bit follows:

- Effective address: **r**A, where **r**A ≠ 0
- Block size: **r**B[3–7] if **r**B[3–7] ≠ 0; otherwise 32
- Block count: **r**B[8–15] if **r**B[8–15] ≠ 0; otherwise 256
- Block stride: **r**B[16–31] if **r**B[16–31] ≠ 0; otherwise 32768

///	Block Size	Block Count	Block Stride
0 2	3 7	8 15	16 31

Other registers altered:

- None

Simplified mnemonics:

dst	**r**A,**r**B,STRM	equivalent to	**dst**	**r**A,**r**B,STRM,**0**
dstt	**r**A,**r**B,STRM	equivalent to	**dst**	**r**A,**r**B**,**STRM,**1**

For more information on the **dst** instruction, refer to Chapter 5, “Cache, Exceptions, and Memory Management.”

dstst dstst

Data Stream Touch for Store

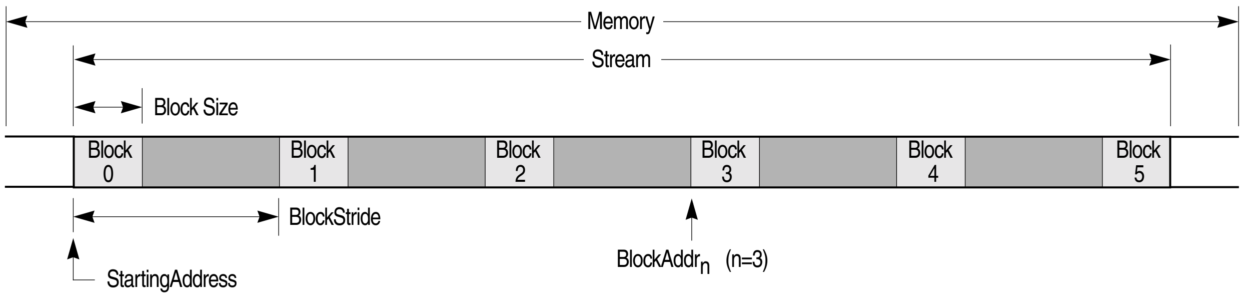
dstst	**r**A,**r**B,STRM	(T=0)	Form X
dststt	**r**A,**r**B,STRM	(T=1)	

31	T	0_0	STRM	A	B	374	0
0 5	6	7 8	9 10	11 15	16 20	21 30	31

```
addr$_{0:63}$ ← (rA)
DataStreamPrefetchControl ← “start” || T || static || (rB) || addr
```

This instruction initiates a software directed cache prefetch. The instruction is a hint to hardware that performance will probably be improved if the cache blocks containing the specified data stream are fetched into the data cache because the program will probably soon write to (store into) the stream.

The instruction associates the data stream specified by the contents of **r**A and **r**B with the stream ID specified by **STRM**. The instruction defines a data stream **STRM** as starting at an effective address (**r**A) and having count units of size quad words separated by stride bytes (as specified in **r**B). The **T** bit of the instruction indicates whether the data stream is likely to be stored into fairly frequently in the near future (**T** = 0) or to be transient and referenced very few times (**T** = 1).



The **dstst** instruction does the following:

- Defines the characteristics of a data stream **STRM** by the contents of **r**A and **r**B
- Associates the stream with a specified stream ID, **STRM** (Range for STRM is 0-3)
- Indicates that the data in the specified stream **STRM** starting at the address in **r**A may soon be stored in to memory
- Indicates whether memory locations within the stream are likely to be stored into fairly frequently in the near future (**T**=0) or be treated as transient data (**T**=1)
- Terminates prefetching from any stream that was previously associated with the specified stream ID, **STRM**.

The specified data stream is encoded for 32-bit follows:

- Effective address: **r**A, where **r**A ≠ 0
- Block size: **r**B[3–7] if **r**B[3–7] ≠ 0; otherwise 32
- Block count: **r**B[8–15] if **r**B[8–15] ≠ 0; otherwise 256
- Block stride: **r**B[16–31] if **r**B[16–31] ≠ 0; otherwise 32768

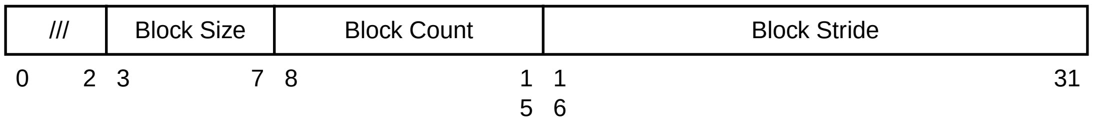


Figure 6-1. Format of rB in dst instruction (32-bit)

Other registers altered:

- None

Simplified mnemonics:

dstst	**r**A,**r**B,STRM	equivalent to	**dstst** **r**A,**r**B,STRM,**0**
dststt	**r**A,**r**B,STRM	equivalent to	**dstst** **r**A,**r**B**,**STRM,**1**

For more information on the **dstst** instruction, refer to Chapter 5, “Cache, Exceptions, and Memory Management.”

lvebx lvebx

Load Vector Element Byte Indexed

lvebx **v**D,**r**A,**r**B Form X

31	**v**D	A	B	7	0
0 5	6 10	11 15	16 20	21 30	31

- For 32-bit:

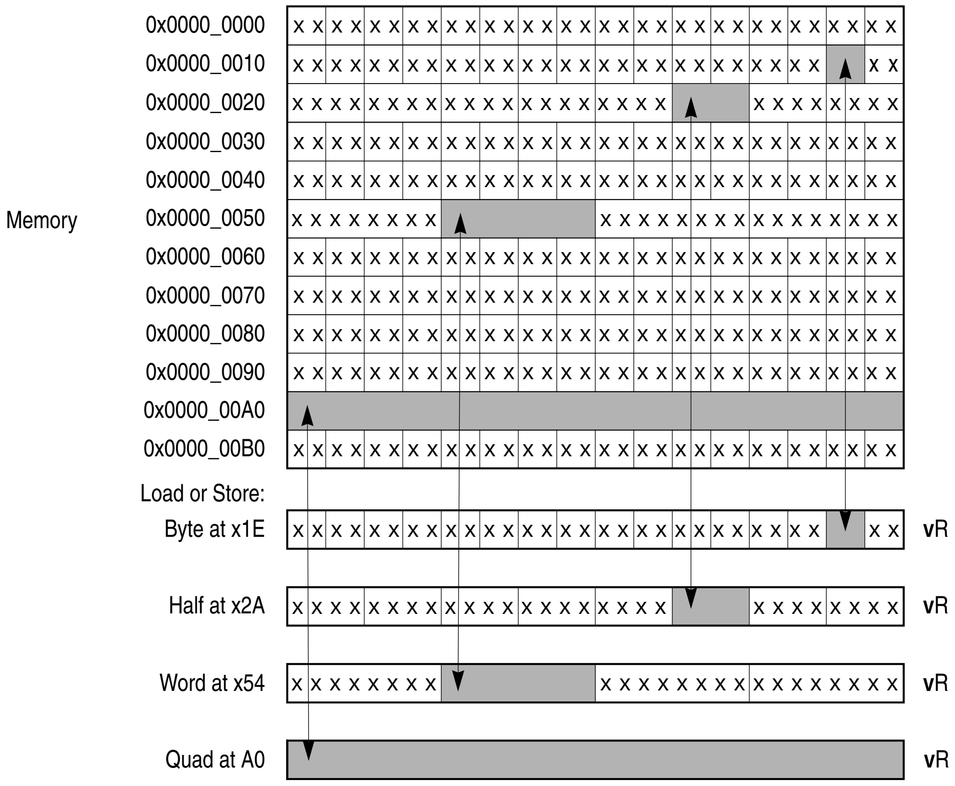
```
if rA=0 then b ← 0
else         b ← (rA)
EA ← b + (rB)
eb ← $EA_{28:31}$
vD ← undefined
if the processor is in big-endian mode
 then $vD_{eb*8:(eb*8)+7}$← MEM(EA,1)
 else $vD_{120-(eb*8):127-(eb*8)}$← MEM(EA,1)
```

— EA = (**r**A|0)+(**r**B); m = EA[28-31] (the offset of the byte in its aligned quadword).

For big-endian mode, the byte addressed by EA is loaded into byte m of **v**D. In little-endian mode, it is loaded into byte (15–m) of **v**D. Remaining bytes in **v**D are undefined.

Other registers altered:

- None



Note: In vector registers, x means boundedly undefined after a load and don’t care after a store. In memory, x means don’t care after a load, and leave at current value after a store.

Figure 6-2. Effects of Example Load/Store Instructions

lvehx lvehx

Load Vector Element Half Word Indexed

lvehx **v**D,**r**A,**r**B Form X

31	**v**D	A	B	39	0
0 5	6 10	11 15	16 20	21 30	31

- For 32-bit:

```
if rA=0 then  b ← 0
else          b ← (rA)
EA ← (b + (rB)) & (~1)
eb ← $EA_{28:31}$
vD ← undefined
if the processor is in big-endian mode
  then $vD_{(eb*8):(eb*8)+15}$ ← MEM(EA,2)
  else $vD_{112-(eb*8):127-(eb*8)}$ ← MEM(EA,2)
```

— Let the EA be the result of ANDing the sum (**r**A|0)+(**r**B) with ~1. Let m = EA[28-30]; m is the half-word offset of the half-word in its aligned quadword in memory.

If the processor is in big-endian mode, the half-word addressed by EA is loaded into half-word m of **v**D. If the processor is in little-endian mode, the half-word addressed by EA is loaded into half-word (7-m) of **v**D. The remaining half-word s in **v**D are set to undefined values. Figure 6-2 shows this instruction.

Other registers altered:

- None

lvewx lvewx

Load Vector Element Word Indexed

lvewx **v**D,**r**A,**r**B Form X

31	**v**D	A	B	71	0
0 5	6 10	11 15	16 20	21 30	31

- For 32-bit:

```
if rA=0 then b ← 0
else         b ← (rA)
EA ← (b + (rB)) & (~3)
eb ← EA28:31
vD ← undefined
if the processor is in big-endian mode
 then vDeb*8:(eb*8)+31← MEM(EA,4)
 else vD96-(eb*8):127-(eb*8)← MEM(EA,4)
```

— Let the EA be the result of ANDing the sum (**r**A|0)+(**r**B) with ~3. Let m = EA[28–29]; m is the word offset of the word in its aligned quadword in memory.

If the processor is in big-endian mode, the word addressed by EA is loaded into word m of **v**D. If the processor is in little-endian mode, the word addressed by EA is loaded into word (3-m) of **v**D. The remaining words in **v**D are set to undefined values. Figure 6-2 shows this instruction.

Other registers altered:

- None

lvsl lvsl

Load Vector for Shift Left

lvsl **v**D,**r**A,**r**B Form X

31	**v**D	A	B	6	0
0 5	6 10	11 15	16 20	21 30	31

- For 32-bit:

```
if rA = 0 then b ← 0
    else b ← (rA)
addr_{0:31} ← b + (rB)
sh ← addr_{28-31}
if sh = 0x0 then (vD)_{0:127} ← 0x000102030405060708090A0B0C0D0E0F
if sh = 0x1 then (vD)_{0:127} ← 0x0102030405060708090A0B0C0D0E0F10
if sh = 0x2 then (vD)_{0:127} ← 0x02030405060708090A0B0C0D0E0F1011
if sh = 0x3 then (vD)_{0:127} ← 0x030405060708090A0B0C0D0E0F101112
if sh = 0x4 then (vD)_{0:127} ← 0x0405060708090A0B0C0D0E0F10111213
if sh = 0x5 then (vD)_{0:127} ← 0x05060708090A0B0C0D0E0F1011121314
if sh = 0x6 then (vD)_{0:127} ← 0x060708090A0B0C0D0E0F101112131415
if sh = 0x7 then (vD)_{0:127} ← 0x0708090A0B0C0D0E0F10111213141516
if sh = 0x8 then (vD)_{0:127} ← 0x08090A0B0C0D0E0F1011121314151617
if sh = 0x9 then (vD)_{0:127} ← 0x090A0B0C0D0E0F101112131415161718
if sh = 0xA then (vD)_{0:127} ← 0x0A0B0C0D0E0F10111213141516171819
if sh = 0xB then (vD)_{0:127} ← 0x0B0C0D0E0F101112131415161718191A
if sh = 0xC then (vD)_{0:127} ← 0x0C0D0E0F101112131415161718191A1B
if sh = 0xD then (vD)_{0:127} ← 0x0D0E0F101112131415161718191A1B1C
if sh = 0xE then (vD)_{0:127} ← 0x0E0F101112131415161718191A1B1C1D
if sh = 0xF then (vD)_{0:127} ← 0x0F101112131415161718191A1B1C1D1E
```

— Let the EA be the sum (**r**A|0)+(**r**B). Let sh = EA[28–31].

Let X be the 32-byte value 0x00 || 0x01 || 0x02 || ... || 0x1E || 0x1F. Bytes sh:sh+15 of X are placed into **v**D. Figure 6-3 shows how this instruction works.

Other registers altered:

- None

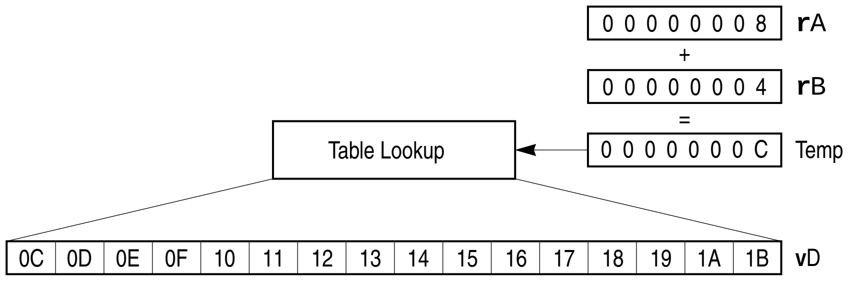


Figure 6-3. Load Vector for Shift Left

The above **lvsl** instruction followed by a Vector Permute (**vperm**) would do a simulated alignment of a four-element floating-point vector misaligned on quad-word boundary at address 0x0....C.

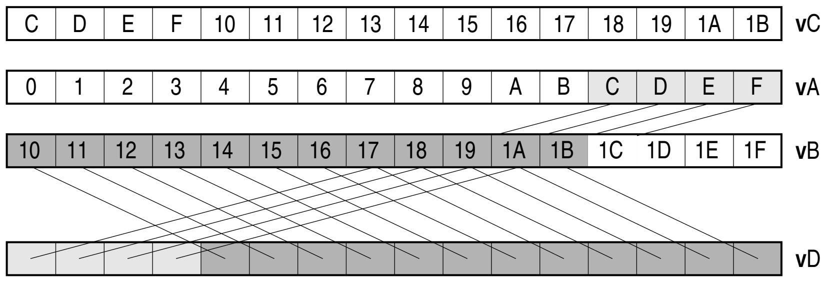

C D E F 10 11 12 13 14 15 16 17 18 19 1A 1B
vC
0 1 2 3 4 5 6 7 8 9 A B C D E F
vA
10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F
vB
vD

Figure 6-4. Instruction vperm Used in Aligning Data

Refer, also, to the description of the **lvsr** instruction for suggested uses of the **lvsl** instruction.

lvsr lvsr

Load Vector for Shift Right

lvsr **v**D,**r**A,**r**B Form X

0–5	6–10	11–15	16–20	21–30	31
31	**v**D	A	B	38	0

- For 32-bit:

```
if rA = 0 then b ← 0
else           b ← (rA)
EA ← b + (rB)
sh ← EA$_{28:31}$
if sh=0x0 then vD ← 0x101112131415161718191A1B1C1D1E1F
if sh=0x1 then vD ← 0x0F101112131415161718191A1B1C1D1E
if sh=0x2 then vD ← 0x0E0F101112131415161718191A1B1C1D
if sh=0x3 then vD ← 0x0D0E0F101112131415161718191A1B1C
if sh=0x4 then vD ← 0x0C0D0E0F101112131415161718191A1B
if sh=0x5 then vD ← 0x0B0C0D0E0F101112131415161718191A
if sh=0x6 then vD ← 0x0A0B0C0D0E0F10111213141516171819
if sh=0x7 then vD ← 0x090A0B0C0D0E0F101112131415161718
if sh=0x8 then vD ← 0x08090A0B0C0D0E0F1011121314151617
if sh=0x9 then vD ← 0x0708090A0B0C0D0E0F10111213141516
if sh=0xA then vD ← 0x060708090A0B0C0D0E0F101112131415
if sh=0xB then vD ← 0x05060708090A0B0C0D0E0F1011121314
if sh=0xC then vD ← 0x0405060708090A0B0C0D0E0F10111213
if sh=0xD then vD ← 0x030405060708090A0B0C0D0E0F101112
if sh=0xE then vD ← 0x02030405060708090A0B0C0D0E0F1011
if sh=0xF then vD ← 0x0102030405060708090A0B0C0D0E0F10
```

— Let the EA be the sum (**r**A|0)+(**r**B). Let sh = EA[28–31].

Let X be the 32-byte value 0x00 || 0x01 || 0x02 || ... || 0x1E || 0x1F. Bytes (16-sh):(31-sh) of X are placed into **v**D.

Note that **lvsl** and **lvsr** can be used to create the permute control vector to be used by a subsequent **vperm** instruction. Let X and Y be the contents of **v**A and **v**B specified by the **vperm**. The control vector created by **lvsl** causes the **vperm** to select the high-order 16 bytes of the result of shifting the 32-byte value X || Y left by sh bytes. The control vector created by **vsr** causes the **vperm** to select the low-order 16 bytes of the result of shifting X || Y right by sh bytes.

These instructions can also be used to rotate or shift the contents of a vector register by sh bytes. For rotating, the vector register to be rotated should be specified as both **v**A and **v**B for **vperm**. For shifting left, the **v**B register for **vperm** should contain all zeros and **v**A should contain the value to be shifted, and vice versa for shifting right. Figure 6-3 shows a similar instruction only in that figure the shift is to the left

No other registers altered.

lvx lvx

Load Vector Indexed

lvx **v**D,**r**A,**r**B (LRU = 0) Form X

31	**v**D	A	B	103	0
0 5	6 10	11 15	16 20	21 30	31

- For 32-bitt:

```
if rA=0 then b ← 0
else         b ← (rA)
EA ← (b + (rB)) & (~0xF)
if the processor is in big-endian mode
 then vD ← MEM(EA,16)
 else vD ← MEM(EA+8,8) || MEM(EA,8)
```

Let the EA be the result of ANDing the sum (**r**A|0)+(**r**B) with ~0xF.

If the processor is in big-endian mode, the quadword in memory addressed by EA is loaded into **v**D.

If the processor is in little-endian mode, the doubleword addressed by EA is loaded into **v**D[64–127] and the doubleword addressed by EA+8 is loaded into **v**D[0–63]. Note that normal little-endian PowerPC address swizzling is also performed. See Section 3.1, “Data Organization in Memory,” for more information.

Figure 6-3 shows this instruction.

Other registers altered:

- None

lvxl lvxl

Load Vector Indexed LRU

lvxl **v**D,**r**A,**r**B (LRU = 1) Form X

31	**v**D	A	B	359	0
0 5	6 10	11 15	16 20	21 30	31

- For 32-bit:

```
if rA=0 then b ← 0
else         b ← (rA)
EA ← (b + (rB)) & (~0xF)
if the processor is in big-endian mode
 then vD ← MEM(EA,16)
 else vD ← MEM(EA+8,8) || MEM(EA,8)
```

Let the EA be the result of ANDing the sum (**r**A|0)+(**r**B) with ~0xF.

If the processor is in big-endian mode, the quadword addressed by EA is loaded into **v**D.

If the processor is in little-endian mode, the doubleword addressed by EA is loaded into **v**D[64–127] and the doubleword addressed by EA+8 is loaded into **v**D[0–63]. Note that normal little-endian PowerPC address swizzling is also performed. See Section 3.1, “Data Organization in Memory,” for more information.

lvxl provides a hint that the program may not need quadword addressed by EA again soon.

Note that on some implementations, the hint provided by the **lvxl** instruction and the corresponding hint provided by the Store Vector Indexed LRU (**stvxl**) instruction (see Section 5.2.1.2, “Transient Streams”) are applied to the entire cache block containing the specified quadword. On such implementations, the effect of the hint may be to cause that cache block to be considered a likely candidate for reuse when space is needed in the cache for a new block. Thus, on such implementations, the hint should be used with caution if the cache block containing the quadword also contains data that may be needed by the program in the near future. Also, the hint may be used before the last reference in a sequence of references to the quadword if the subsequent references are likely to occur sufficiently soon that the cache block containing the quadword is not likely to be displaced from the cache before the last reference. Figure 6-3 shows this instruction.

Other registers altered:

- None

mfvscr mfvscr

Move from Vector Status and Control Register

mfvscr **v**D Form VX

04	**v**D	0_0000	0000_0	1540
0 5	6 10	11 15	16 20	21 31

```
vD ← $^{96}0$ || (VSCR)
```

The contents of the VSCR are placed into **v**D.

Note that the programmer should assume that **mtvscr** and **mfvscr** take substantially longer to execute than other VX instructions

Other registers altered:

- None

mtvscr mtvscr

Move to Vector Status and Control Register

mtvscr **v**B Form VX

04	00_000	0_0000	**v**B	1604
0 5	6 10	11 15	16 20	21 31

```
    VSCR ← $(\mathbf{v}B)_{96:127}$
```

The contents of **v**B are placed into the VSCR.

Other registers altered:

- None

stvebx stvebx

Store Vector Element Byte Indexed

stvebx **v**S,**r**A,**r**B Form X

31	**v**S	A	B	135	0
0 5	6 10	11 15	16 20	21 30	31

- For 32-bit:

```
if rA=0 then b ← 0
else         b ← (rA)
EA ← b + (rB)
eb ← $EA_{28:31}$
if the processor is in big-endian mode
 then MEM(EA,1) ← $(vS)_{eb*8:(eb*8)+7}$
 else MEM(EA,1) ← $(vS)_{120-(eb*8):127-eb*8}$
```

— Let the EA be the sum (**r**A|0)+(**r**B). Let m = EA[28–31]; m is the byte offset of the byte in its aligned quadword in memory.

If the processor is in big-endian mode, byte m of **v**S is stored into the byte in memory addressed by EA. If the processor is in little-endian mode, byte (15-m) of **v**S is stored into the byte addressed by EA. Figure 6-2 shows how a store instruction is performed for a vector register.

Other registers altered:

- None

stvehx stvehx

Store Vector Element Half Word Indexed

stvehx **v**S,**r**A,**r**B Form X

31	**v**S	A	B	167	0
0 5	6 10	11 15	16 20	21 30	31

- For 32-bit:

```
if rA=0 then b ← 0
else         b ← (rA)
EA ← (b + (rB)) & (~0x1)
eb ← $EA_{28:31}$
if the processor is in big-endian mode
 then MEM(EA,2) ← $(vS)_{eb*8:(eb*8)+15}$
 else MEM(EA,2) ← $(vS)_{112-eb*8:127-(eb*8)}$
```

— Let the EA be the result of ANDing the sum (**r**A|0)+(**r**B) with ~0x1. Let m = EA[28–30]; m is the half-word offset of the half-word in its aligned quadword in memory.

If the processor is in big-endian mode, half-word m of **v**S is stored into the half-word addressed by EA. If the processor is in little-endian mode, half-word (7-m) of **v**S is stored into the half-word addressed by EA. Figure 6-2 shows how a store instruction is performed for a vector register.

Other registers altered:

- None

stvewx stvewx

Store Vector Element Word Indexed

stvewx **v**S,**r**A,**r**B Form X

31	**v**S	A	B	199	0
0 5	6 10	11 15	16 20	21 30	31

- For 32-bit:

```
if rA=0 then b ← 0
else         b ← (rA)
EA ← (b + (rB)) & 0xFFFF_FFFC
eb ← $EA_{28:31}$
if the processor is in big-endian mode
 then MEM(EA,4) ← $(vS)_{eb*8:(eb*8)+31}$
 else MEM(EA,4) ← $(vS)_{96-eb*8:127-(eb*8)}$
```

— Let the EA be the result of ANDing the sum (**r**A|0)+(**r**B) with 0xFFFF_FFFC. Let m = EA[28-29]; m is the word offset of the word in its aligned quadword in memory.

If the processor is in big-endian mode, word m of **v**S is stored into the word addressed by EA. If the processor is in little-endian mode, word (3-m) of **v**S is stored into the word addressed by EA. Figure 6-2 shows how a store instruction is performed for a vector register.

Other registers altered:

- None

stvx stvx

Store Vector Indexed

stvx **v**S,**r**A,**r**B (LRU = 0) Form X

0–5	6–10	11–15	16–20	21–30	31
31	**v**S	A	B	231	0

- For 32-bit:

```
if rA=0 then b ← 0
else b ← (rA)
EA ← (b + (rB)) & 0xFFFF_FFF0
if the processor is in big-endian mode
 then MEM(EA,16) ← (vS)
 else MEM(EA,16) ← $(vS)_{64:127} \| (vS)_{0:63}$
```

— Let the EA be the result of ANDing the sum (**r**A|0)+(**r**B) with 0xFFFF_FFF0.

If the processor is in big-endian mode, the contents of **v**S are stored into the quadword addressed by EA. If the processor is in little-endian mode, the contents of **v**S[64–127] are stored into the doubleword addressed by EA, and the contents of **v**S[0–63] are stored into the doubleword addressed by EA+8.

stvxl and **stvxlt** provide a hint that the quadword addressed by EA will probably not be needed again by the program in the near future.

Figure 6-2 shows how a store instruction is performed for a vector register.

Other registers altered:

- None

stvxl stvxl

Store Vector Indexed LRU

stvxl **v**S,**r**A,**r**B (LRU = 1) Form X

31	**v**S	A	B	487	0
0 5	6 10	11 15	16 20	21 30	31

- For 32-bit:

```
if rA=0 then b ← 0
else b ← (rA)
EA ← (b + (rB)) & 0xFFFF_FFF0
if the processor is in big-endian mode
 then MEM(EA,16) ← (vS)
 else MEM(EA,16) ← (vS)$_{64:127}$ || (vS)$_{0:63}$
```

— Let the EA be the result of ANDing the sum (**r**A|0)+(**r**B) with 0xFFFF_FFF0.

Let the EA be the result of ANDing the sum (**r**A|0)+(**r**B) with 0xFFFF_FFFF_FFFF_FFF0. If the processor is in big-endian mode, the contents of **v**S are stored into the quadword addressed by EA. If the processor is in little-endian mode, the contents of **v**S[64–127] are stored into the doubleword addressed by EA, and the contents of **v**S[0–63] are stored into the doubleword addressed by EA+8. The **stvxl** and **stvxlt** instructions provide a hint that the quad word addressed by EA will probably not be needed again by the program in the near future.

Note that on some implementations, the hint provided by the **stvxl** instruction (see Section 5.2.2, “Prioritizing Cache Block Replacement”) is applied to the entire cache block containing the specified quadword. On such implementations, the effect of the hint may be to cause that cache block to be considered a likely candidate for reuse when space is needed in the cache for a new block. Thus, on such implementations, the hint should be used with caution if the cache block containing the quadword also contains data that may be needed by the program in the near future. Also, the hint may be used before the last reference in a sequence of references to the quadword if the subsequent references are likely to occur sufficiently soon that the cache block containing the quadword is not likely to be displaced from the cache before the last reference. Figure 6-2 shows how a store instruction is performed on the vector registers.

Other registers altered:

- None

vaddcuw vaddcuw

Vector Add Carryout Unsigned Word

vaddcuw **v**D,**v**A,**v**B Form VX

04	**v**D	**v**A	**v**B	384
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 32

    aop$_{0:32}$← ZeroExtend((vA)$_{i:i+31}$,33)
    bop$_{0:32}$← ZeroExtend((vB)$_{i:i+31}$,33)
    temp$_{0:32}$← aop$_{0:32}$ +$_{int}$ bop$_{0:32}$
    vD$_{i:i+31}$← ZeroExtend(temp$_0$,32)

end
```

Each unsigned-integer word element in **v**A is added to the corresponding unsigned-integer word element in **v**B. The carry out of bit 0 of the 32-bit sum is zero-extended to 32 bits and placed into the corresponding word element of **v**D.

Other registers altered:

- None

Figure 6-5 shows the usage of the **vaddcuw** instruction. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits long.

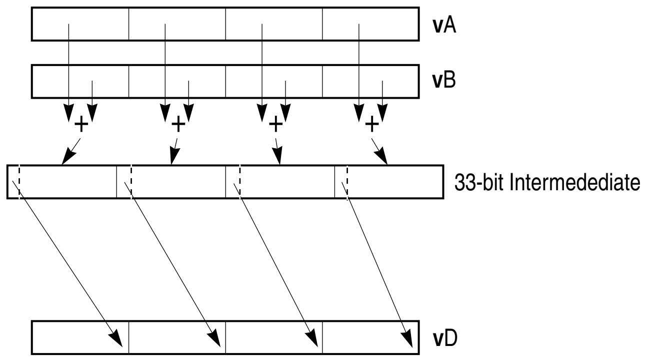


Figure 6-5. vaddcuw—Determine Carries of Four Unsigned Integer Adds (32-Bit)

vaddfp vaddfp

Vector Add Floating Point

vaddfp **v**D,**v**A,**v**B Form VX

04	**v**D	**v**A	**v**B	10
0 5	6 10	11 15	16 20	21 31

```
do i = 0,127,32
    (vD)$_{i:i+31}$ ← RndToNearFP32((vA)$_{i:i+31}$ +$_{fp}$ (vB)$_{i:i+31}$)
end
```

The four 32-bit floating-point values in **v**A are added to the four 32-bit floating-point values in **v**B. The four intermediate results are rounded and placed in VD.

If VSCR[NJ] = 1, every denormalized operand element is truncated to a 0 of the same sign before the operation is carried out, and each denormalized result element truncates to a 0 of the same sign.

Other registers altered:

- None

Figure 6-6 shows the usage of the **vaddfp** instruction. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits long.

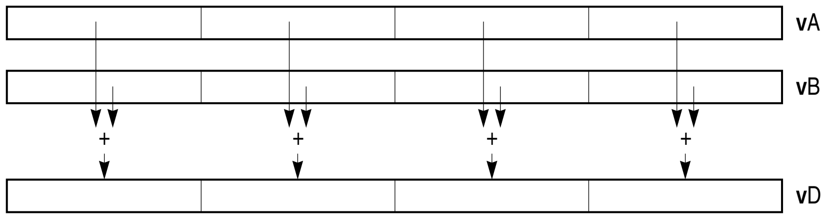


Figure 6-6. vaddfp—Add Four Floating-Point Elements (32-Bit)

vaddsbs vaddsbs

Vector Add Signed Byte Saturate

vaddsbs **v**D,**v**A,**v**B Form VX

04	**v**D	**v**A	**v**B	768
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 8
    aop$_{0:8}$← SignExtend((vA)$_{i:i+7}$,9)
    bop$_{0:8}$← SignExtend((vB)$_{i:i+7}$,9)
    temp$_{0:8}$← aop$_{0:8}$ +$_{int}$ bop$_{0:8}$
    vD$_{i:i+7}$← SItoSIsat(temp$_{0:8}$,8)
end
```

Each element of **vaddsbs** is a byte.

Each signed-integer element in **v**A is added to the corresponding signed-integer element in **v**B.

If the sum is greater than (2^7-1) it saturates to (2^7-1) and if it is less than -2^7 it saturates to -2^7. If saturation occurs, the SAT bit is set.

The signed-integer result is placed into the corresponding element of **v**D.

Other registers altered:

- Vector status and control register (VSCR):

  Affected: SAT

Figure 6-7 shows the usage of the **vaddsbs** instruction. Each of the sixteen elements in the vectors, **v**A, **v**B, and **v**D, is 8 bits long.

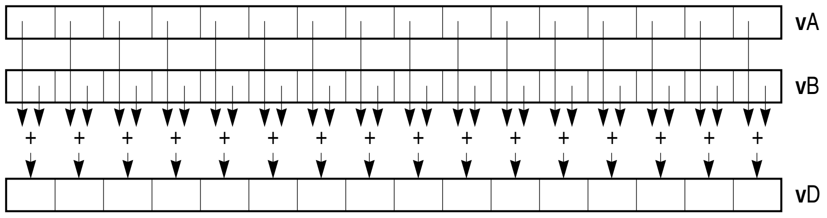


Figure 6-7. vaddsbs—Add Saturating Sixteen Signed Integer Elements (8-Bit)

vaddshs vaddshs

Vector Add Signed Half Word Saturate

vaddshs **v**D,**v**A,**v**B Form VX

04	**v**D	**v**A	**v**B	832
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 16
    $aop_{0:16}$ ← SignExtend((vA)$_{i:i+15}$,16)
    $bop_{0:16}$ ← SignExtend((vB)$_{i:i+15}$,16)
    $temp_{0:16}$ ← $aop_{0:16}$ $+_{int}$ $bop_{0:16}$
    vD$_{i:i+15}$ ← SItoSIsat($temp_{0:16}$,16)
end
```

Each element of **vaddshs** is a half word.

Each signed-integer element in **v**A is added to the corresponding signed-integer element in **v**B.

If the sum is greater than (2^{15}-1) it saturates to (2^{15}-1) and if it is less than -2^{15} it saturates to -2^{15}. If saturation occurs, the SAT bit is set.

The signed-integer result is placed into the corresponding element of **v**D.

Other registers altered:

- Vector status and control register (VSCR):
  Affected: SAT

Figure 6-8 shows the usage of the **vaddshs** instruction. Each of the eight elements in the vectors, **v**A, **v**B, and **v**D, is 16 bits long.

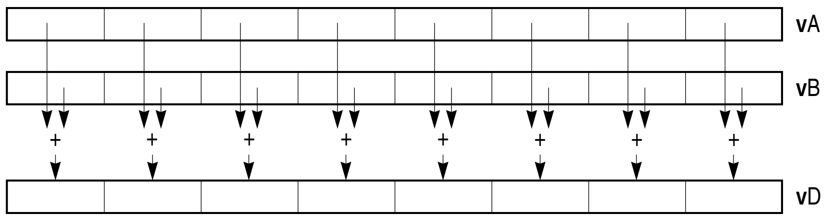


Figure 6-8. vaddshs— Add Saturating Eight Signed Integer Elements (16-Bit)

vaddsws vaddsws

Vector Add Signed Word Saturate

vaddsws **v**D,**v**A,**v**B Form VX

04	vD	vA	vB	896
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 32
    aop_{0:32} ← SignExtend((vA)_{i:i+31},33)
    bop_{0:32} ← SignExtend((vB)_{i:i+31},33)
    temp_{0:32} ← aop_{0:32} +_{int} bop_{0:32}
    vD_{i:i+31} ← SItoSIsat(temp_{0:32},32)
end
```

Each element of **vaddsws** is a word.

Each signed-integer element in **v**A is added to the corresponding signed-integer element in **v**B.

If the sum is greater than (2^{31}-1) it saturates to (2^{31}-1) and if it is less than (-2^{31}) it saturates to (-2^{31}). If saturation occurs, the SAT bit is set.

The signed-integer result is placed into the corresponding element of **v**D.

Other registers altered:

- Vector status and control register (VSCR):

  Affected: SAT

Figure 6-9 shows the usage of the **vaddsws** instruction. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits long.

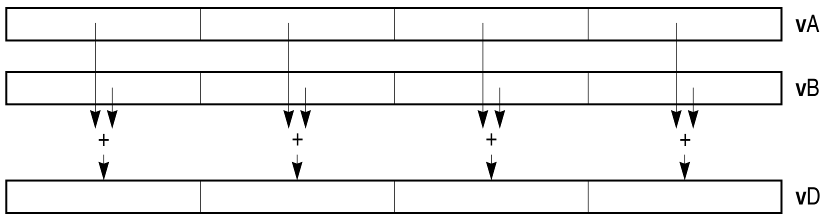


Figure 6-9. vaddsws—Add Saturating Four Signed Integer Elements (32-Bit)

vaddubm vaddubm

Vector Add Unsigned Byte Modulo

vaddubm **v**D,**v**A,**v**B Form VX

04	**v**D	**v**A	**v**B	0
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 8
    vD_{i:i+7} ← (vA)_{i:i+7} +_{int} (vB)_{i:i+7}
end
```

Each element of **vaddubm** is a byte.

Each integer element in **v**A is modulo added to the corresponding integer element in **v**B. The integer result is placed into the corresponding element of **v**D.

Note that the **vaddubm** instruction can be used for unsigned or signed integers.

Other registers altered:

- None

Figure 6-10 shows the **vaddubm** instruction usage. Each of the sixteen elements in the vectors, **v**A, **v**B, and **v**D, is 8 bits long.

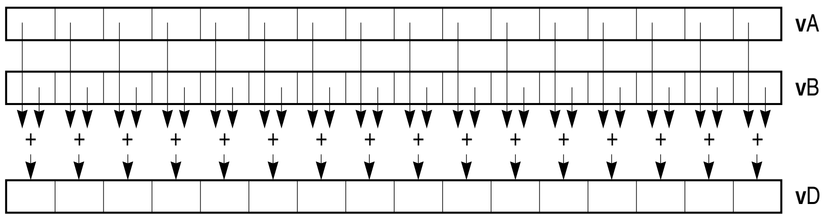


Figure 6-10. vaddubm—Add Sixteen Integer Elements (8-Bit)

vaddubs vaddubs

Vector Add Unsigned Byte Saturate

vaddubs **v**D,**v**A,**v**B Form VX

04	**v**D	**v**A	**v**B	512
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 8
    $aop_{0:8} \leftarrow$ ZeroExtend((vA)$_{i:i+7}$,9)
    $bop_{0:8} \leftarrow$ ZeroExtend((vB)$_{i:i+7}$,9)
    $temp_{0:8} \leftarrow aop_{0:8} +_{int} bop_{0:8}$
    vD$_{i:i+7} \leftarrow$ UItoUIsat(temp$_{0:8}$,8)
end
```

Each element of **vaddubs** is a byte.

Each unsigned-integer element in **v**A is added to the corresponding unsigned-integer element in **v**B.

If the sum is greater than (2^8-1) it saturates to (2^8-1) and the SAT bit is set.

The unsigned-integer result is placed into the corresponding element of **v**D.

Other registers altered:

- Vector status and control register (VSCR):
  Affected: SAT

Figure 6-11 shows the usage of the **vaddubs** instruction. Each of the sixteen elements in the vectors, **v**A, **v**B, and **v**D, is 8 bits long.

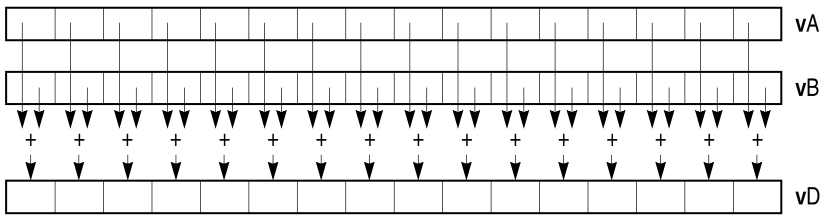


Figure 6-11. vaddubs—Add Saturating Sixteen Unsigned Integer Elements (8-Bit)

vadduhm vadduhm

Vector Add Unsigned Half Word Modulo

vadduhm **v**D,**v**A,**v**B Form VX

04	**v**D	**v**A	**v**B	64
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 16
    vD_{i:i+15} ← (vA)_{i:i+15} +_int (vB)_{i:i+15}
end
```

Each element of **vadduhm** is a half word.

Each integer element in **v**A is added to the corresponding integer element in **v**B. The integer result is placed into the corresponding element of **v**D.

Note that the **vadduhm** instruction can be used for unsigned or signed integers.

Other registers altered:

- None

Figure 6-12 shows the usage of the **vadduhm** instruction. Each of the eight elements in the vectors, **v**A, **v**B, and **v**D, is 16 bits long.

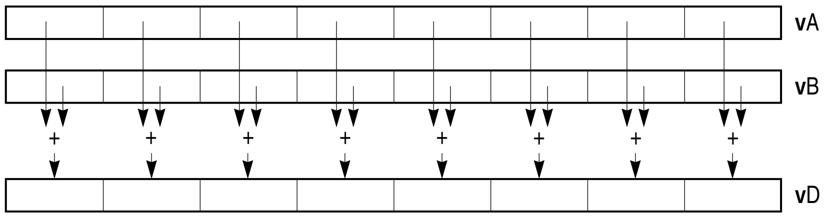


Figure 6-12. vadduhm—Add Eight Integer Elements (16-Bit)

vadduhs vadduhs

Vector Add Unsigned Half Word Saturate

vadduhs **v**D,**v**A,**v**B Form VX

04	**v**D	**v**A	**v**B	576
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 16

    $aop_{0:16} \leftarrow$ ZeroExtend((vA)$_{i:i+15}$,17)
    $bop_{0:16} \leftarrow$ ZeroExtend((vB)$_{i:i+15}$,17)
    $temp_{0:16} \leftarrow aop_{0:16} +_{int} bop_{0:16}$
    vD$_{i:i+15} \leftarrow$ UItoUIsat($temp_{0:16}$,16)

end
```

Each element of **vadduhs** is a half word.

Each unsigned-integer element in **v**A is added to the corresponding unsigned-integer element in **v**B.

If the sum is greater than (2^{16}-1) it saturates to (2^{16}-1) and the SAT bit is set.

The unsigned-integer result is placed into the corresponding element of **v**D.

Other registers altered:

- Vector status and control register (VSCR):

  Affected: SAT

Figure 6-13 shows the usage of the **vadduhs** instruction. Each of the eight elements in the vectors, **v**A, **v**B, and **v**D, is 16 bits long.

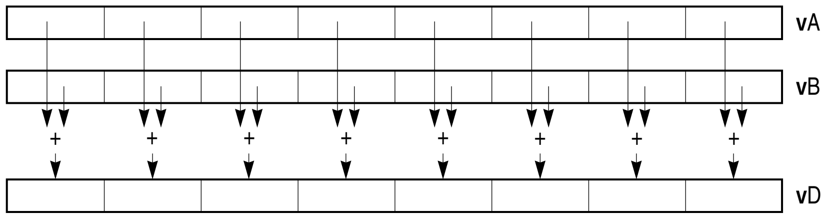


Figure 6-13. vadduhs—Add Saturating Eight Unsigned Integer Elements (16-Bit)

vadduwm vadduwm

Vector Add Unsigned Word Modulo

vadduwm **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	128
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 32
    $vD_{i:i+31} \leftarrow (vA)_{i:i+31} +_{int} (vB)_{i:i+31}$
end
```

Each element of **vadduwm** is a word.

Each integer element in **v**A is modulo added to the corresponding integer element in **v**B. The integer result is placed into the corresponding element of **v**D.

Note that the **vadduwm** instruction can be used for unsigned or signed integers.

Other registers altered:

- None

Form:

- VX

Figure 6-14 shows the usage of the **vadduwm** instruction. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits long.

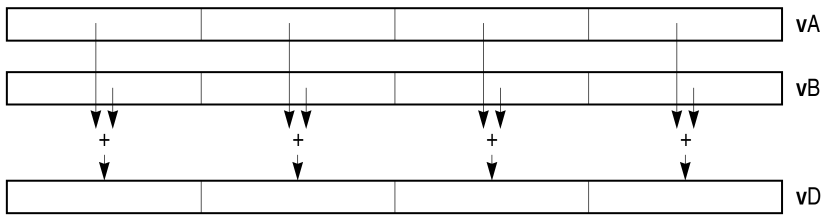


Figure 6-14. vadduwm—Add Four Integer Elements (32-Bit)

vadduws vadduws

Vector Add Unsigned Word Saturate

vadduws **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	640
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 3
    $aop_{0:32} \leftarrow$ ZeroExtend((vA)$_{i:i+31}$,33)
    $bop_{0:32} \leftarrow$ ZeroExtend((vB)$_{i:i+31}$,33)
    $temp_{0:32} \leftarrow aop_{0:32} +_{int} bop_{0:32}$
    $vD_{i:i+31} \leftarrow$ UItoUIsat($temp_{0:32}$,32)
end
```

Each element of **vadduws** is a word.

Each unsigned-integer element in **v**A is added to the corresponding unsigned-integer element in **v**B.

If the sum is greater than (2^{32}-1) it saturates to (2^{32}-1) and the SAT bit is set.

The unsigned-integer result is placed into the corresponding element of **v**D.

Other registers altered:

- Vector status and control register (VSCR):
  Affected: SAT

Figure 6-15 shows the usage of the **vadduws** instruction. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits long.

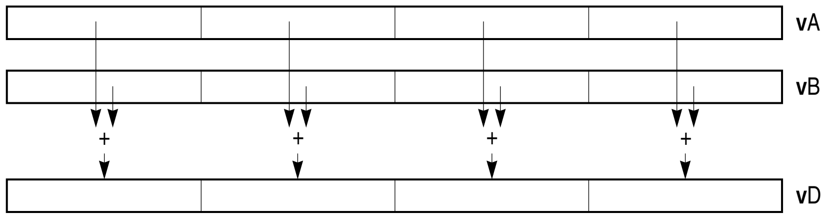


Figure 6-15. vadduws—Add Saturating Four Unsigned Integer Elements (32-Bit)

vand vand

Vector Logical AND

vand **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	1028
0 5	6 10	11 15	16 20	21 31

```
vD ← (vA) & (vB)
```

The contents of **v**A are bitwise ANDed with the contents of **v**B and the result is placed into **v**D.

Other registers altered:

- None

Figure 6-16 shows usage of the **vand** instruction.

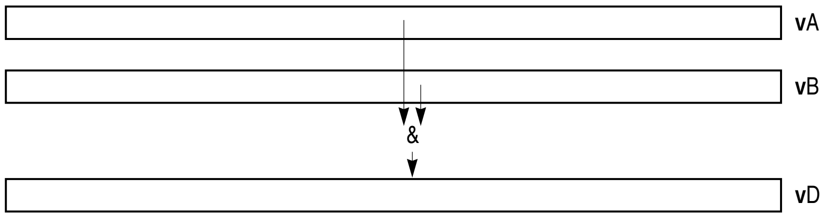


Figure 6-16. vand—Logical Bitwise AND

vandc vandc

Vector Logical AND with Complement

vandc **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	1092
0 5	6 10	11 15	16 20	21 31

```
vD ← (vA) & ¬(vB)
```

The contents of **v**A are ANDed with the one’s complement of the contents of **v**B and the result is placed into **v**D.

Other registers altered:

- None

Figure 6-16 shows usage of the **vandc** instruction.

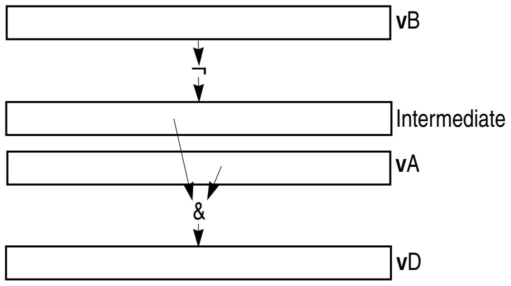


Figure 6-17. vand—Logical Bitwise AND with Complement

vavgsb vavgsb

Vector Average Signed Byte

vavgsb vD,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	1282
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 8

    $aop_{0:8} \leftarrow$ SignExtend((vA)$_{i:i+7}$,9)
    $bop_{0:8} \leftarrow$ SignExtend((vB)$_{i:i+7}$,9)
    $temp_{0:8} \leftarrow aop_{0:8} +_{int} bop_{0:8} +_{int} 1$
    vD$_{i:i+7} \leftarrow temp_{0:7}$

end
```

Each element of **vavgsb** is a byte.

Each signed-integer byte element in **v**A is added to the corresponding signed-integer byte element in **v**B, producing an 9-Bit signed-integer sum. The sum is incremented by 1. The high-order 8 bits of the result are placed into the corresponding element of **v**D.

Other registers altered:

- None

Figure 6-18 shows the usage of the **vavgsb** instruction. Each of the sixteen elements in the vectors, **v**A, **v**B, and **v**D, is 8 bits long.

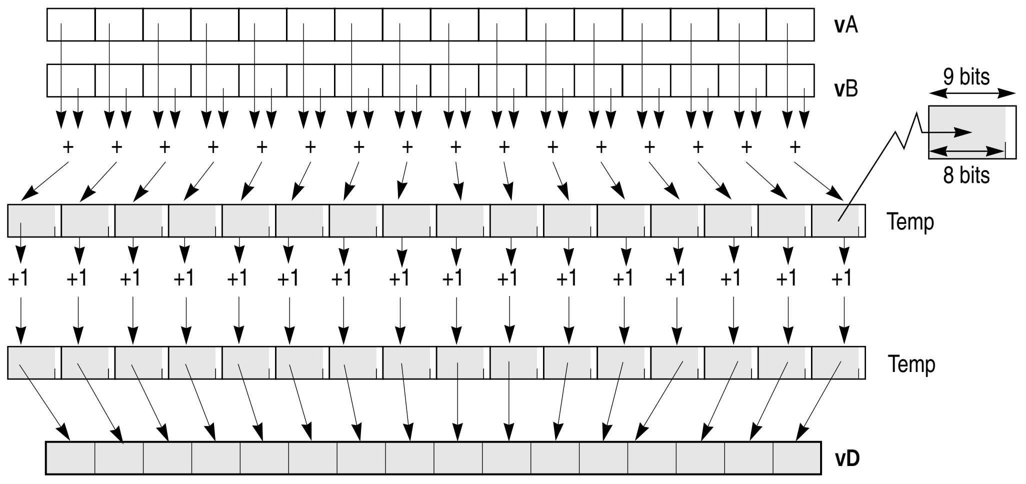


Figure 6-18. vavgsb— Average Sixteen Signed Integer Elements (8-Bit)

vavgsh vavgsh

Vector Average Signed Half Word

vavgsh **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	1346
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 16
    aop_{0:16}← SignExtend((vA)_{i:i+15},17)
    bop_{0:16}← SignExtend((vB)_{i:i+15},17)
    temp_{0:16}← aop_{0:15} +_{int} bop_{0:15} +_{int} 1
    vD_{i:i+15}← temp_{0:15}
end
```

Each element of **vavgsh** is a half word.

Each signed-integer element in **v**A is added to the corresponding signed-integer element in **v**B, producing an 17-bit signed-integer sum. The sum is incremented by 1. The high-order 16 bits of the result are placed into the corresponding element of **v**D.

Other registers altered:

- None

Figure 6-19 shows the usage of the **vavgsh** instruction. Each of the eight elements in the vectors, **v**A, **v**B, and **v**D, is 16 bits long.

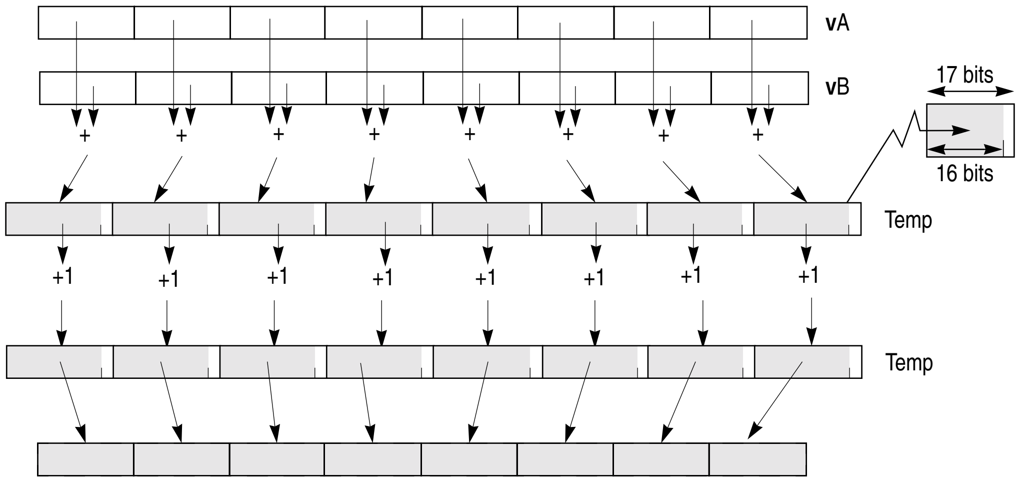


Figure 6-19. vavgsh—Average Eight Signed Integer Elements (16-bits)

vavgsw vavgsw

Vector Average Signed Word

vavgsw **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	1410
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 32
    $aop_{0:32} \leftarrow$ SignExtend((vA)$_{i:i+31}$,33)
    $bop_{0:32} \leftarrow$ SignExtend((vB)$_{i:i+31}$,33)
    $temp_{0:32} \leftarrow aop_{0:32} +_{int} bop_{0:32} +_{int} 1$
    vD$_{i:i+31} \leftarrow temp_{0:31}$
end
```

Each element of **vavgsw** is a word.

Each signed-integer element in **v**A is added to the corresponding signed-integer element in **v**B, producing an 33-bit signed-integer sum. The sum is incremented by 1. The high-order 32 bits of the result are placed into the corresponding element of **v**D.

Other registers altered:

- None

Figure 6-20 shows the usage of the **vavgsw** instruction. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits long.

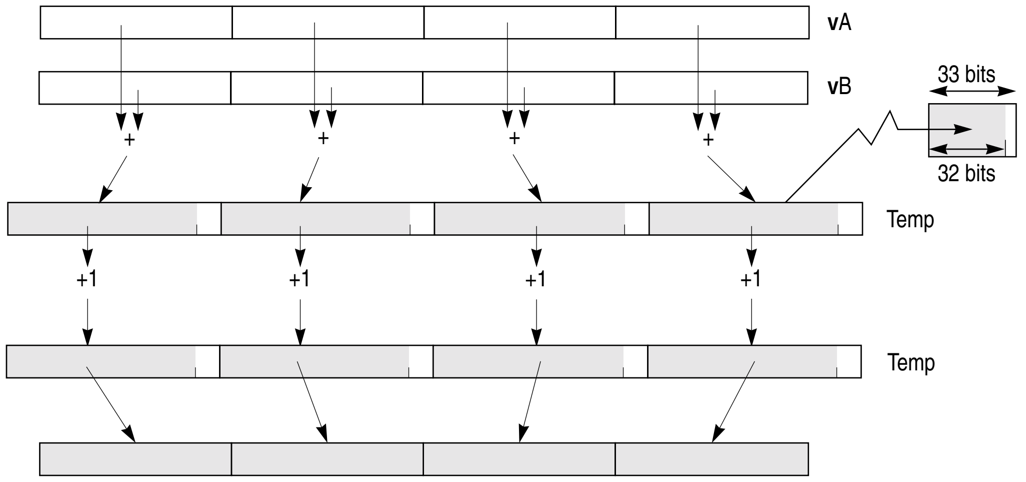


Figure 6-20. vavgsw— Average Four Signed Integer Elements (32-Bit)

vavgub vavgub

Vector Average Unsigned Byte

vavgub **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	1026
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 8
    $aop_{0:8} \leftarrow$ ZeroExtend(($\mathbf{v}$A)$_{i:i+7}$,9)
    $bop_{0:n} \leftarrow$ ZeroExtend(($\mathbf{v}$B)$_{i:i+71}$,9)
    $temp_{0:n} \leftarrow aop_{0:8} +_{int} bop_{0:8} +_{int} 1$
    $\mathbf{v}D_{i:i+7} \leftarrow temp_{0:7}$
end
```

Each element of **vavgub** is a byte.

Each unsigned-integer element in **v**A is added to the corresponding unsigned-integer element in **v**B, producing an 9-bit unsigned-integer sum. The sum is incremented by 1. The high-order 8 bits of the result are placed into the corresponding element of **v**D.

Other registers altered:

- None

Figure 6-21 shows the usage of the **vavgub** instruction. Each of the sixteen elements in the vectors, **v**A, **v**B, and **v**D, is 8 bits long.

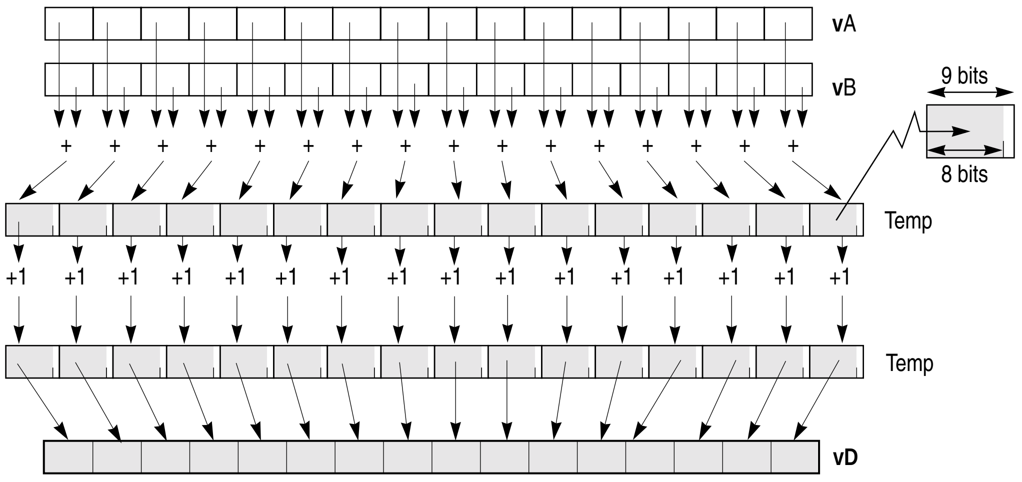


Figure 6-21. vavgub—Average Sixteen Unsigned Integer Elements (8-bits)

vavguh vavguh

Vector Average Unsigned Half Word

vavguh **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	1090
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 16
    aop$_{0:16}$← ZeroExtend((vA)$_{i:i+15}$,17)
    bop$_{0:16}$← ZeroExtend((vB)$_{i:i+15}$,17)
    temp$_{0:16}$← aop$_{0:16}$ +$_{int}$ bop$_{0:16}$ +$_{int}$ 1
    vD$_{i:i+15}$← temp$_{0:15}$
end
```

Each element of **vavguh** is a half word.

Each unsigned-integer element in **v**A is added to the corresponding unsigned-integer element in **v**B, producing a 17-bit unsigned-integer. The sum is incremented by 1. The high-order 16 bits of the result are placed into the corresponding element of **v**D.

Other registers altered:

- None

Figure 6-22 shows the usage of the **vavgsh** instruction. Each of the eight elements in the vectors, **v**A, **v**B, and **v**D, is 16 bits long.

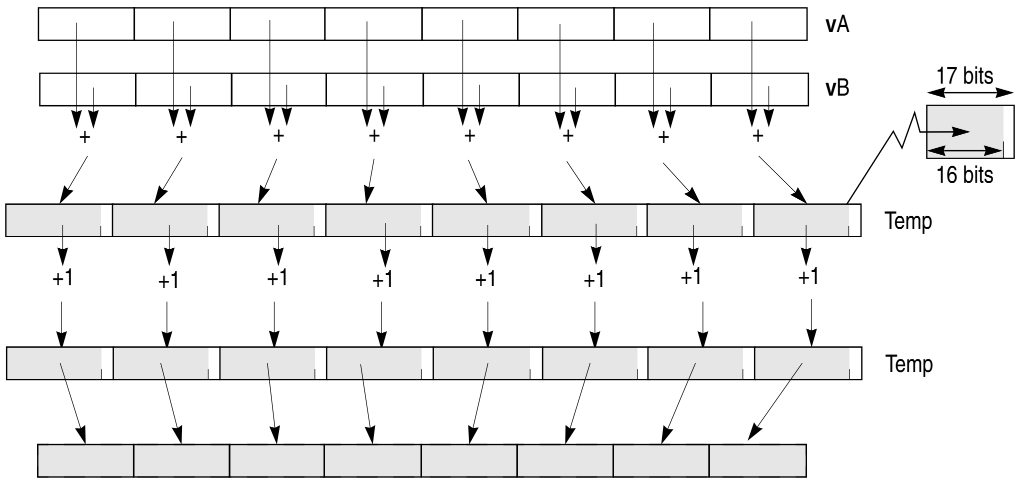


Figure 6-22. vavgsh— Average Eight Signed Integer Elements (16-Bit)

vavguw vavguw

Vector Average Unsigned Word

vavguw **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	1154
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 32
    $aop_{0:32} \leftarrow$ ZeroExtend((vA)$_{i:i+31}$,33)
    $bop_{0:32} \leftarrow$ ZeroExtend((vB)$_{i:i+31}$,33)
    $temp_{0:32} \leftarrow aop_{0:32} +_{int} bop_{0:32} +_{int} 1$
    $vD_{i:i+31} \leftarrow temp_{0:31}$
end
```

Each element of **vavguw** is a word.

Each unsigned-integer element in **v**A is added to the corresponding unsigned-integer element in **v**B, producing an 33-bit unsigned-integer sum. The sum is incremented by 1. The high-order 32 bits of the result are placed into the corresponding element of **v**D.

Other registers altered:

- None

Figure 6-23 shows the usage of the **vavguw** instruction. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits long.

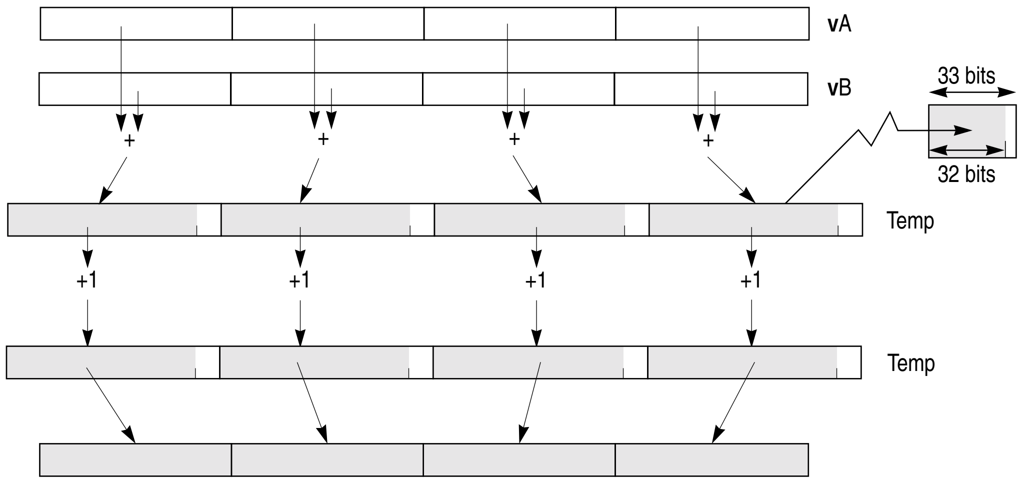


Figure 6-23. vavguw—Average Four Unsigned Integer Elements (32-Bit)

vcfsx vcfsx

Vector Convert from Signed Fixed-Point Word

vcfsx **v**D,**v**B,UIMM Form: VX

04	**v**D	UIMM	**v**B	842
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 32
    $vD_{i:i+31} \leftarrow$ CnvtSI32ToFP32((vB)$_{i:i+31}$) $\div_{fp}$ $2^{UIMM}$
end
```

Each signed fixed-point integer word element in **v**B is converted to the nearest single-precision floating-point value. The result is divided by 2^{UIMM} (UIMM = Unsigned immediate value) and placed into the corresponding word element of **v**D.

Other registers altered:

- None

Figure 6-24 shows the usage of the **vcfsx** instruction. Each of the four elements in the vectors **v**B and **v**D is 32 bits long.

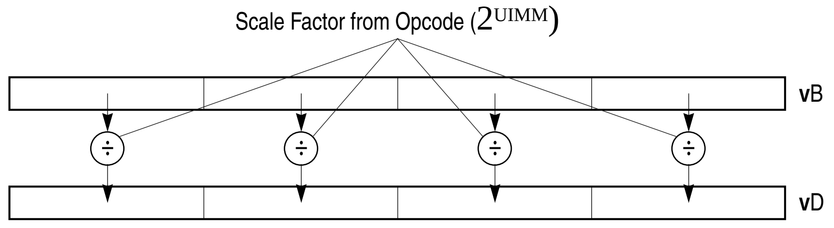


Figure 6-24. vcfsx—Convert Four Signed Integer Elements to Four Floating-Point Elements (32-Bit)

vcfux vcfux

Vector Convert from Unsigned Fixed-Point Word

vcfux **v**D,**v**B,UIMM Form: VX

04	**v**D	UIMM	**v**B	778
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 32
    $vD_{i:i+31} \leftarrow$ CnvtUI32ToFP32((vB)$_{i:i+31}$) $\div_{fp}$ $2^{UIMM}$
end
```

Each unsigned fixed-point integer word element in **v**B is converted to the nearest single-precision floating-point value. The result is divided by 2^{UIMM} and placed into the corresponding word element of **v**D.

Other registers altered:

- None

Figure 6-25 shows the usage of the **vcfux** instruction. Each of the four elements in the vectors **v**B and **v**D is 32 bits long.

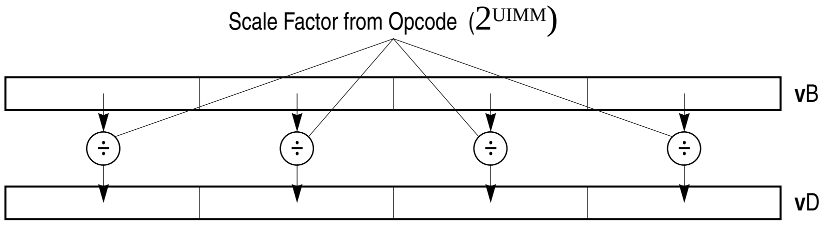


Figure 6-25. vcfux—Convert Four Unsigned Integer Elements to Four Floating-Point Elements (32-Bit)

vcmpbfp*x* vcmpbfp*x*

Vector Compare Bounds Floating Point

vcmpbfp	**v**D,**v**A,**v**B	(Rc = 0)	Form: VXR
vcmpbfp.	**v**D,**v**A,**v**B	(Rc = 1)	

04	**v**D	**v**A	**v**B	Rc	966
0 5	6 10	11 15	16 20	21	22 31

```
do i=0 to 127 by 32
    le ← $((\mathbf{v}A)_{i:i+31} \le_{fp} (\mathbf{v}B)_{i:i+31})$
    ge ← $((\mathbf{v}A)_{i:i+31} \ge_{fp} -(\mathbf{v}B)_{i:i+31})$
    $\mathbf{v}D_{i:i+31}$ ← ¬le || ¬ge || $^{30}0$
end
if Rc=1 then do
    ib ← ($\mathbf{v}D = {}^{128}0$)
    $CR_{24:27}$ ← 0b00 || ib || 0b0
end
```

Each single-precision word element in **v**A is compared to the corresponding element in **v**B. A 2-bit value is formed that indicates whether the element in **v**A is within the bounds specified by the element in **v**B, as follows.

Bit 0 of the 2-bit value is zero if the element in **v**A is less than or equal to the element in **v**B, and is one otherwise. Bit 1 of the 2-bit value is zero if the element in **v**A is greater than or equal to the negative of the element in **v**B, and is one otherwise.

The 2-bit value is placed into the high-order two bits of the corresponding word element (bits 0–1 for word element 0, bits 32–33 for word element 1, bits 64–65 for word element 2, bits 96–97 for word element 3) of **v**D and the remaining bits of the element are cleared.

If Rc=1, CR Field 6 is set to indicate whether all four elements in **v**A are within the bounds specified by the corresponding element in **v**B, as follows.

- CR6 = 0b00 || all_within_bounds || 0

Note that if any single-precision floating-point word element in **v**B is negative; the corresponding element in **v**A is out of bounds. Note that if a **v**A or a **v**B element is a NaN, the two high order bits of the corresponding result will both have the value 1.

If VSCR[NJ] = 1, every denormalized operand element is truncated to 0 before the comparison is made.

Other registers altered:

- Condition register (CR6):
  Affected: Bit 2 (if Rc = 1)

Figure 6-26 shows the usage of the **vcmpbfp** instruction. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits long.

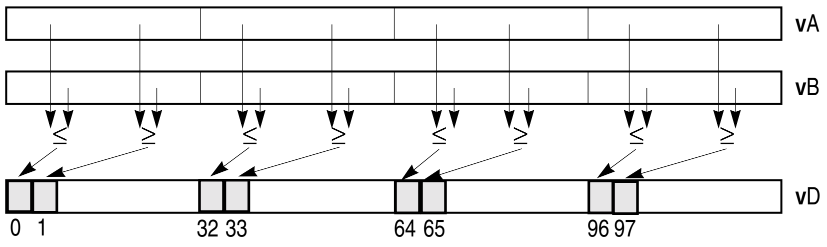


Figure 6-26. vcmpbfp—Compare Bounds of Four Floating-Point Elements (32-Bit)

vcmpeqfp*x* vcmpeqfp*x*

Vector Compare Equal-to-Floating Point

vcmpeqfp	**v**D,**v**A,**v**B	Form: VXR
vcmpeqfp.	**v**D,**v**A,**v**B	

04	**v**D	**v**A	**v**B	Rc	198
0 5	6 10	11 15	16 20	21	22 31

```
do i=0 to 127 by 32
    if (vA)$_{i:i+31}$ =$_{fp}$ (vB)$_{i:i+31}$
        then vD$_{i:i+31}$ ← 0xFFFF_FFFF
        else vD$_{i:i+31}$ ← 0x0000_0000
end
if Rc=1 then do
    t ← (vD = $^{128}$1)
    f ← (vD = $^{128}$0)
    CR$_{24:27}$ ← t || 0b0 || f || 0b0
end
```

Each single-precision floating-point word element in **v**A is compared to the corresponding single-precision floating-point word element in **v**B. The corresponding word element in **v**D is set to all 1s if the element in **v**A is equal to the element in **v**B, and is cleared to all 0s otherwise.

If Rc = 1. CR6 filed is set according to all, some, or none of the elements pairs compare equal.

- CR6 = all_equal || 0b0 || none_equal || 0b0

Note that if a **v**A or **v**B element is a NaN, the corresponding result will be 0x0000_0000.

Other registers altered:

- Condition register (CR6):
  Affected: Bits 0-3 (if Rc = 1)

Figure 6-27 shows the usage of the **vcmpeqfp** instruction. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits long.

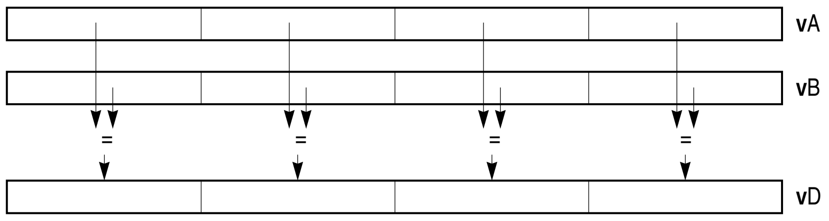


Figure 6-27. vcmpeqfp—Compare Equal of Four Floating-Point Elements (32-Bit)

vcmpequb*x* vcmpequb*x*

Vector Compare Equal-to Unsigned Byte

vcmpequb	**v**D,**v**A,**v**B	Form: VXR
vcmpequb.	**v**D,**v**A,**v**B	

04	**v**D	**v**A	**v**B	Rc	6
0 5	6 10	11 15	16 20	21	22 31

```
do i=0 to 127 by 8
    if $(\mathbf{v}A)_{i:i+7} =_{int} (\mathbf{v}B)_{i:i+7}$
    then $\mathbf{v}D_{i:i+7} \leftarrow {}^{8}1$
    else $\mathbf{v}D_{i:i+7} \leftarrow {}^{8}0$
end
if Rc=1 then do
    t ← (vD = $^{128}1$)
    f ← (vD = $^{128}0$)
    CR[24:27] ← t || 0b0 || f || 0b0
end
```

Each element of **vcmpequb** is a byte.

Each integer element in **v**A is compared to the corresponding integer element in **v**B. The corresponding element in **v**D is set to all 1s if the element in **v**A is equal to the element in **v**B, and is cleared to all 0s otherwise.

The CR6 is set according to whether all, some, or none of the elements compare equal.

- CR6 = all_equal || 0b0 || none_equal || 0b0

Note that **vcmpequb**[.] can be used for unsigned or signed integers.

Other registers altered:

- Condition register (CR6):

  Affected: Bits 0–3 (if Rc = 1)

Figure 6-28 shows the usage of the **vcmpequb** instruction. Each of the sixteen elements in the vectors, **v**A, **v**B, and **v**D, is 8 bits long.

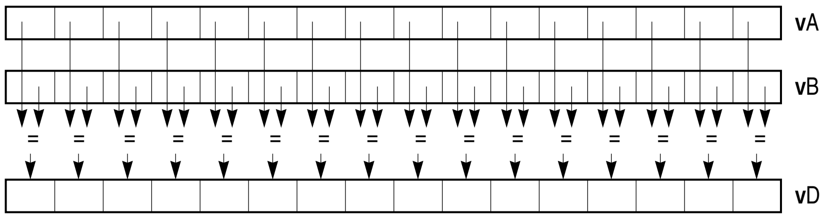


Figure 6-28. vcmpequb—Compare Equal of Sixteen Integer Elements (8-bits)

vcmpequh*x* vcmpequh*x*

Vector Compare Equal-to Unsigned Half Word

vcmpequh	**v**D,**v**A,**v**B	Form: VXR
vcmpequh.	**v**D,**v**A,**v**B	

04	**v**D	**v**A	**v**B	Rc	70
0 5	6 10	11 15	16 20	21	22 31

```
do i=0 to 127 by 16
    if (vA)$_{i:i+15}$ =$_{int}$ (vB)$_{i:i+15}$
    then vD$_{i:i+15}$ ← $^{16}1$
    else vD$_{i:i+15}$ ← $^{16}0$
end
if Rc=1 then do
    t ← (vD = $^{128}1$)
    f ← (vD = $^{128}0$)
    CR[24:27] ← t || 0b0 || f || 0b0
end
```

Each element of **vcmpequh** is a half word.

Each integer element in **v**A is compared to the corresponding integer element in **v**B. The corresponding element in **v**D is set to all 1s if the element in **v**A is equal to the element in **v**B, and is cleared to all 0s otherwise.

The CR6 is set according to whether all, some, or none of the elements compare equal.

- CR6 = all_equal || 0b0 || none_equal || 0b0.

Note that **vcmpequh**[**.**] can be used for unsigned or signed integers.

Other registers altered:

- Condition register (CR6):

  Affected: Bits 0–3 (if Rc = 1)

Figure 6-29 shows the usage of the **vcmpequh** instruction. Each of the eight elements in the vectors, **v**A, **v**B, and **v**D, is 16 bits long.

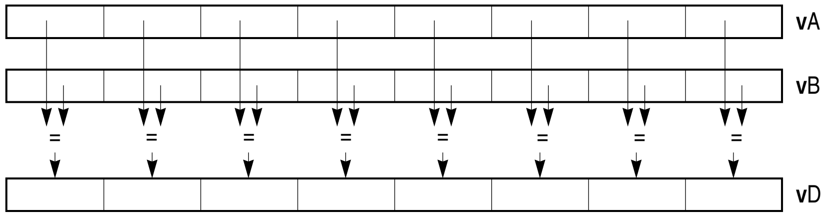


Figure 6-29. vcmpequh—Compare Equal of Eight Integer Elements (16-Bit)

vcmpequw*x* vcmpequw*x*

Vector Compare Equal-to Unsigned Word

vcmpequw	**v**D,**v**A,**v**B	Form: VXR
vcmpequw.	**v**D,**v**A,**v**B	

04	**v**D	**v**A	**v**B	Rc	134
0 5	6 10	11 15	16 20	21	22 31

```
do i=0 to 127 by 32
    if (vA)_{i:i+311} =_int (vB)_{i:i+31}
        then vD_{i:i+31} ← ^n1
        else vD_{i:i+31} ← ^n0
end
if Rc=1 then do
    t ← (vD = ^{128}1)
    f ← (vD = ^{128}0)
    CR[24:27] ← t || 0b0 || f || 0b0
end
```

Each element of **vcmpequw** is a word.

Each integer element in **v**A is compared to the corresponding integer element in **v**B. The corresponding element in **v**D is set to all 1s if the element in **v**A is equal to the element in **v**B, and is cleared to all 0s otherwise.

The CR6 is set according to whether all, some, or none of the elements compare equal.

- CR6 = all_equal || 0b0 || none_equal || 0b0

Note that **vcmpequw**[.] can be used for unsigned or signed integers.

Other registers altered:

- Condition register (CR6):

  Affected: Bits 0-3 (if Rc = 1)

Figure 6-30 shows the usage of the **vcmpequw** instruction. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits long.

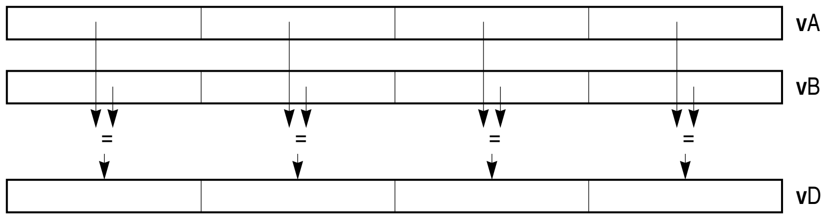


Figure 6-30. vcmpequw—Compare Equal of Four Integer Elements (32-Bit)

vcmpgefp*x* vcmpgefp*x*

Vector Compare Greater-Than-or-Equal-to Floating Point

vcmpgefp	**v**D,**v**A,**v**B	(Rc = 0)	Form: VXR
vcmpgefp.	**v**D,**v**A,**v**B	(Rc = 1)	

04	**v**D	**v**A	**v**B	Rc	454
0 5	6 10	11 15	16 20	21	22 31

```
do i=0 to 127 by 32
    if (vA)_{i:i+31} ≥_{fp} (vB)_{i:i+31}
    then vD_{i:i+31} ← 0xFFFF_FFFF
    else vD_{i:i+31} ← 0x0000_0000
end
if Rc=1 then do
    t ← (vD = ^{128}1)
    f ← (vD = ^{128}0)
    CR_{24:27} ← t || 0b0 || f || 0b0
end
```

Each single-precision floating-point word element in **v**A is compared to the corresponding single-precision floating-point word element in **v**B. The corresponding word element in **v**D is set to all 1s if the element in **v**A is greater than or equal to the element in **v**B, and is cleared to all 0s otherwise.

If Rc = 1, CR6 is set according to all_greater_or_equal || some_greater_or_equal || none_great_or_equal.

CR6 = all_greater_or_equal || 0b0 || none greater_or_equal || 0b0.

Note that if a **v**A or **v**B element is a NaN, the corresponding results will be 0x0000_0000.

Other registers altered:

- Condition register (CR6):

  Affected: Bits 0-3 (if Rc = 1)

Figure 6-31 shows the usage of the **vcmpgefp** instruction. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits long

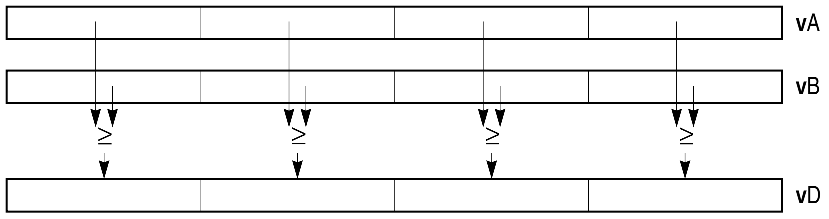


Figure 6-31. vcmpgefp—Compare Greater-Than-or-Equal of Four Floating-Point Elements (32-Bit)

vcmpgtfp*x* vcmpgtfp*x*

Vector Compare Greater-Than Floating-Point

vcmpgtfp	**v**D,**v**A,**v**B	Form: VXR
vcmpgtfp.	**v**D,**v**A,**v**B	

04	**v**D	**v**A	**v**B	Rc	710
0 5	6 10	11 15	16 20	21	22 31

```
do i=0 to 127 by 32
    if (vA)_{i:i+31} >_{fp} (vB)_{i:i+31}
        then vD_{i:i+31} ← 0xFFFF_FFFF
        else vD_{i:i+31} ← 0x0000_0000
end
if Rc=1 then do
    t ← (vD = ^{128}1)
    f ← (vD = ^{128}0)
    CR[24:27] ← t || 0b0 || f || 0b0
end
```

Each single-precision floating-point word element in **v**A is compared to the corresponding single-precision floating-point word element in **v**B. The corresponding word element in **v**D is set to all 1s if the element in **v**A is greater than the element in **v**B, and is cleared to all 0s otherwise.

If Rc = 1, CR6 is set according to all_greater_than || some_greater_than || none_greater_than.

CR6 = all_greater_than || 0b0 || none greater_than || 0b0.

Note that if a **v**A or **v**B element is a NaN, the corresponding results will be 0x0000_0000.

Other registers altered:

- Condition register (CR6):

  Affected: Bits 0-3 (if Rc = 1)

Figure 6-32 shows the usage of the **vcmpgtfp** instruction. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits long.

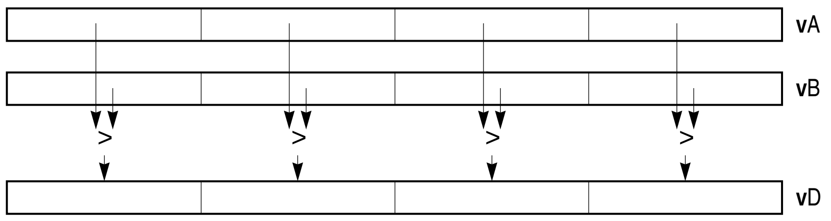


Figure 6-32. vcmpgtfp—Compare Greater-Than of Four Floating-Point Elements (32-Bit)

vcmpgtsbx vcmpgtsbx

Vector Compare Greater-Than Signed Byte

vcmpgtsb **v**D,**v**A,**v**B Form: VXR
vcmpgtsb. **v**D,**v**A,**v**B

04	**v**D	**v**A	**v**B	Rc	774
0 5	6 10	11 15	16 20	21	22 31

```
do i=0 to 127 by 8
    if $(vA)_{i:i+7} >_{si} (vB)_{i:i+7}$
        then $vD_{i:i+7} \leftarrow {}^{8}1$
        else $vD_{i:i+7} \leftarrow {}^{8}0$
end
if Rc=1 then do
    $t \leftarrow (vD = {}^{128}1)$
    $f \leftarrow (vD = {}^{128}0)$
    $CR_{24:27} \leftarrow t \,\|\, 0b0 \,\|\, f \,\|\, 0b0$
end
```

Each element of **vcmpgtsb** is a byte.

Each signed-integer element in **v**A is compared to the corresponding signed-integer element in **v**B. The corresponding element in **v**D is set to all 1s if the element in **v**A is greater than the element in **v**B, and is cleared to all 0s otherwise.

If Rc = 1, CR6 is set according to all_greater_than || some_greater_than || none_great_than.

CR6 = all_greater_than || 0b0 || none greater_than || 0b0.

Note that if a **v**A or **v**B element is a NaN, the corresponding results will be 0x0000_0000.

Other registers altered:

- Condition register (CR6):

  Affected: Bits 0-3 (if Rc = 1)

Figure 6-33 shows the usage of the **vcmpgtsb** instruction. Each of the sixteen elements in the vectors, **v**A, **v**B, and **v**D, is 8 bits long.

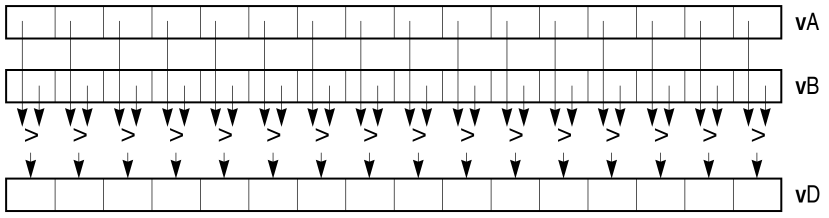


Figure 6-33. vcmpgtsb—Compare Greater-Than of Sixteen Signed Integer Elements (8-Bit)

vcmpgtsh*x* vcmpgtsh*x*

Vector Compare Greater-Than Condition Register Signed Half Word

vcmpgtsh	**v**D,**v**A,**v**B	Form: VXR
vcmpgtsh.	**v**D,**v**A,**v**B	

04	**v**D	**v**A	**v**B	Rc	838
0 5	6 10	11 15	16 20	21	22 31

```
do i=0 to 127 by 16
    if (vA)$_{i:i+15}$ >$_{si}$ (vB)$_{i:i+15}$
        then vD$_{i:i+15}$ ← $^{16}$1
        else vD$_{i:i+15}$ ← $^{16}$0
end
if Rc=1 then do
    t ← (vD = $^{128}$1)
    f ← (vD = $^{128}$0)
    CR$_{24:27}$ ← t || 0b0 || f || 0b0
end
```

Each element of **vcmpgtsh** is a half word.

Each signed-integer element in **v**A is compared to the corresponding signed-integer element in **v**B. The corresponding element in **v**D is set to all 1s if the element in **v**A is greater than the element in **v**B, and is cleared to all 0s otherwise.

If Rc = 1, CR6 is set according to all_greater_than || some_greater_than || none_great_than.

CR6 = all_greater_than || 0b0 || none greater_than || 0b0.

Note that if a **v**A or **v**B element is a NaN, the corresponding results will be 0x0000_0000.

Other registers altered:

- Condition register (CR6):

  Affected: Bits 0-3 (if Rc = 1)

Figure 6-34 shows the usage of the **vcmpgtsh** instruction. Each of the eight elements in the vectors, **v**A, **v**B, and **v**D, is 16 bits long.

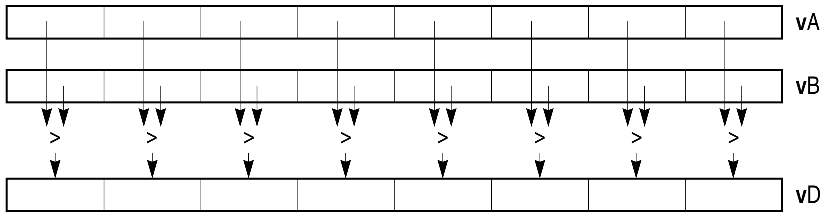


Figure 6-34. vcmpgtsh—Compare Greater-Than of Eight Signed Integer Elements (16-Bit)

vcmpgtsw*x* vcmpgtsw*x*

Vector Compare Greater-Than Signed Word

vcmpgtsw	**v**D,**v**A,**v**B	Form: VXR
vcmpgtsw.	**v**D,**v**A,**v**B	

04	**v**D	**v**A	**v**B	Rc	902
0 5	6 10	11 15	16 20	21	22 31

```
do i=0 to 127 by 32
if $(\mathbf{v}A)_{i:i+31} >_{si} (\mathbf{v}B)_{i:i+31}$
    then $\mathbf{v}D_{i:i+31} \leftarrow {}^{32}1$
    else $\mathbf{v}D_{i:i+31} \leftarrow {}^{32}0$
end

if Rc=1 then do

    t ← ($\mathbf{v}D = {}^{128}1$)
    f ← ($\mathbf{v}D = {}^{128}0$)
    $CR_{24:27}$ ← t || 0b0 || f || 0b0
end
```

Each element of **vcmpgtsw** is a word.

Each signed-integer element in **v**A is compared to the corresponding signed-integer element in **v**B. The corresponding element in **v**D is set to all 1s if the element in **v**A is greater than the element in **v**B, and is cleared to all 0s otherwise.

If Rc = 1, CR6 is set according to all_greater_than || some_greater_than || none_great_than.

CR6 = all_greater_than || 0b0 || none greater_than || 0b0.

Note that if a **v**A or **v**B element is a NaN, the corresponding results will be 0x0000_0000.

Other registers altered:

- Condition register (CR6):

  Affected: Bits 0-3 (if Rc = 1)

Figure 6-35 shows the usage of the **vcmpgtsw** instruction. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits long.

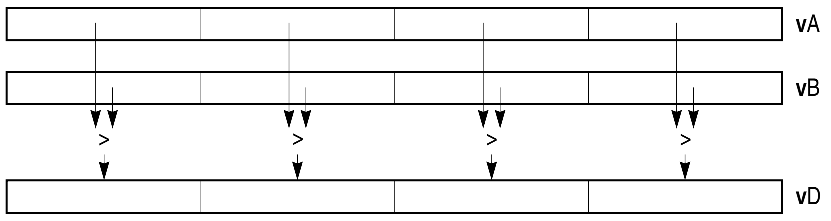


Figure 6-35. vcmpgtsw—Compare Greater-Than of Four Signed Integer Elements (32-Bit)

vcmpgtub*x* vcmpgtub*x*

Vector Compare Greater-Than Unsigned Byte

vcmpgtub	**v**D,**v**A,**v**B	Form: VXR
vcmpgtub.	**v**D,**v**A,**v**B	

04	**v**D	**v**A	**v**B	Rc	518
0 5	6 10	11 15	16 20	21	22 31

```
do i=0 to 127 by 8
    if (vA)$_{i:i+7}$ >$_{ui}$ (vB)$_{i:i+7}$
        then vD$_{i:i+7}$ ← $^{8}$1
        else vD$_{i:i+7}$ ← $^{8}$0
end
if Rc=1 then do
    t ← (vD = $^{128}$1)
    f ← (vD = $^{128}$0)
    CR[24–27] ← t || 0b0 || f || 0b0
end
```

Each element of **vcmpgtub** is a byte. Each unsigned-integer element in **v**A is compared to the corresponding unsigned-integer element in **v**B. The corresponding element in **v**D is set to all 1s if the element in **v**A is greater than the element in **v**B, and is cleared to all 0s otherwise.

If Rc = 1, CR6 is set according to all_greater_than || some_greater_than || none_great_than.

CR6 = all_greater_than || 0b0 || none greater_than || 0b0.

Note that if a **v**A or **v**B element is a NaN, the corresponding results will be 0x0000_0000.

Other registers altered:

- Condition register (CR6):

  Affected: Bits 0-3 (if Rc = 1)

Figure 6-36 shows the usage of the **vcmpgtub** instruction. Each of the sixteen elements in the vectors, **v**A, **v**B, and **v**D, is 8 bits long.

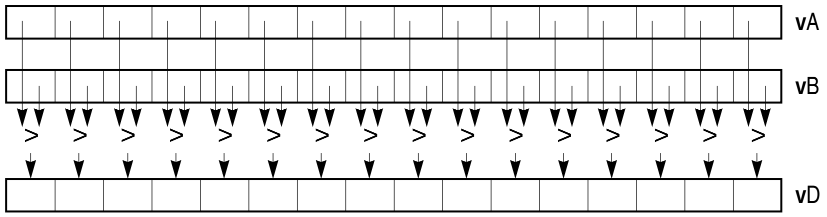


Figure 6-36. vcmpgtub—Compare Greater-Than of Sixteen Unsigned Integer Elements (8-Bit)

vcmpgtuhx vcmpgtuhx

Vector Compare Greater-Than Unsigned Half Word

vcmpgtuh vD,vA,vB Form: VXR
vcmpgtuh. vD,vA,vB

04	vD	vA	vB	Rc	582
0 5	6 10	11 15	16 20	21	22 31

```
do i=0 to 127 by 16
    if (vA)$_{i:i+151}$ $>_{ui}$ (vB)$_{i:i+15}$
        then vD$_{i:i+15}$ ← $^{16}1$
        else vD$_{i:i+15}$ ← $^{16}0$
end
if Rc=1 then do
    t ← (vD = $^{128}1$)
    f ← (vD = $^{128}0$)
    CR[24–27] ← t || 0b0 || f || 0b0
end
```

Each element of **vcmpgtuh** is a half word. Each unsigned-integer element in **v**A is compared to the corresponding unsigned-integer element in **v**B. The corresponding element in **v**D is set to all 1s if the element in **v**A is greater than the element in **v**B, and is cleared to all 0s otherwise.

If Rc = 1, CR6 is set according to all_greater_than || some_greater_than || none_great_than.

CR6 = all_greater_than || 0b0 || none greater_than || 0b0.

Note that if a **v**A or **v**B element is a NaN, the corresponding results will be 0x0000_0000.

Other registers altered:

- Condition register (CR6):
  Affected: Bits 0-3 (if Rc = 1)

Figure 6-37 shows the usage of the **vcmpgtuh** instruction. Each of the eight elements in the vectors, **v**A, **v**B, and **v**D, is 16 bits long.

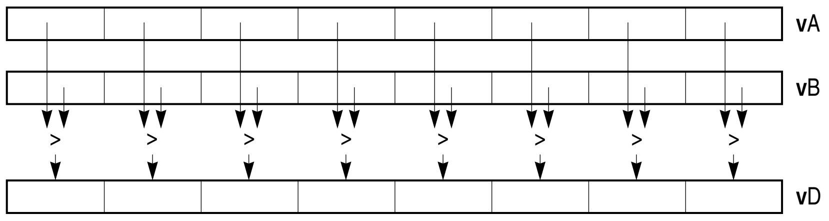


Figure 6-37. vcmpgtuh—Compare Greater-Than of Eight Unsigned Integer Elements (16-Bit)

vcmpgtuw*x* vcmpgtuw*x*

Vector Compare Greater-Than Unsigned Word

vcmpgtuw	**v**D,**v**A,**v**B	Form: VXR
vcmpgtuw.	**v**D,**v**A,**v**B	

04	**v**D	**v**A	**v**B	Rc	646
0 5	6 10	11 15	16 20	21	22 31

```
do i=0 to 127 by 32
    if $(vA)_{i:i+31} >_{ui} (vB)_{i:i+31}$
        then $vD_{i:i+31} \leftarrow {}^{32}1$
        else $vD_{i:i+31} \leftarrow {}^{32}0$
end
if Rc=1 then do
    $t \leftarrow (vD = {}^{128}1)$
    $f \leftarrow (vD = {}^{128}0)$
    $CR[24–27] \leftarrow t \| 0b0 \| f \| 0b0$
end
```

Each element of **vcmpgtuw** is a word. Each unsigned-integer element in **v**A is compared to the corresponding unsigned-integer element in **v**B. The corresponding element in **v**D is set to all 1s if the element in **v**A is greater than the element in **v**B, and is cleared to all 0s otherwise.

If Rc = 1, CR6 is set according to all_greater_than || some_greater_than || none_great_than.

CR6 = all_greater_than || 0b0 || none_greater_than || 0b0.

Note that if a **v**A or **v**B element is a NaN, the corresponding results will be 0x0000_0000.

Other registers altered:

- Condition register (CR6):

  Affected: Bits 0-3 (if Rc = 1)

Figure 6-38 shows the usage of the **vcmpgtuw** instruction. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits long.

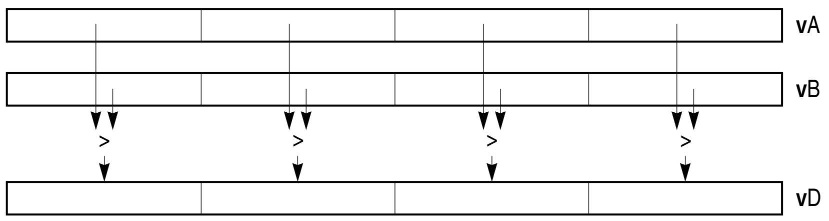


Figure 6-38. vcmpgtuw—Compare Greater-Than of Four Unsigned Integer Elements (32-Bit)

vctsxs vctsxs

Vector Convert to Signed Fixed-Point Word Saturate

vctsxs **v**D,**v**B,UIMM Form: VX

04	vD	UIMM	vB	970
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 32
    if (vB)$_{i+1:i+8}$=255 | (vB)$_{i+1:i+8}$ + UIMM ≤ 254 then
        vD$_{i:i+31}$ ← CnvtFP32ToSI32Sat((vB)$_{i:i+31}$ *$_{fp}$ 2$^{UIMM}$)
     else
        do
        if (vB)$_i$=0 then vD$_{i:i+31}$ ← 0x7FFF_FFFF
          else vD$_{i:i+31}$ ← 0x8000_0000
          VSCR$_{SAT}$ ← 1
     end
end
```

Each single-precision word element in **v**B is multiplied by 2^{UIMM}. The product is converted to a signed integer using the rounding mode, Round toward Zero. If the intermediate result is greater than (2^{31}-1) it saturates to (2^{31}-1); if it is less than -2^{31} it saturates to -2^{31}. A signed-integer result is placed into the corresponding word element of **v**D.

Fixed-point integers used by the vector convert instructions can be interpreted as consisting of 32-UIMM integer bits followed by UIMM fraction bits. The vector convert to fixed-point word instructions support only the rounding mode, Round toward Zero. A single-precision number can be converted to a fixed-point integer using any of the other three rounding modes by executing the appropriate vector round to floating-point integer instruction before the vector convert to fixed-point word instruction.

Other registers altered:

- Vector status and control register (VSCR):

  Affected: SAT

Figure 6-39 shows the usage of the **vctsxs** instruction. Each of the four elements in the vectors **v**B and **v**D is 32 bits long.

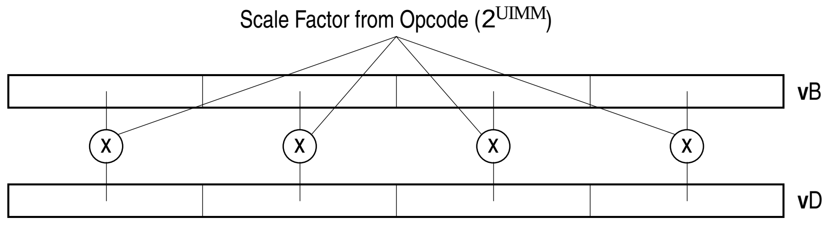


Figure 6-39. vctsxs—Convert Four Floating-Point Elements to Four Signed Integer Elements (32-Bit)

vctuxs vctuxs

Vector Convert to Unsigned Fixed-Point Word Saturate

vctuxs **v**D,**v**B,UIMM Form: VX

04	**v**D	UIMM	**v**B	906
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 32
    if $(\mathbf{v}B)_{i+1:i+8}$=255 | $(\mathbf{v}B)_{i+1:i+8}$ + UIMM ≤ 254 then
        $\mathbf{v}D_{i:i+31}$ ← CnvtFP32ToUI32Sat($(\mathbf{v}B)_{i:i+31}$ $*_{fp}$ $2^{UIM}$)
     else
        do
          if $(\mathbf{v}B)_i$=0 then$\mathbf{v}D_{i:i+31}$ ← 0xFFFF_FFFF
          else$\mathbf{v}D_{i:i+31}$ ← 0x0000_0000
          $VSCR_{SAT}$ ← 1
     end
end
```

Each single-precision floating-point word element in **v**B is multiplied by 2^{UIM}. The product is converted to an unsigned fixed-point integer using the rounding mode Round toward Zero.

If the intermediate result is greater than (2^{32}-1) it saturates to (2^{32}-1) and if it is less than 0 it saturates to 0.

The unsigned-integer result is placed into the corresponding word element of **v**D.

Other registers altered:

- Vector status and control register (VSCR):

  Affected: SAT

Figure 6-40 shows the usage of the **vctuxs** instruction. Each of the four elements in the vectors **v**B and **v**D is 32 bits long.

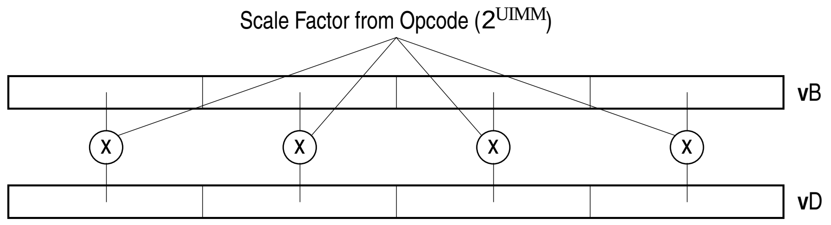


Figure 6-40. vctuxs—Convert Four Floating-Point Elements to Four Unsigned Integer Elements (32-Bit)

vexptefp vexptefp

Vector 2 Raised to the Exponent Estimate Floating Point

vexptefp **v**D,**v**B Form: VX

04	**v**D	0_0000	**v**B	394
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 32
    x ← (vB)$_{i:i+31}$
    vD$_{i:i+31}$ ← $2^x$
end
```

The single-precision floating-point estimate of 2 raised to the power of each single-precision floating-point element in **v**B is placed into the corresponding element of **v**D.

The estimate has a relative error in precision no greater than one part in 16, that is,

$$\left| \frac{\text{estimate} - 2^x}{2^x} \right| \le \frac{1}{16}$$

where *x* is the value of the element in **v**B. The most significant 12 bits of the estimate's significant are monotonic. Note that the value placed into the element of **v**D may vary between implementations, and between different executions on the same implementation.

If an operation has an integral value and the resulting value is not 0 or +∞, the result is exact.

Operation with various special values of the element in **v**B is summarized in Table 6-5 below.

Table 6-5. Special Values of the Element in vB

Value of Element in vB	Result
-∞	+0
-0	+1
+0	+1
+∞	+∞
NaN	QNaN

If VSCR[NJ] = 1, every denormalized operand element is truncated to a 0 of the same sign before the operation is carried out, and each denormalized result element truncates to a 0 of the same sign.

Other registers altered:

- None

Figure 6-41 shows the usage of the **vexptefp** instruction. Each of the four elements in the vectors **v**B and **v**D is 32 bits long.

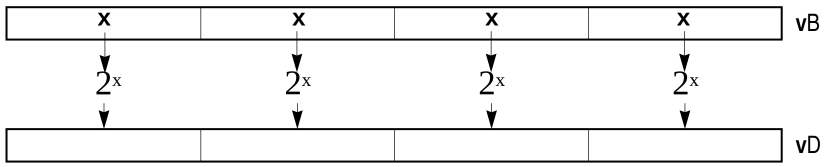


Figure 6-41. vexptefp—2 Raised to the Exponent Estimate Floating-Point for Four Floating-Point Elements (32-Bit)

vlogefp <span>vlogefp</span>

Vector Log_2 Estimate Floating Point

vlogefp **v**D,**v**B Form: VX

04	**v**D	0_0000	**v**B	458
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 32
    x ← (vB)$_{i:i+31}$
    vD$_{i:i+31}$ ← log$_2$(x)
end
```

The single-precision floating-point estimate of the base 2 logarithm of each single-precision floating-point element in **v**B is placed into the corresponding element of **v**D.

The estimate has an absolute error in precision (absolute value of the difference between the estimate and the infinitely precise value) no greater than 2^{-5}. The estimate has a relative error in precision no greater than one part in 8, as described below:

$$\left(\left|\text{estimate - }\log_2(\mathrm{x})\right| \le \frac{1}{32}\right) \qquad \text{unless} \qquad |\mathrm{x}-1| \le \frac{1}{8}$$

where x is the value of the element in **v**B, except when $|x\text{-}1| \le 1 \div 8$. The most significant 12 bits of the estimate's significant are monotonic. Note that the value placed into the element of **v**D may vary between implementations, and between different executions on the same implementation.

Operation with various special values of the element in **v**B is summarized below in Table 6-6.

Table 6-6. Special Values of the Element in vB

Value	Result
-∞	QNaN
less than 0	QNaN
±0	-∞
+∞	+∞
NaN	QNaN

If VSCR[NJ] = 1, every denormalized operand element is truncated to a 0 of the same sign before the operation is carried out, and each denormalized result element truncates to a 0 of the same sign.

Other registers altered:

- None

Figure 6-42 shows the usage of the **vexptefp** instruction. Each of the four elements in the vectors **v**B and **v**D is 32 bits long.

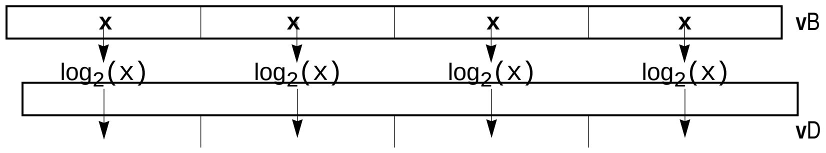


Figure 6-42. vexptefp—Log$_2$ Estimate Floating-Point for Four Floating-Point Elements (32-Bit)

vmaddfp vmaddfp

Vector Multiply Add Floating Point

vmaddfp **v**D,**v**A,**v**C,**v**B Form: VA

04	**v**D	**v**A	**v**B	**v**C	46
0 5	6 10	11 15	16 20	21 25	26 31

```
do i=0 to 127 by 32
    vD_{i:i+31} ← RndToNearFP32(((vA)_{i:i+31} *_{fp} (vC)_{i:i+31}) +_{fp} (vB)_{i:i+31})
end
```

Each single-precision floating-point word element in **v**A is multiplied by the corresponding single-precision floating-point word element in **v**C. The corresponding single-precision floating-point word element in **v**B is added to the product. The result is rounded to the nearest single-precision floating-point number and placed into the corresponding word element of **v**D.

Note that a vector multiply floating-point instruction is not provided. The effect of such an instruction can be obtained by using **vmaddfp** with **v**B containing the value -0.0 (0x8000_0000) in each of its four single-precision floating-point word elements. (The value must be -0.0, not +0.0, in order to obtain the IEEE-conforming result of -0.0 when the result of the multiplication is -0.)

Other registers altered:

- None

If VSCR[NJ] = 1, every denormalized operand element is truncated to a 0 of the same sign before the operation is carried out, and each denormalized result element truncates to a 0 of the same sign. Figure 6-43 shows the usage of the **vmaddfp** instruction. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits long.

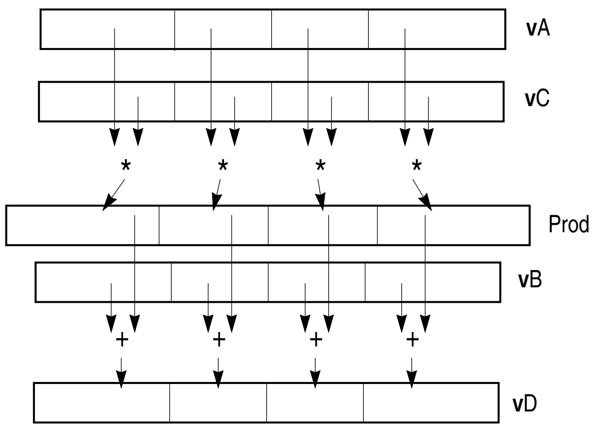


Figure 6-43. vmaddfp—Multiply-Add Four Floating-Point Elements (32-Bit)

vmaxfp vmaxfp

Vector Maximum Floating Point

vmaxfp **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	1034
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 32
    if $(vA)_{i:i+31} \geq_{fp} (vB)_{i:i+31}$
        then $vD_{i:i+31} \leftarrow (vA)_{i:i+31}$
        else $vD_{i:i+31} \leftarrow (vB)_{i:i+31}$
end
```

Each single-precision floating-point word element in **v**A is compared to the corresponding single-precision floating-point word element in **v**B. The larger of the two single-precision floating-point values is placed into the corresponding word element of **v**D.

The maximum of +0 and -0 is +0. The maximum of any value and a NaN is a QNaN.

Other registers altered:

- None

Figure 6-44 shows the usage of the **vmaxfp** instruction. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits long.

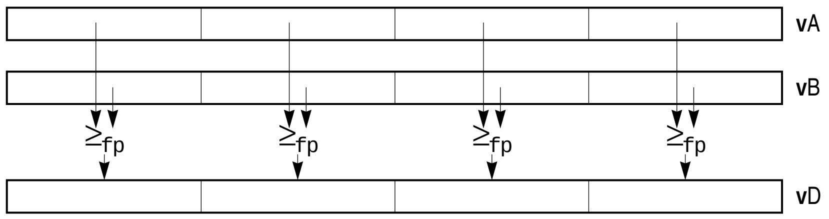


Figure 6-44. vmaxfp—Maximum of Four Floating-Point Elements (32-Bit)

vmaxsb vmaxsb

Vector Maximum Signed Byte

vmaxsb **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	258
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 8

    if $(vA)_{i:i+7} \ge_{si} (vB)_{i:i+7}$
        then $vD_{i:i+7} \leftarrow (vA)_{i:i+7}$
        else $vD_{i:i+7} \leftarrow (vB)_{i:i+7}$

end
```

Each element of **vmaxsb** is a byte.

Each signed-integer element in **v**A is compared to the corresponding signed-integer element in **v**B. The larger of the two signed-integer values is placed into the corresponding element of **v**D.

Other registers altered:

- None

Figure 6-45 shows the usage of the **vmaxsb** instruction. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits long.

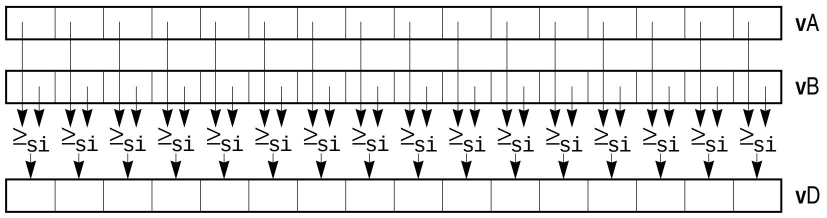


Figure 6-45. vmaxsb—Maximum of Sixteen Signed Integer Elements (8-Bit)

vmaxsh vmaxsh

Vector Maximum Signed Half Word

vmaxsh **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	322
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 16
    if $(\mathbf{v}A)_{i:i+7} \geq_{si} (\mathbf{v}B)_{i:i+15}$
        then $\mathbf{v}D_{i:i+15} \leftarrow (\mathbf{v}A)_{i:i+15}$
        else $\mathbf{v}D_{i:i+15} \leftarrow (\mathbf{v}B)_{i:i+15}$
end
```

Each element of **vmaxsh** is a half word.

Each signed-integer element in **v**A is compared to the corresponding signed-integer element in **v**B. The larger of the two signed-integer values is placed into the corresponding element of **v**D.

Other registers altered:

- None

Figure 6-46 shows the usage of the **vmaxsh** instruction. Each of the eight elements in the vectors, **v**A, **v**B, and **v**D, is 16 bits longlong.

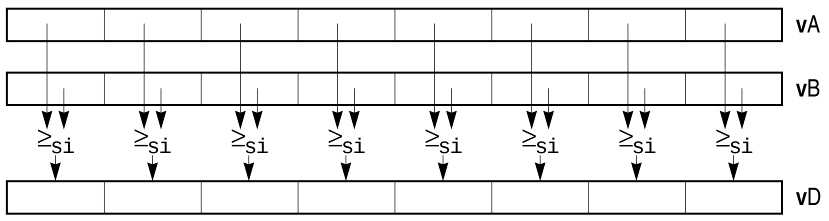


Figure 6-46. vmaxsh—Maximum of Eight Signed Integer Elements (16-Bit)

vmaxsw vmaxsw

Vector Maximum Signed Word

vmaxsw **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	386
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 32
    if $(\mathbf{v}A)_{i:i+31} \geq_{si} (\mathbf{v}B)_{i:i+31}$
        then $\mathbf{v}D_{i:i+31} \leftarrow (\mathbf{v}A)_{i:i+31}$
        else $\mathbf{v}D_{i:i+31} \leftarrow (\mathbf{v}B)_{i:i+31}$
end
```

Each element of **vmaxsw** is a word.

Each signed-integer element in **v**A is compared to the corresponding signed-integer element in **v**B. The larger of the two signed-integer values is placed into the corresponding element of **v**D.

Other registers altered:

- None

Figure 6-47 shows the usage of the **vmaxsw** instruction. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits long.

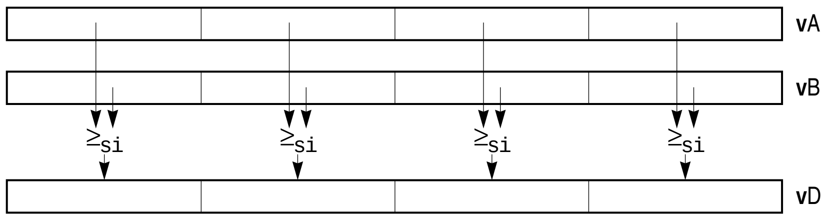


Figure 6-47. vmaxsw—Maximum of Four Signed Integer Elements (32-Bit)

vmaxub vmaxub

Vector Maximum Signed Byte

vmaxub **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	2
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 8
    if $(vA)_{i:i+7} \geq_{ui} (vB)_{i:i+7}$
        then $vD_{i:i+7} \leftarrow (vA)_{i:i+7}$
        else $vD_{i:i+7} \leftarrow (vB)_{i:i+7}$
end
```

Each element of **vmaxub** is a byte.

Each unsigned-integer element in **v**A is compared to the corresponding unsigned-integer element in **v**B. The larger of the two unsigned-integer values is placed into the corresponding element of **v**D.

Other registers altered:

- None

Figure 6-48 shows the usage of the **vmaxub** instruction. Each of the sixteen elements in the vectors, **v**A, **v**B, and **v**D, is 8 bits long.

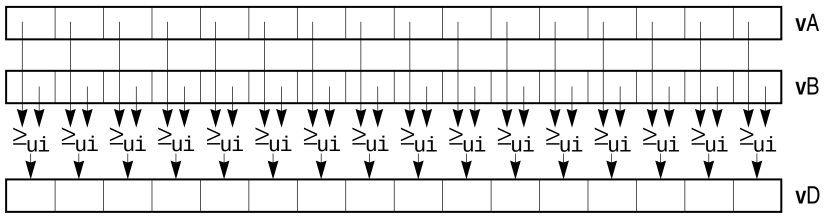


Figure 6-48. vmaxub—Maximum of Sixteen Unsigned Integer Elements (8-Bit)

vmaxuh vmaxuh

Vector Maximum Unsigned Half Word

vmaxuh **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	66
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 16
    if $(\mathbf{v}A)_{i:i+15} \geq_{ui} (\mathbf{v}B)_{i:i+15}$
        then $\mathbf{v}D_{i:i+15} \leftarrow (\mathbf{v}A)_{i:i+15}$
        else $\mathbf{v}D_{i:i+15} \leftarrow (\mathbf{v}B)_{i:i+15}$
end
```

Each element of **vmaxuh** is a half word.

Each unsigned-integer element in **v**A is compared to the corresponding unsigned-integer element in **v**B. The larger of the two unsigned-integer values is placed into the corresponding element of **v**D.

Other registers altered:

- None

Figure 6-49 shows the usage of the **vmaxuh** instruction. Each of the eight elements in the vectors, **v**A, **v**B, and **v**D, is 16 bits long.

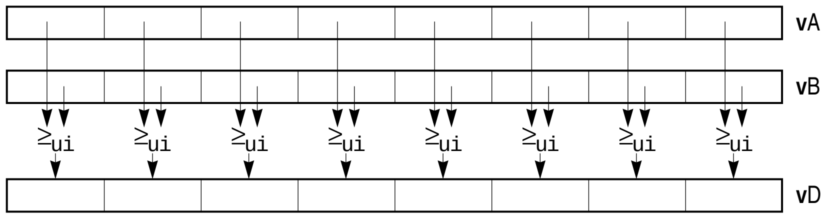


Figure 6-49. vmaxuh—Maximum of Eight Unsigned Integer Elements (16-Bit)

vmaxuw vmaxuw

Vector Maximum Unsigned Word

vmaxuw **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	130
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 32
    if $(vA)_{i:i+31} \geq_{ui} (vB)_{i:i+31}$
        then $vD_{i:i+31} \leftarrow (vA)_{i:i+31}$
        else $vD_{i:i+31} \leftarrow (vB)_{i:i+31}$
end
```

Each element of **vmaxuw** is a word.

Each unsigned-integer element in **v**A is compared to the corresponding unsigned-integer element in **v**B. The larger of the two unsigned-integer values is placed into the corresponding element of **v**D.

Other registers altered:

- None

Figure 6-50 shows the usage of the **vmaxuw** instruction. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits long.

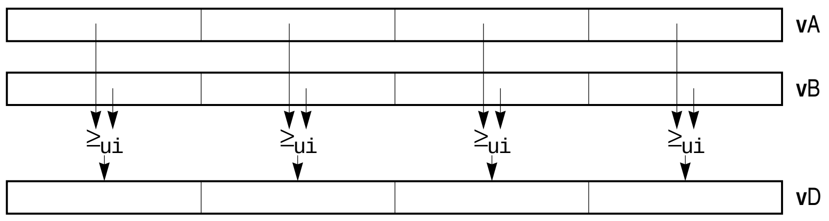


Figure 6-50. vmaxuw—Maximum of Four Unsigned Integer Elements (32-Bit)

vmhaddshs vmhaddshs

Vector Multiply High and Add Signed Half Word Saturate

vmhaddshs **v**D,**v**A,**v**B,**v**C Form: VA

04	**v**D	**v**A	**v**B	**v**C	32
0 5	6 10	11 15	16 20	21 25	26 31

```
do i=0 to 127 by 16
    prod$_{0:31}$← (vA)$_{i:i+15}$ *$_{si}$ (vB)$_{i:i+15}$
    temp$_{0:16}$← prod$_{0:16}$ +$_{int}$ SignExtend((vC)$_{i:i+15}$,17)
    vD$_{i:i+15}$← SItoSIsat(temp$_{0:16}$,16)
end
```

Each signed-integer half word element in **v**A is multiplied by the corresponding signed-integer half word element in **v**B, producing a 32-bit signed-integer product. Bits 0-16 of the intermediate product are added to the corresponding signed-integer half-word element in **v**C after they have been sign extended to 17-bits. The 16-bit saturated result from each of the eight 17-bit sums is placed in register **v**D.

If the intermediate result is greater than (2^{15}-1) it saturates to (2^{15}-1) and if it is less than (-2^{15}) it saturates to (-2^{15}).

The signed-integer result is placed into the corresponding half-word element of **v**D.

Other registers altered:

- Vector status and control register (VSCR):

  Affected: SAT

Figure 6-51 shows the usage of the **vmhaddshs** instruction. Each of the eight elements in the vectors, **v**A, **v**B, **v**C, and **v**D, is 16 bits long.

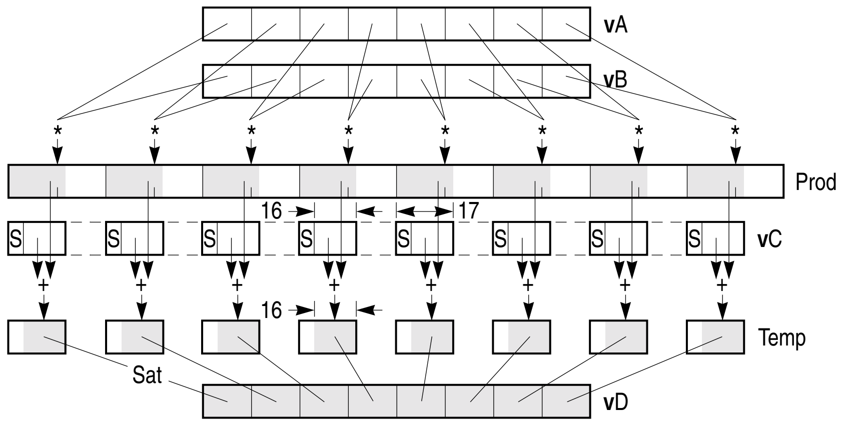


Figure 6-51. vmhaddshs—Multiply-High and Add Eight Signed Integer Elements (16-Bit)

vmhraddshs vmhraddshs

Vector Multiply High Round and Add Signed Half Word Saturate

vmhraddshs **v**D,**v**A,**v**B,**v**C Form: VA

04	**v**D	**v**A	**v**B	**v**C	33
0 5	6 10	11 15	16 20	21 25	26 31

```
do i=0 to 127 by 16
    $prod_{0:31} \leftarrow (vA)_{i:i+15} *_{si} (vB)_{i:i+15}$
    $prod_{0:31} \leftarrow prod_{0:31} +_{int}$ 0x0000_4000
    $temp_{0:16} \leftarrow prod_{0:16} +_{int}$ SignExtend($(vC)_{i:i+15}$,17)
    $(vD)_{i:i+15} \leftarrow$ SItoSIsat($temp_{0:16}$,16)
end
```

Each signed integer halfword element in register **v**A is multiplied by the corresponding signed integer halfword element in register **v**B, producing a 32-bit signed integer product. The value 0x0000_4000 is added to the product, producing a 32-bit signed integer sum. Bits 0—16 of the sum are added to the corresponding signed integer halfword element in register **v**D.

If the intermediate result is greater than (2^{15}-1) it saturates to (2^{15}-1) and if it is less than (-2^{15}) it saturates to (-2^{15}).

The signed integer result is and placed into the corresponding halfword element of register **v**D.

Figure 6-52 shows the usage of the **vmhraddshs** instruction. Each of the eight elements in the vectors, **v**A, **v**B, **v**C, and **v**D, is 16 bits long.

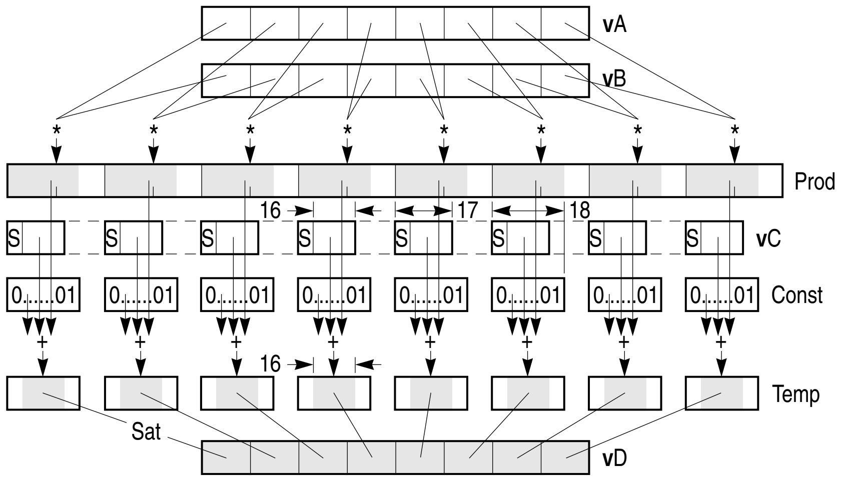


Figure 6-52. vmhraddshs—Multiply-High Round and Add Eight Signed Integer Elements (16-Bit)

vminfp vminfp

Vector Minimum Floating Point

vminfp **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	1098
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 32
    if $(\mathbf{v}A)_{i:i+31} <_{fp} (\mathbf{v}B)_{i:i+31}$
        then $\mathbf{v}D_{i:i+31} \leftarrow (\mathbf{v}A)_{i:i+31}$
        else $\mathbf{v}D_{i:i+31} \leftarrow (\mathbf{v}B)_{i:i+31}$
end
```

Each single-precision floating-point word element in register **v**A is compared to the corresponding single-precision floating-point word element in register **v**B. The smaller of the two single-precision floating-point values is placed into the corresponding word element of register **v**D.

The minimum of + 0.0 and - 0.0 is - 0.0. The minimum of any value and a NaN is a QNaN.

If VSCR[NJ] = 1, every denormalized operand element is truncated to 0 before the comparison is made.

Figure 6-53 shows the usage of the **vminfp** instruction. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits long.

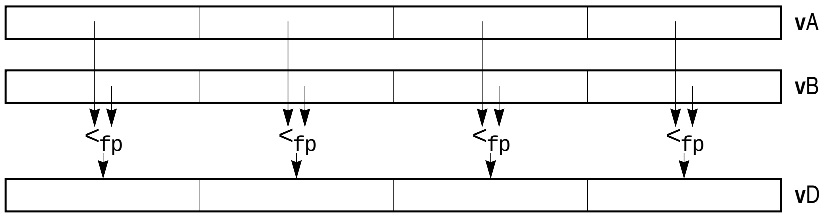


Figure 6-53. vminfp—Minimum of Four Floating-Point Elements (32-Bit)

vminsb vminsb

Vector Minimum Signed Byte

vminsb **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	770
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 8
    if $(\mathbf{v}A)_{i:i+7} <_{si} (\mathbf{v}B)_{i:i+7}$
        then $\mathbf{v}D_{i:i+7} \leftarrow (\mathbf{v}A)_{i:i+7}$
        else $\mathbf{v}D_{i:i+7} \leftarrow (\mathbf{v}B)_{i:i+7}$
end
```

Each element of **vminsb** is a byte.

Each signed-integer element in **v**A is compared to the corresponding signed-integer element in **v**B. The larger of the two signed-integer values is placed into the corresponding element of **v**D.

Other registers altered:

- None

Figure 6-54 shows the usage of the **vminsb** instruction. Each of the sixteen elements in the vectors, **v**A, **v**B, and **v**D, is 8 bits long.

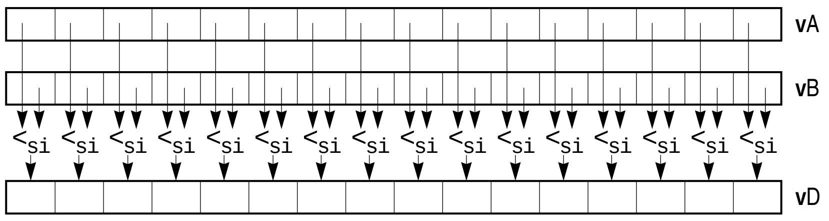


Figure 6-54. vminsb—Minimum of Sixteen Signed Integer Elements (8-Bit)

vminsh vminsh

Vector Minimum Signed Half Word

vminsh **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	834
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 16
    if (vA)_{i:i+15} <_si (vB)_{i:i+15}
        then vD_{i:i+15} ← (vA)_{i:i+15}
        else vD_{i:i+15} ← (vB)_{i:i+15}
end
```

Each element of **vminsh** is a half word.

Each signed-integer element in **v**A is compared to the corresponding signed-integer element in **v**B. The larger of the two signed-integer values is placed into the corresponding element of **v**D.

Other registers altered:

- None

Figure 6-55 shows the usage of the **vminsh** instruction. Each of the eight elements in the vectors, **v**A, **v**B, and **v**D, is 16 bits long.

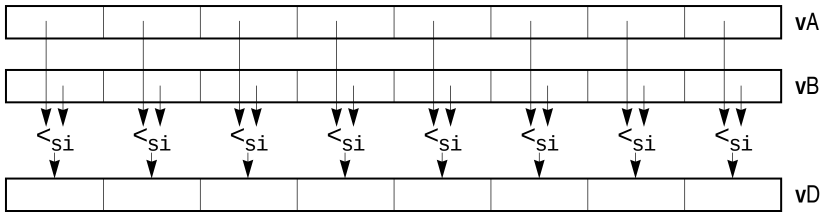


Figure 6-55. vminsh—Minimum of Eight Signed Integer Elements (16-Bit)

vminsw vminsw

Vector Minimum Signed Word

vminsw **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	898
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 32
    if $(\mathbf{v}A)_{i:i+31} <_{si} (\mathbf{v}B)_{i:i+31}$
        then $\mathbf{v}D_{i:i+31} \leftarrow (\mathbf{v}A)_{i:i+31}$
        else $\mathbf{v}D_{i:i+31} \leftarrow (\mathbf{v}B)_{\mathbf{i:i+31}}$
end
```

Each element of **vminsw** is a word.

Each signed-integer element in **v**A is compared to the corresponding signed-integer element in **v**B. The larger of the two signed-integer values is placed into the corresponding element of **v**D.

Other registers altered:

- None

Figure 6-56 shows the usage of the **vminsw** instruction. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits long.

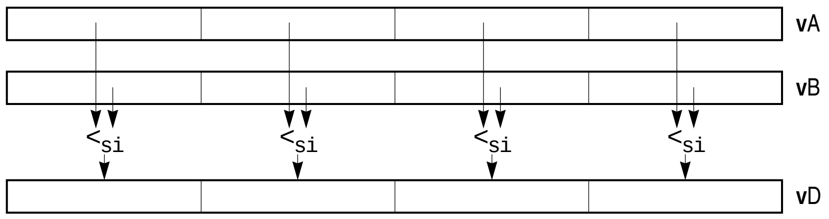


Figure 6-56. vminsw—Minimum of Four Signed Integer Elements (32-Bit)

vminub vminub

Vector Minimum Unsigned Byte

vminub **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	514
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 8
    if $(\mathbf{v}A)_{i:i+7} <_{ui} (\mathbf{v}B)_{i:i+7}$
        then $\mathbf{v}D_{i:i+7} \leftarrow (\mathbf{v}A)_{i:i+7}$
        else $\mathbf{v}D_{i:i+7} \leftarrow (\mathbf{v}B)_{\mathbf{i:i+7}}$
end
```

Each element of **vminub** is a byte.

Each unsigned-integer element in **v**A is compared to the corresponding unsigned-integer element in **v**B. The larger of the two unsigned-integer values is placed into the corresponding element of **v**D.

Other registers altered:

- None

Figure 6-57 shows the usage of the **vminub** instruction. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits long.

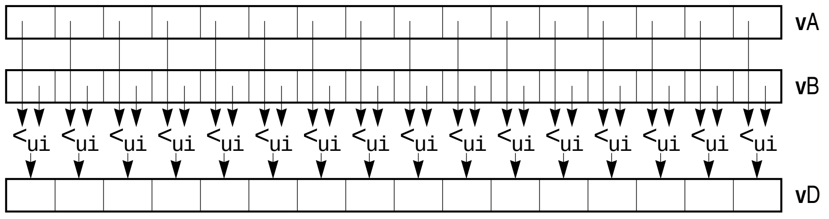


Figure 6-57. vminub—Minimum of Sixteen Unsigned Integer Elements (8-Bit)

vminuh vminuh

Vector Minimum Unsigned Half Word

vminuh **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	578
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 16
    if $(\mathbf{v}A)_{i:i+15} <_{ui} (\mathbf{v}B)_{i:i+15}$
        then $\mathbf{v}D_{i:i+15} \leftarrow (\mathbf{v}A)_{i:i+15}$
        else $\mathbf{v}D_{i:i+15} \leftarrow (\mathbf{v}B)_{\mathbf{i:i+15}}$
end
```

Each element of **vminuh** is a half word.

Each unsigned-integer element in **v**A is compared to the corresponding unsigned-integer element in **v**B. The larger of the two unsigned-integer values is placed into the corresponding element of **v**D.

Other registers altered:

- None

Figure 6-58 shows the usage of the **vminuh** instruction. Each of the eight elements in the vectors, **v**A, **v**B, and **v**D, is 16 bits long.

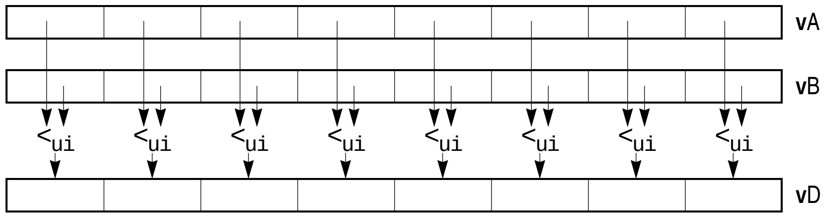


Figure 6-58. vminuh—Minimum of Eight Unsigned Integer Elements (16-Bit)

vminuw vminuw

Vector Minimum Unsigned Word

vminuw **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	642
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 32
    if $(\mathbf{v}A)_{i:i+31} <_{ui} (\mathbf{v}B)_{i:i+31}$
        then $\mathbf{v}D_{i:i+31} \leftarrow (\mathbf{v}A)_{i:i+31}$
        else $\mathbf{v}D_{i:i+31} \leftarrow (\mathbf{v}B)_{\mathbf{i:i+31}}$
end
```

Each element of **vminuw** is a word.

Each unsigned-integer element in **v**A is compared to the corresponding unsigned-integer element in **v**B. The larger of the two unsigned-integer values is placed into the corresponding element of **v**D.

Other registers altered:

- None

Figure 6-59 shows the usage of the **vminuw** instruction. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits long.

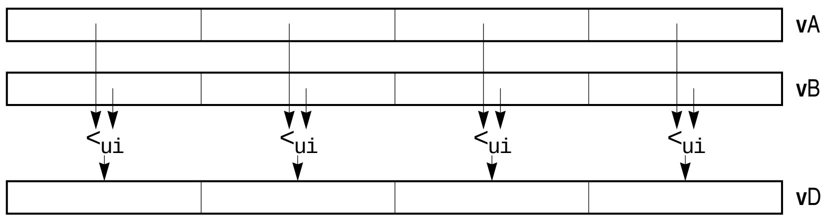


Figure 6-59. vminuw—Minimum of Four Unsigned Integer Elements (32-Bit)

vmladduhm vmladduhm

Vector Multiply Low and Add Unsigned Half Word Modulo

vmladduhm **v**D,**v**A,**v**B,**v**C Form: VA

04	**v**D	**v**A	**v**B	**v**C	34
0 5	6 10	11 15	16 20	21 25	26 31

```
do i=0 to 127 by 16
    $prod_{0:31} \leftarrow (\mathbf{v}A)_{i:i+15} *_{ui} (\mathbf{v}B)_{i:i+15}$
    $\mathbf{v}D_{i:i+15} \leftarrow prod_{0:31} +_{int} (\mathbf{v}C)_{i:i+15}$
end
```

Each integer half-word element in **v**A is multiplied by the corresponding integer half-word element in **v**B, producing a 32-bit integer product. The product is added to the corresponding integer half-word element in **v**C. The integer result is placed into the corresponding half-word element of **v**D.

Note that **vmladduhm** can be used for unsigned or signed integers.

Other registers altered:

- None

Figure 6-60 shows the usage of the **vmladduhm** instruction. Each of the eight elements in the vectors, **v**A, **v**B, **v**C, and **v**D, is 16 bits long.

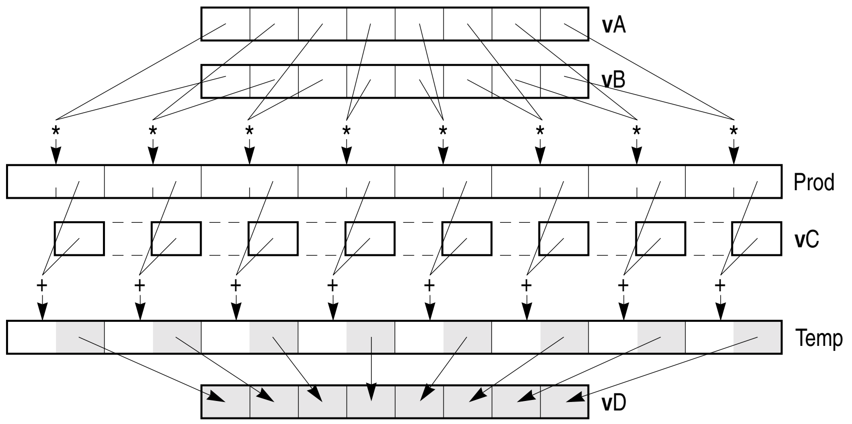


Figure 6-60. vmladduhm—Multiply-Add of Eight Integer Elements (16-Bit)

vmrghb vmrghb

Vector Merge High Byte

vmrghb **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	12
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 63 by 8
    vD_{i*2:(i*2)+15} ← (vA)_{i:i+7} || (vB)_{i:i+7}
end
```

Each element of **vmrghb** is a byte.

The elements in the high-order half of **v**A are placed, in the same order, into the even-numbered elements of **v**D. The elements in the high-order half of **v**B are placed, in the same order, into the odd-numbered elements of **v**D.

Other registers altered:

- None

Figure 6-61 shows the usage of the **vmrghb** instruction. Each of the sixteen elements in the vectors, **v**A, **v**B, and **v**D, is 8 bits long.

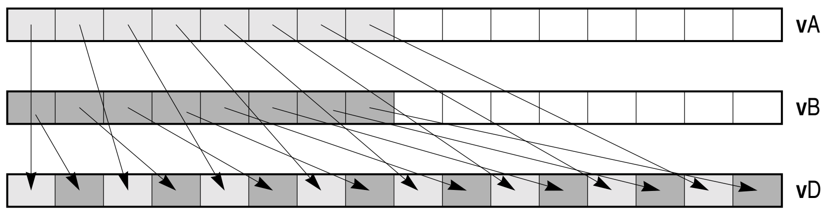


Figure 6-61. vmrghb—Merge Eight High-Order Elements (8-Bit)

vmrghh vmrghh

Vector Merge High Half word

vmrghh **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	76
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 63 by 16
    vD_{i*2:(i*2)+31} ← (vA)_{i:i+15} ‖ (vB)_{i:i+15}
end
```

Each element of **vmrghh** is a half word.

The elements in the high-order half of **v**A are placed, in the same order, into the even-numbered elements of **v**D. The elements in the high-order half of **v**B are placed, in the same order, into the odd-numbered elements of **v**D.

Other registers altered:

- None

Figure 6-62 shows the usage of the **vmrghh** instruction. Each of the eight elements in the vectors, **v**A, **v**B, and **v**D, is 16 bits long.

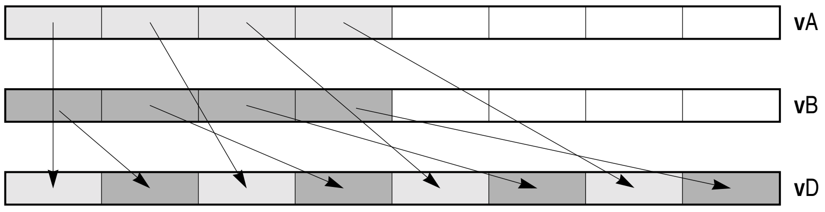


Figure 6-62. vmrghh—Merge Four High-Order Elements (16-Bit)

vmrghw vmrghw

Vector Merge High Word

vmrghw **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	140
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 63 by 32
    $\mathbf{v}D_{i*2:(i*2)+63} \leftarrow (\mathbf{v}A)_{i:i+31} \parallel (\mathbf{v}B)_{i:i+31}$
end
```

Each element of **vmrghw** is a word.

The elements in the high-order half of **v**A are placed, in the same order, into the even-numbered elements of **v**D. The elements in the high-order half of **v**B are placed, in the same order, into the odd-numbered elements of **v**D.

Other registers altered:

- None

Figure 6-63 shows the usage of the **vmrghw** instruction. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits long.

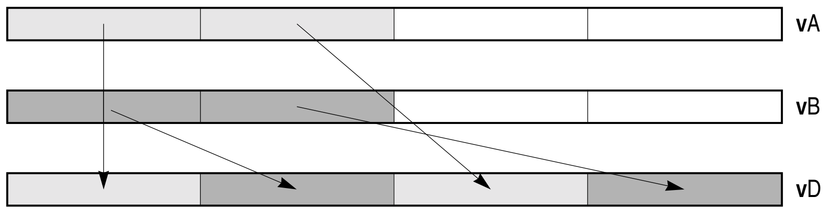


Figure 6-63. vmrghw—Merge Four High-Order Elements (32-Bit)

vmrglb vmrglb

Vector Merge Low Byte

vmrglb **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	268
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 63 by 8
    $vD_{i*2:(i*2)+15} \leftarrow (vA)_{i+64:i+71} \| (vB)_{i+64:i+71}$
end
```

Each element offer **vmrglb** is a byte.

The elements in the low-order half of **v**A are placed, in the same order, into the even-numbered elements of **v**D. The elements in the low-order half of **v**B are placed, in the same order, into the odd-numbered elements of **v**D.

Other registers altered:

- None

Figure 6-64 shows the usage of the **vmrglb** instruction. Each of the sixteen elements in the vectors, **v**A, **v**B, and **v**D, is 8 bits long.

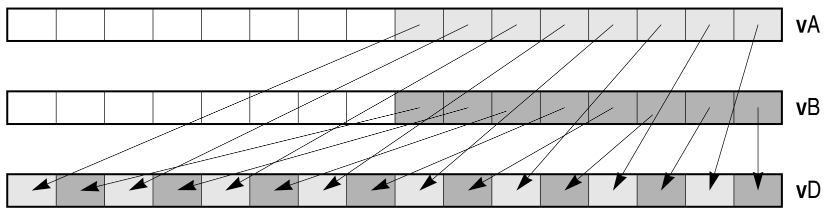


Figure 6-64. vmrglb—Merge Eight Low-Order Elements (8-Bit)

vmrglh vmrglh

Vector Merge Low Half Word

vmrglh **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	332
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 63 by 16
    $\mathbf{v}D_{i*2:(i*2)+31} \leftarrow (\mathbf{v}A)_{i+64:i+79} \parallel (\mathbf{v}B)_{i+64:i+79}$
end
```

Each element of **vmrglh** is a half word.

The elements in the low-order half of **v**A are placed, in the same order, into the even-numbered elements of **v**D. The elements in the low-order half of **v**B are placed, in the same order, into the odd-numbered elements of **v**D.

Other registers altered:

- None

Figure 6-65 shows the usage of the **vmrglh** instruction. Each of the eight elements in the vectors, **v**A, **v**B, and **v**D, is 16 bits long.

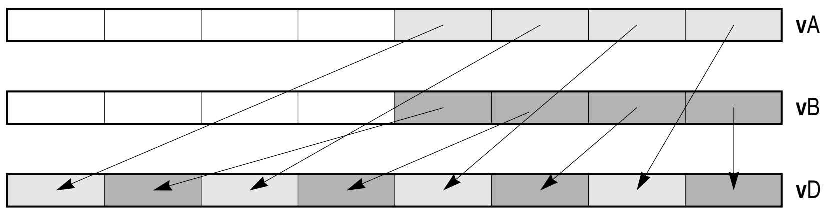


Figure 6-65. vmrglh—Merge Four Low-Order Elements (16-Bit)

vmrglw vmrglw

Vector Merge Low Word

vmrglw **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	396
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 63 by 32
    $vD_{i*2:(i*2)+63} \leftarrow (vA)_{i+64:i+95} \| (vB)_{i+64:i+95}$
end
```

Each element of **vmrglw** is a word.

The elements in the low-order half of **v**A are placed, in the same order, into the even-numbered elements of **v**D. The elements in the low-order half of **v**B are placed, in the same order, into the odd-numbered elements of **v**D.

Other registers altered:

- None

Figure 6-66 shows the usage of the **vmrglw** instruction. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits long.

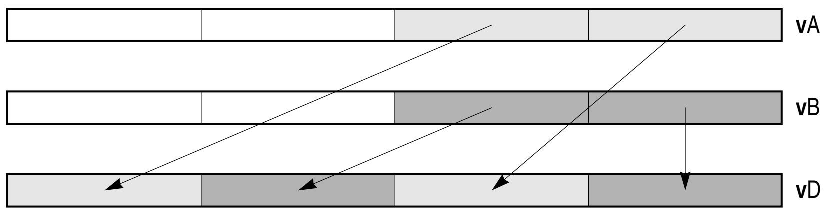


Figure 6-66. vmrglw—Merge Four Low-Order Elements (32-Bit)

vmsummbm vmsummbm

Vector Multiply Sum Mixed-Sign Byte Modulo

vmsummbm **v**D,**v**A,**v**B,**v**C Form: VA

04	**v**D	**v**A	**v**B	**v**C	37
0 5	6 10	11 15	16 20	21 25	26 31

```
do i=0 to 127 by 32
    $temp_{0:31} \leftarrow (vC)_{i:i+31}$
     do j=0 to 31 by 8
        $prod_{0:15} \leftarrow (vA)_{i+j:i+j+7} *_{sui} (vB)_{i+j:i+j+7}$
        $temp_{0:31} \leftarrow temp_{0:31} +_{int} SignExtend(prod_{0:15},32)$
        end
    $vD_{i:i+31} \leftarrow temp_{0:31}$
end
```

For each word element in **v**C the following operations are performed in the order shown.

- Each of the four signed-integer byte elements contained in the corresponding word element of **v**A is multiplied by the corresponding unsigned-integer byte element in **v**B, producing a signed-integer 16-bit product.
- The signed-integer modulo sum of these four products is added to the signed-integer word element in **v**C.
- The signed-integer result is placed into the corresponding word element of **v**D.

Other registers altered:

- None

Figure 6-67 shows the usage of the **vmsummbm** instruction. Each of the sixteen elements in the vectors, **v**A, and **v**B, are 8 bits long. Each of the four elements in the vectors, **v**C and **v**D are 32 bits long.

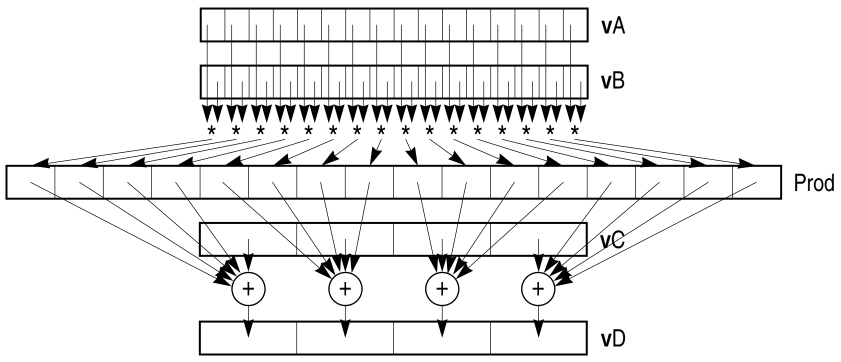


Figure 6-67. vmsummbm—Multiply-Sum of Integer Elements (8-Bit to 32-Bit)

vmsumshm vmsumshm

Vector Multiply Sum Signed Half Word Modulo

vmsumshm **v**D,**v**A,**v**B,**v**C Form: VA

04	**v**D	**v**A	**v**B	**v**C	40
0 5	6 10	11 15	16 20	21 25	26 31

```
do i=0 to 127 by 32
    $temp_{0:31} \leftarrow (vC)_{i:i+31}$
     do j=0 to 31 by 16
        $prod_{0:31} \leftarrow (vA)_{i+j:i+j+15} *_{si} (vB)_{i+j:i+j+15}$
        $temp_{0:31} \leftarrow temp_{0:31} +_{int} prod_{0:31}$
        $vD_{i:i+31} \leftarrow temp_{0:31}$
    end
end
```

For each word element in **v**C the following operations are performed in the order shown.

- Each of the two signed-integer half-word elements contained in the corresponding word element of **v**A is multiplied by the corresponding signed-integer half-word element in **v**B, producing a signed-integer 32-bit product.
- The signed-integer modulo sum of these two products is added to the signed-integer word element in **v**C.
- The signed-integer result is placed into the corresponding word element of **v**D.

Other registers altered:

- None

Figure 6-68 shows the usage of the **vmsumshm** instruction. Each of the eight elements in the vectors, **v**A, and **v**B, are 16 bits long. Each of the four elements in the vectors, **v**C and **v**D are 32 bits long.

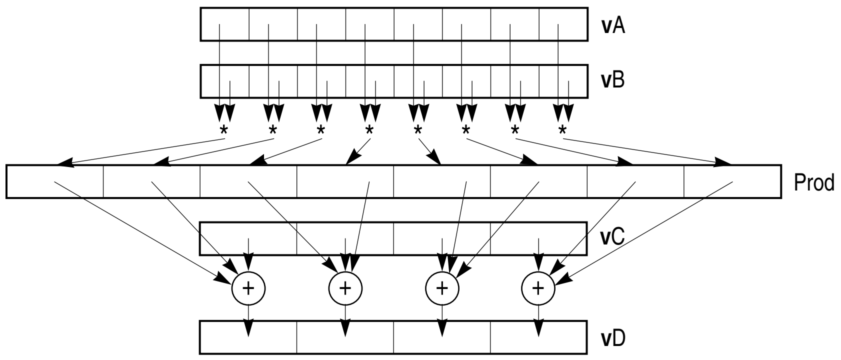


Figure 6-68. vmsumshm—Multiply-Sum of Signed Integer Elements (16-Bit to 32-Bit)

vmsumshs vmsumshs

Vector Multiply Sum Signed Half Word Saturate

vmsumshs **v**D,**v**A,**v**B,**v**C Form: VA

04	**v**D	**v**A	**v**B	**v**C	41
0 5	6 10	11 15	16 20	21 25	26 31

```
do i=0 to 127 by 32
    $temp_{0:33} \leftarrow$ SignExtend((vC)$_{i:i+31}$,34)
     do j=0 to 31 by 16
        $prod_{0:31} \leftarrow$ (vA)$_{i+j:i+j+15}$ $*_{si}$ (vB)$_{i+j:i+j+15}$
        $temp_{0:33} \leftarrow temp_{0:33} +_{int}$ SignExtend($prod_{0:31}$,34)
        vD$_{i:i+31} \leftarrow$ SItoSIsat($temp_{0:33}$,32)
    end
end
```

For each word element in **v**C the following operations are performed in the order shown.

- Each of the two signed-integer half-word elements in the corresponding word element of **v**A is multiplied by the corresponding signed-integer half-word element in **v**B, producing a signed-integer 32-bit product.
- The signed-integer sum of these two products is added to the signed-integer word element in **v**C.
- If this intermediate result is greater than (2^{31}-1) it saturates to (2^{31}-1) and if it is less than -2^{31} it saturates to -2^{31}.
- The signed-integer result is placed into the corresponding word element of **v**D.

Other registers altered:

- SAT

Figure 6-69 shows the usage of the **vmsumshs** instruction. Each of the eight elements in the vectors, **v**A, and **v**B, are 16 bits long. Each of the four elements in the vectors, **v**C and **v**D are 32 bits long.

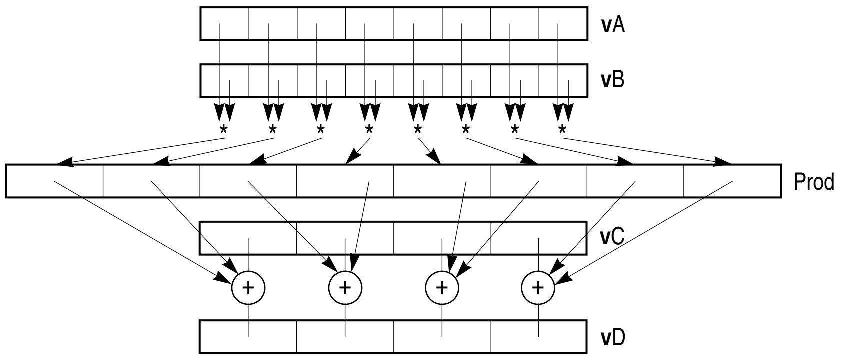


Figure 6-69. vmsumshs—Multiply-Sum of Signed Integer Elements (16-Bit to 32-Bit)

vmsumubm vmsumubm

Vector Multiply Sum Unsigned Byte Modulo

vmsumubm **v**D,**v**A,**v**B,**v**C Form: VA

04	**v**D	**v**A	**v**B	**v**C	36
0 5	6 10	11 15	16 20	21 25	26 31

```
do i=0 to 127 by 32
    $temp_{0:31} \leftarrow (vC)_{i:i+31}$
     do j=0 to 31 by 8
        $prod_{0:15} \leftarrow (vA)_{i+j:i+j+7} *_{ui} (vB)_{i+j:i+j+7}$
        $temp_{0:32} \leftarrow temp_{0:32} +_{int} ZeroExtend(prod_{0:15},32)$
        $vD_{i:i+31} \leftarrow temp_{0:31}$
    end
end
```

For each word element in **v**C the following operations are performed in the order shown.

- Each of the four unsigned-integer byte elements contained in the corresponding word element of **v**A is multiplied by the corresponding unsigned-integer byte element in **v**B, producing an unsigned-integer 16-bit product.
- The unsigned-integer modulo sum of these four products is added to the unsigned-integer word element in **v**C.
- The unsigned-integer result is placed into the corresponding word element of **v**D.

Other registers altered:

- None

Figure 6-70 shows the usage of the **vmsumubm** instruction. Each of the sixteen elements in the vectors, **v**A, and **v**B, are 8 bits long. Each of the four elements in the vectors, **v**C and **v**D are 32 bits long.

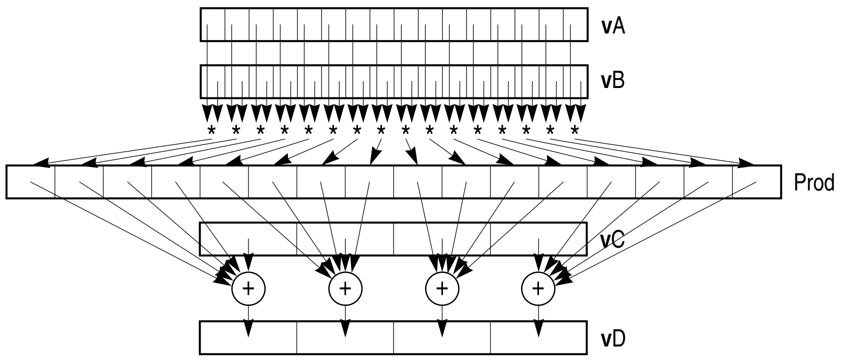


Figure 6-70. vmsumubm—Multiply-Sum of Unsigned Integer Elements (8-Bit to 32-Bit)

vmsumuhm vmsumuhm

Vector Multiply Sum Unsigned Half Word Modulo

vmsumuhm **v**D,**v**A,**v**B,**v**C Form: VA

04	**v**D	**v**A	**v**B	**v**C	38
0 5	6 10	11 15	16 20	21 25	26 31

```
do i=0 to 127 by 32
    $temp_{0:31} \leftarrow (vC)_{i:i+31}$
     do j=0 to 31 by 16
        $prod_{0:31} \leftarrow (vA)_{i+j:i+j+15}$ $*_{ui}$ $(vB)_{i+j:i+j+15}$
        $temp_{0:31} \leftarrow temp_{0:31}$ $+_{int}$ $prod_{0:31}$
        $vD_{i:i+31} \leftarrow temp_{2:33}$
    end
end
```

For each word element in **v**C the following operations are performed in the order shown.

- Each of the two unsigned-integer half-word elements contained in the corresponding word element of **v**A is multiplied by the corresponding unsigned-integer half-word element in **v**B, producing a unsigned-integer 32-bit product.
- The unsigned-integer sum of these two products is added to the unsigned-integer word element in **v**C.
- The unsigned-integer result is placed into the corresponding word element of **v**D.

Other registers altered:

- None

Figure 6-71 shows the usage of the **vmsumuhm** instruction. Each of the eight elements in the vectors, **v**A, and **v**B, are 16 bits long. Each of the four elements in the vectors, **v**C and **v**D are 32 bits long.

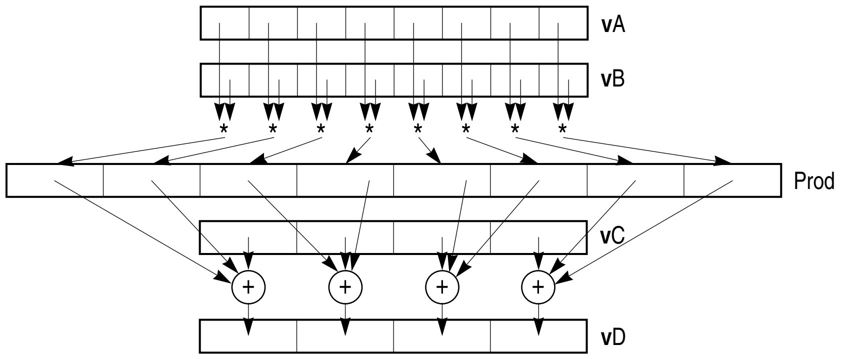


Figure 6-71. vmsumuhm—Multiply-Sum of Unsigned Integer Elements (16-Bit to 32-Bit)

vmsumuhs vmsumuhs

Vector Multiply Sum Unsigned Half Word Saturate

vmsumuhs **v**D,**v**A,**v**B,**v**C Form: VA

04	**v**D	**v**A	**v**B	**v**C	39
0 5	6 10	11 15	16 20	21 25	26 31

```
do i=0 to 127 by 32
    temp$_{0:33}$ ← ZeroExtend((vC)$_{i:i+31}$,34)
     do j=0 to 31 by 16
        prod$_{0:31}$ ← (vA)$_{i+j:i+j+15}$ *$_{ui}$ (vB)$_{i+j:i+j+15}$
        temp$_{0:33}$ ← temp$_{0:33}$ +$_{int}$ ZeroExtend(prod$_{0:31}$,34)
        vD$_{i:i+31}$ ← UItoUIsat(temp$_{0:33}$,32)
    end
end
```

For each word element in **v**C the following operations are performed in the order shown.

- Each of the two unsigned-integer half-word elements contained in the corresponding word element of **v**A is multiplied by the corresponding unsigned-integer half-word element in **v**B, producing an unsigned-integer 32-bit product.
- The unsigned-integer sum of these two products is saturate-added to the unsigned-integer word element in **v**C.
- The unsigned-integer result is placed into the corresponding word element of **v**D.

Other registers altered:

- SAT

Figure 6-72 shows the usage of the **vmsumuhs** instruction. Each of the eight elements in the vectors, **v**A, and **v**B, are 16 bits long. Each of the four elements in the vectors, **v**C and **v**D are 32 bits long.

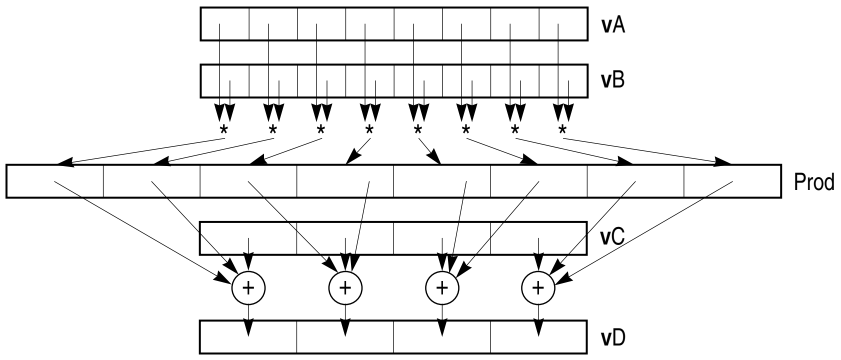


Figure 6-72. vmsumuhs—Multiply-Sum of Unsigned Integer Elements (16-Bit to 32-Bit)

vmulesb vmulesb

Vector Multiply Even Signed Byte

vmulesb **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	776
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 16
    $prod_{0:15} \leftarrow (\mathbf{v}A)_{i:i+7} *_{si} (\mathbf{v}B)_{i:i+7}$
    $\mathbf{v}D_{i:i+15} \leftarrow prod_{0:15}$
end
```

Each even-numbered signed-integer byte element in **v**A is multiplied by the corresponding signed-integer byte element in **v**B. The eight 16-bit signed-integer products are placed, in the same order, into the eight half-words of **v**D.

Other registers altered:

- None

Figure 6-73 shows the usage of the **vmulesb** instruction. Each of the sixteen elements in the vectors, **v**A, and **v**B, is 8 bits long. Each of the eight elements in the vector **v**D, is 16 bits long.

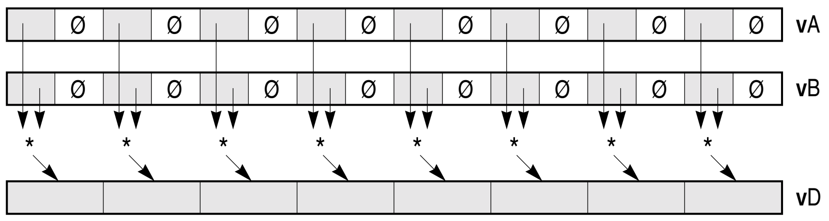


Figure 6-73. vmulesb—Even Multiply of Eight Signed Integer Elements (8-Bit)

vmulesh vmulesh

Vector Multiply Even Signed Half Word

vmulesh **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	840
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 32
    prod$_{0:31}$← (vA)$_{i:i+15}$ *$_{si}$ (vB)$_{i:i+15}$
    vD$_{i:i+31}$← prod$_{0:31}$
end
```

Each even-numbered signed-integer half-word element in **v**A is multiplied by the corresponding signed-integer half-word element in **v**B. The four 32-bit signed-integer products are placed, in the same order, into the four words of **v**D.

Other registers altered:

- None

Figure 6-74 shows the usage of the **vmulesh** instruction. Each of the eight elements in the vectors, **v**A, and **v**B, is 16 bits long. Each of the four elements in the vector **v**D, is 32 bits long.

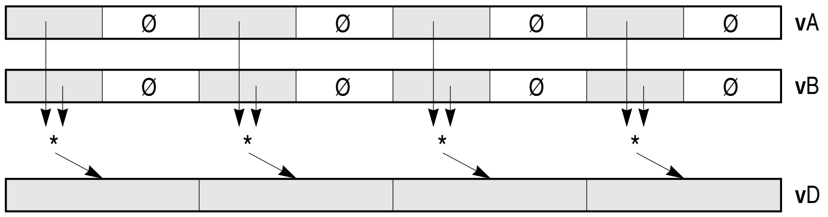


Figure 6-74. vmulesb—Even Multiply of Four Signed Integer Elements (16-Bit)

vmuleub vmuleub

Vector Multiply Even Unsigned Byte

vmuleub **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	520
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 16
    prod$_{0:15}$ ← (vA)$_{i:i+7}$ *$_{ui}$ (vB)$_{i:i+7}$
    (vD)$_{i:i+15}$ ← prod$_{0:15}$
end
```

Each even-numbered unsigned-integer byte element in register **v**A is multiplied by the corresponding unsigned-integer byte element in register **v**B. The eight 16-bit unsigned-integer products are placed, in the same order, into the eight halfwords of register **v**D.

Other registers altered:

- None

Figure 6-75 shows the usage of the **vmuleub** instruction. Each of the sixteen elements in the vectors, **v**A, and **v**B, is 8 bits long. Each of the eight elements in the vector **v**D, is 16 bits long.

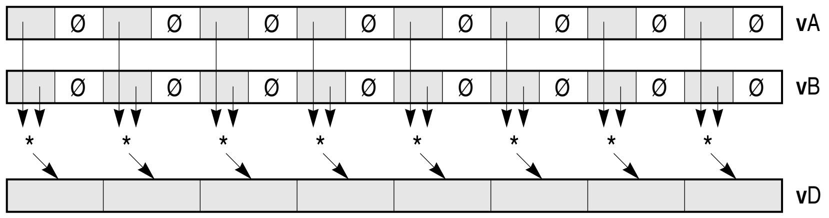


Figure 6-75. vmuleub—Even Multiply of Eight Unsigned Integer Elements (8-Bit)

vmuleuh vmuleuh

Vector Multiply Even Unsigned Half Word

vmuleuh **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	584
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 32

    $prod_{0:31} \leftarrow (\mathbf{v}A)_{i:i+15} *_{ui} (\mathbf{v}B)_{i:i+15}$
    $(\mathbf{v}D)_{i:i+31} \leftarrow prod_{0:31}$

end
```

Each even-numbered unsigned-integer halfword element in register **v**A is multiplied by the corresponding unsigned-integer halfword element in register **v**B. The four 32-bit unsigned-integer products are placed, in the same order, into the four words of register **v**D.

Other registers altered:

- None

Figure 6-76 shows the usage of the **vmuleuh** instruction. Each of the eight elements in the vectors, **v**A, and **v**B, is 16 bits long. Each of the four elements in the vector **v**D, is 32 bits long.

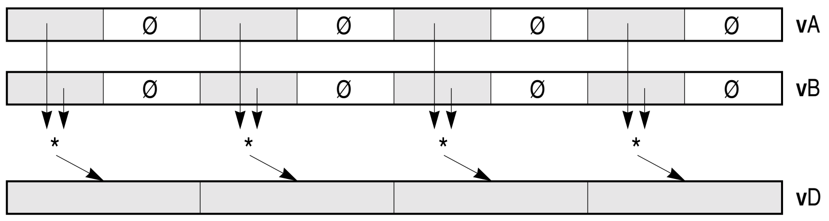


Figure 6-76. vmuleuh—Even Multiply of Four Unsigned Integer Elements (16-Bit)

vmulosb vmulosb

Vector Multiply Odd Signed Byte

vmulosb **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	264
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 16
    prod$_{0:15}$← (vA)$_{i+8:i+15}$ *$_{si}$ (vB)$_{i+8:i+15}$
    vD$_{i:i+15}$← prod$_{0:15}$
end
```

Each odd-numbered signed-integer byte element in **v**A is multiplied by the corresponding signed-integer byte element in **v**B. The eight 16-bit signed-integer products are placed, in the same order, into the eight half-words of **v**D.

Other registers altered:

- None

Figure 6-77 shows the usage of the **vmulosb** instruction. Each of the sixteen elements in the vectors, **v**A, and **v**B, is 8 bits long. Each of the eight elements in the vector **v**D, is 16 bits long.

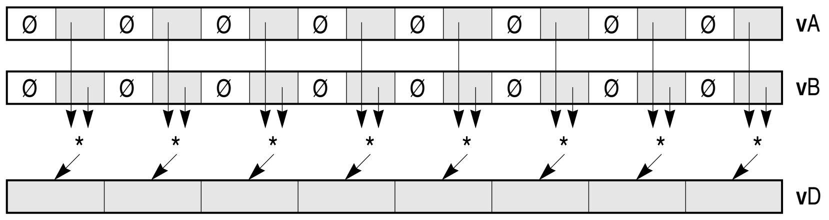


Figure 6-77. vmulosb—Odd Multiply of Eight Signed Integer Elements (8-Bit)

vmulosh vmulosh

Vector Multiply Odd Signed Half Word

vmulosh **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	328
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 32
    $prod_{0:31} \leftarrow (\mathbf{v}A)_{i+16:i+31} \,*_{si}\, (\mathbf{v}B)_{i+16:i+31}$
    $\mathbf{v}D_{i:i+31} \leftarrow prod_{0:31}$
end
```

Each odd-numbered signed-integer half-word element in **v**A is multiplied by the corresponding signed-integer half-word element in **v**B. The four 32-bit signed-integer products are placed, in the same order, into the four words of **v**D.

Other registers altered:

- None

Figure 6-78 shows the usage of the **vmuleuh** instruction. Each of the eight elements in the vectors, **v**A, and **v**B, is 16 bits long. Each of the four elements in the vector **v**D, is 32 bits long.

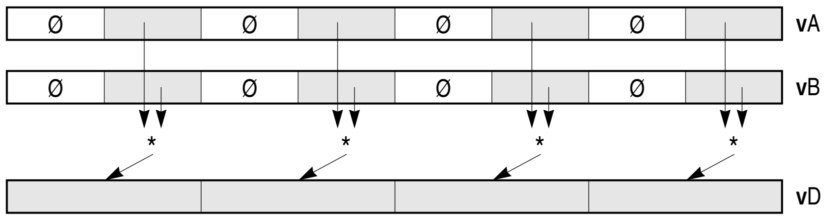


Figure 6-78. vmuleuh—Odd Multiply of Four Unsigned Integer Elements (16-Bit)

vmuloub vmuloub

Vector Multiply Odd Unsigned Byte

vmuloub **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	8
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 8
    $prod_{0:15} \leftarrow (vA)_{i+8:i+15}\ {}^{*}_{ui}\ (vB)_{i+n:i+15}$
    $vD_{i:i+15} \leftarrow prod_{0:15}$
end
```

Each odd-numbered unsigned-integer byte element in **v**A is multiplied by the corresponding unsigned-integer byte element in **v**B. The eight 16-bit unsigned-integer products are placed, in the same order, into the eight half-word s of **v**D.

Other registers altered:

- None

Figure 6-79 shows the usage of the **vmuloub** instruction. Each of the sixteen elements in the vectors, **v**A, and **v**B, is 8 bits long. Each of the eight elements in the vector **v**D, is 16 bits long.

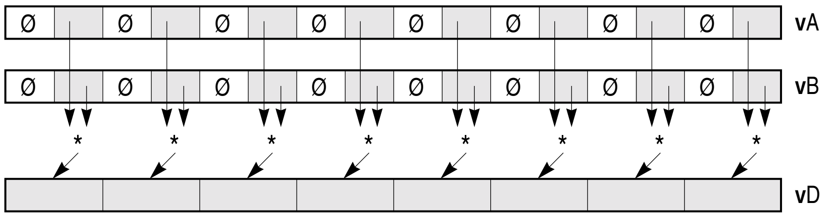


Figure 6-79. vmuloub—Odd Multiply of Eight Unsigned Integer Elements (8-Bit)

vmulouh vmulouh

Vector Multiply Odd Unsigned Half Word

vmulouh **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	72
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 16
    prod_{0:31} ← (vA)_{i+16:i+31} *_{ui} (vB)_{i+n:i+311}
    vD_{i:i+31} ← prod_{0:31}
end
```

Each odd-numbered unsigned-integer half-word element in **v**A is multiplied by the corresponding unsigned-integer half-word element in **v**B. The four 32-bit unsigned-integer products are placed, in the same order, into the four words of **v**D.

Other registers altered:

- None

Figure 6-80 shows the usage of the **vmulouh** instruction. Each of the eight elements in the vectors, **v**A, and **v**B, is 16 bits long. Each of the four elements in the vector **v**D, is 32 bits long.

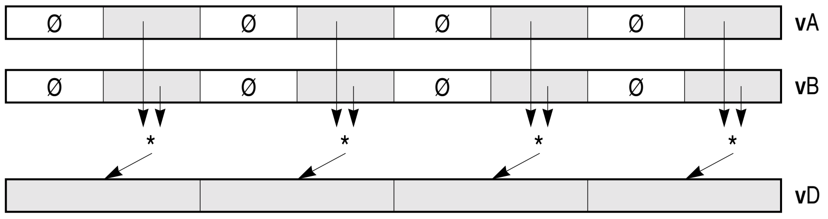


Figure 6-80. vmulouh—Odd Multiply of Four Unsigned Integer Elements (16-Bit)

vnmsubfp vnmsubfp

Vector Negative Multiply-Subtract Floating Point

vnmsubfp **v**D,**v**A,**v**C,**v**B Form: VA

04	**v**D	**v**A	**v**B	**v**C	47
0 5	6 10	11 15	16 20	21 25	26 31

```
do i=0 to 127 by 32
    vD_{i:i+31} ← -RndToNearFP32(((vA)_{i:i+31} *_{fp} (vC)_{i:i+31}) -_{fp} (vB)_{i:i+31})
end
```

Each single-precision floating-point word element in **v**A is multiplied by the corresponding single-precision floating-point word element in **v**C. The corresponding single-precision floating-point word element in **v**B is subtracted from the product. The sign of the difference is inverted. The result is rounded to the nearest single-precision floating-point number and placed into the corresponding word element of **v**D.

Note that only one rounding occurs in this operation. Also note that a QNaN result is not negated.

Other registers altered:

- None

Figure 6-81 shows the usage of the **vnmsubfp** instruction. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits long.

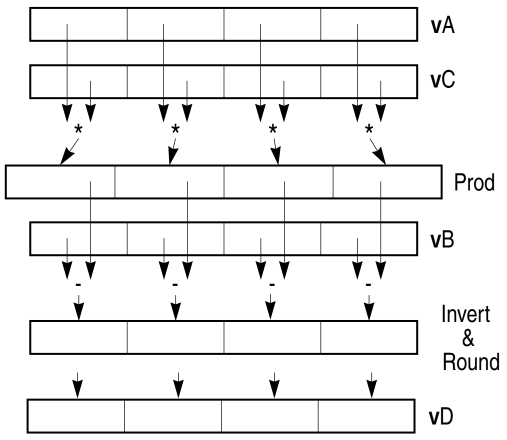


Figure 6-81. vnmsubfp—Negative Multiply-Subtract of Four Floating-Point Elements (32-Bit)

vnor vnor

Vector Logical NOR

vnor **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	1284
0 5	6 10	11 15	16 20	21 31

```
    vD ← ¬((vA) | (vB))
```

The contents of **v**A are bitwise ORed with the contents of **v**B and the complemented result is placed into **v**D.

Other registers altered:

- None

Simplified mnemonics:

vnot **v**D, **v**S equivalent to **vnor** **v**D, **v**S, **v**S

Figure 6-82 shows the usage of the **vnor** instruction.

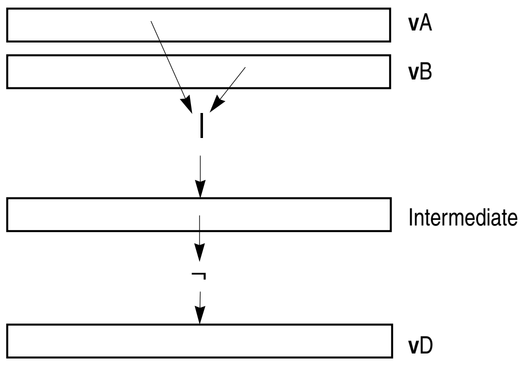


Figure 6-82. vnor—Bitwise NOR of 128-bit Vector

vor vor

Vector Logical OR

vor **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	1156
0 5	6 10	11 15	16 20	21 31

```
vD ← (vA) | (vB)
```

The contents of **v**A are ORed with the contents of **v**B and the result is placed into **v**D.

Other registers altered:

- None

Simplified mnemonics:

vmr vD, **v**S equivalent to **vor v**D, **v**S, **v**S

Figure 6-83 shows the usage of the **vor** instruction.

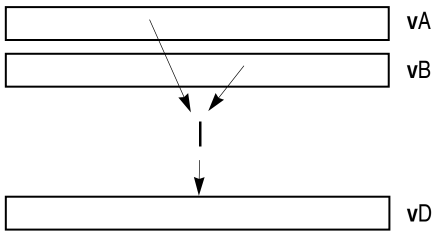


Figure 6-83. vor—Bitwise OR of 128-bit Vector

vperm vperm

Vector Permute

vperm **v**D,**v**A,**v**B,**v**C Form: VA

04	**v**D	**v**A	**v**B	**v**C	43
0 5	6 10	11 15	16 20	21 25	26 31

```
$temp_{0:255}$ ← (vA) ‖ (vB)
do i=0 to 127 by 8
    b ← $(vC)_{i+3:i+7}$ ‖ 0b000
    $vD_{i:i+7}$ ← $temp_{b:b+7}$
end
```

Let the source vector be the concatenation of the contents of **v**A followed by the contents of **v**B. For each integer i in the range 0–15, the contents of the byte element in the source vector specified in bits 3–7 of byte element i in **v**C are placed into byte element i of **v**D.

Other registers altered:

- None

Programming note: See the programming notes with the Load Vector for Shift Left and Load Vector for Shift Right instructions for examples of usage on the **vperm** instruction.

Figure 6-84 shows the usage of the **vperm** instruction. Each of the sixteen elements in the vectors, **v**A, **v**B, **v**C, and **v**D, is 8 bits long.

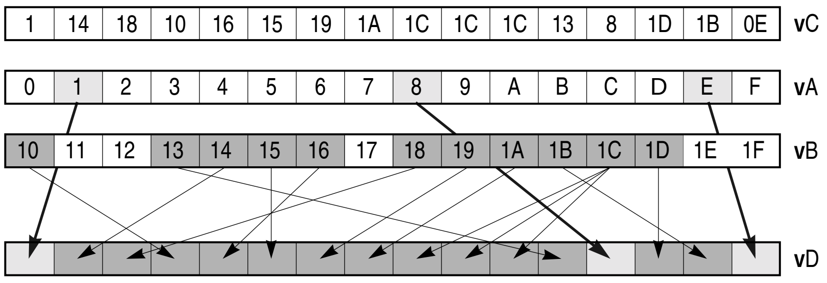

1 14 18 10 16 15 19 1A 1C 1C 1C 13 8 1D 1B 0E
vC
0 1 2 3 4 5 6 7 8 9 A B C D E F
vA
10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F
vB
vD

Figure 6-84. vperm—Concatenate Sixteen Integer Elements (8-Bit)

vpkpx vpkpx

Vector Pack Pixel32

vpkpx **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	782
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 63 by 16
    $vD_{i} \leftarrow (vA)_{i*2+7}$
    $vD_{i+1:i+5} \leftarrow (vA)_{(i*2)+8:(i*2)+12}$
    $vD_{i+6:i+10} \leftarrow (vA)_{(i*2)+16:(i*2)+20}$
    $vD_{i+11:i+15} \leftarrow (vA)(_{(i*2)+24:(i*2)+28}$
    $vD_{i+64} \leftarrow (vB)_{(i*2)+7}$
    $vD_{i+65:i+69} \leftarrow (vB)_{(i*2)+8:(i*2)+12}$
    $vD_{i+70:i+74} \leftarrow (vB)_{(i*2)+16:(i*2)+20}$
    $vD_{i+75:i+79} \leftarrow (vB)_{(i*2)+24:(i*2)+28}$
end
```

The source vector is the concatenation of the contents of **v**A followed by the contents of **v**B. Each 32-bit word element in the source vector is packed to produce a 16-bit half-word value as described below and placed into the corresponding half-word element of **v**D. A word is packed to 16 bits by concatenating, in order, the following bits.

- bit 7 of the first byte (bit 7 of the word)
- bits 0–4 of the second byte (bits 8–12 of the word)
- bits 0–4 of the third byte (bits 16–20 of the word)
- bits 0–4 of the fourth byte (bits 24–28 of the word)

Figure 6-85 shows which bits of the source word are packed to form the half word in the destination register.

Source Word

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
							▨	▨	▨	▨	▨	▨				▨	▨	▨	▨	▨				▨	▨	▨	▨	▨			

vD Packed Half Word

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
7	8	9	10	11	12	12	17	18	19	20	24	25	26	27	28

Figure 6-85. How a Word is Packed to a Half Word

Other registers altered:

- None

Programming note: Each source word can be considered to be a 32-bit pixel consisting of four 8-bit channels. Each target half-word can be considered to be a 16-bit pixel consisting of one 1-bit channel and three 5-bit channels. A channel can be used to specify the intensity of a particular color, such as red, green, or blue, or to provide other information needed by the application.

Figure 6-86 shows the usage of the **vpkpx** instruction. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits long.

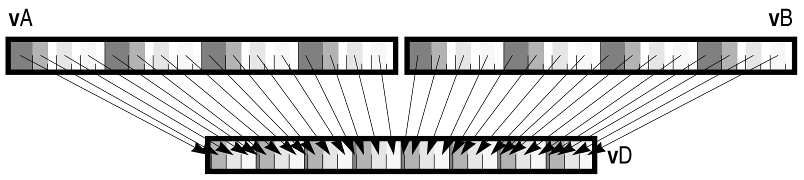


Figure 6-86. vpkpx—Pack Eight Elements (32-Bit) to Eight Elements (16-Bit)

vpkshss vpkshss

Vector Pack Signed Half Word Signed Saturate

vpkshss **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	398
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 63 by 8
    vD_{i:i+7} ← SItoSIsat((vA)_{i*2:(i*2)+15},8)
    vD_{i+64:i+71} ← SItoSIsat((vB)_{i*2:(i*2)+15},8)
end
```

Let the source vector be the concatenation of the contents of **v**A followed by the contents of **v**B.

Each signed integer half-word element in the source vector is converted to an 8-bit signed integer. If the value of the element is greater than $(2^7 - 1)$ the result saturates to $(2^7 - 1)$ and if the value is less than -2^7 the result saturates to -2^7. The result is placed into the corresponding byte element of **v**D.

Other registers altered:

- SAT

Figure 6-87 shows the usage of the **vpkshss** instruction. Each of the eight elements in the vectors, **v**A, and **v**B, is 16 bits long. Each of the sixteen elements in the vector **v**D, is 8 bits long.

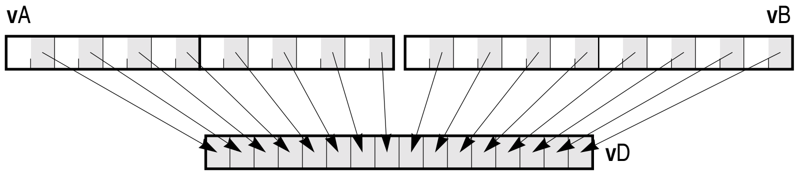


Figure 6-87. vpkshss—Pack Sixteen Signed Integer Elements (16-Bit) to Sixteen Signed Integer Elements (8-Bit)

vpkshus vpkshus

Vector Pack Signed Half Word Unsigned Saturate

vpkshus **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	270
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 63 by 8
    $\mathbf{v}D_{i:i+7}$ ← SItoUIsat(($\mathbf{v}A$)$_{i*2:(i*2)+7}$,8)
    $\mathbf{v}D_{i+64:i+71}$ ← SItoUIsat(($\mathbf{v}B$)$_{i*2:(i*2)+7}$,8)
end
```

Let the source vector be the concatenation of the contents of **v**A followed by the contents of **v**B.

Each signed integer half-word element in the source vector is converted to an 8-bit unsigned integer. If the value of the element is greater than (2^8 - 1) the result saturates to (2^8 - 1) and if the value is less than 0 the result saturates to 0. The result is placed into the corresponding byte element of **v**D.

Other registers altered:

- SAT

Figure 6-88 shows the usage of the **vpkshus** instruction. Each of the eight elements in the vectors, **v**A, and **v**B, is 16 bits long. Each of the sixteen elements in the vector **v**D, is 8 bits long.

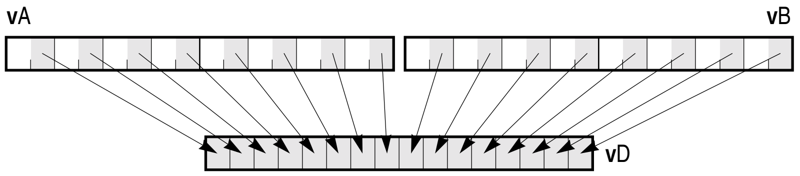


Figure 6-88. vpkshus—Pack Sixteen Signed Integer Elements (16-Bit) to Sixteen Unsigned Integer Elements (8-Bit)

vpkswss vpkswss

Vector Pack Signed Word Signed Saturate

vpkswss **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	462
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 63 by 16
    vD_{i:i+15} ← SItoSIsat((vA)_{i*2:(i*2)+31},16)
    vD_{i+64:i+79} ← SItoSIsat((vB)_{i*2:(i*2)+31},16)
end
```

Let the source vector be the concatenation of the contents of **v**A followed by the contents of **v**B.

Each signed integer word element in the source vector is converted to a 16-bit signed integer half word. If the value of the element is greater than (2^{15} - 1) the result saturates to (2^{15} - 1) and if the value is less than -2^{15} the result saturates to -2^{15}. The result is placed into the corresponding half-word element of **v**D.

Other registers altered:

- SAT

Figure 6-89 shows the usage of the **vpkswss** instruction. Each of the four elements in the vectors, **v**A, and **v**B, is 32 bits long. Each of the eight elements in the vector **v**D, is 16 bits long.

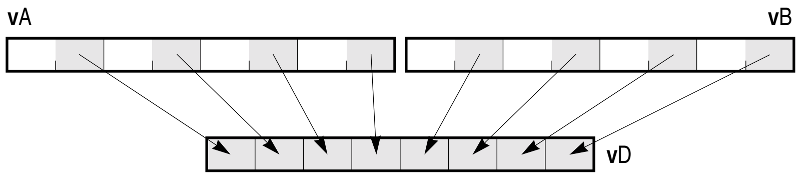


Figure 6-89. vpkswss—Pack Eight Signed Integer Elements (32-Bit) to Eight Signed Integer Elements (16-Bit)

vpkswus vpkswus

Vector Pack Signed Word Unsigned Saturate

vpkswus **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	334
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 63 by 16
    $vD_{i:i+15}$ ← SItoUIsat(($vA)_{i*2:(i*2)+31}$,16)
    $vD_{i+64:i+79}$ ← SItoUIsat(($vB)_{i*2:(i*2)+31}$,16)
end
```

Let the source vector be the concatenation of the contents of **v**A followed by the contents of **v**B.

Each signed integer word element in the source vector is converted to a 16-bit unsigned integer. If the value of the element is greater than (2^{16} - 1) the result saturates to (2^{16} - 1) and if the value is less than 0 the result saturates to 0. The result is placed into the corresponding half-word element of **v**D.

Other registers altered:

- SAT

Figure 6-90 shows the usage of the **vpkswus** instruction. Each of the four elements in the vectors, **v**A, and **v**B, is 32 bits long. Each of the eight elements in the vector **v**D, is 16 bits long.

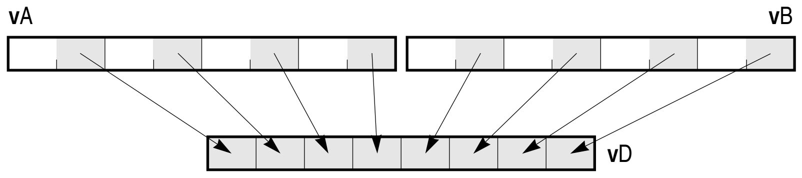


Figure 6-90. vpkswus—Pack Eight Signed Integer Elements (32-Bit) to Eight Unsigned Integer Elements (16-Bit)

vpkuhum vpkuhum

Vector Pack Unsigned Half Word Unsigned Modulo

vpkuhum **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	14
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 63 by 8
    vD_{i:i+7} ← (vA)_{(i*2)+8:(i*2)+15}
    vD_{i+64:i+71} ← (vB)_{(i*2)+8:(i*2)+15}
end
```

Let the source vector be the concatenation of the contents of **v**A followed by the contents of **v**B.

The low-order byte of each half-word element in the source vector is placed into the corresponding byte element of **v**D.

Other registers altered:

- None

Figure 6-91 shows the usage of the **vpkuhum** instruction. Each of the eight elements in the vectors, **v**A, and **v**B, is 16 bits long. Each of the sixteen elements in the vector **v**D, is 8 bits long.

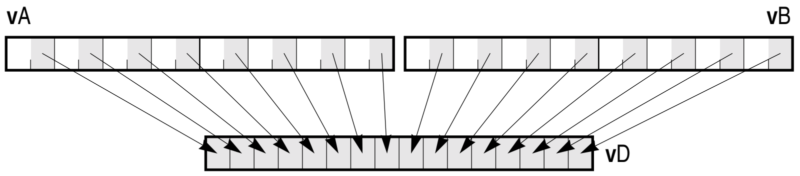


Figure 6-91. vpkuhum—Pack Sixteen Unsigned Integer Elements (16-Bit) to Sixteen Unsigned Integer Elements (8-Bit)

vpkuhus vpkuhus

Vector Pack Unsigned Half Word Unsigned Saturate

vpkuhus **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	142
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 63 by 8
    $vD_{i:i+7} \leftarrow$ UItoUIsat(($vA)_{i*2:(i*2)+15}$,8)
    $vD_{i+64:i+71} \leftarrow$ UItoUIsat(($vB)_{i*2:(i*2)+15}$,8)
end
```

Let the source vector be the concatenation of the contents of **v**A followed by the contents of **v**B.

Each unsigned integer half-word element in the source vector is converted to an 8-bit unsigned integer. If the value of the element is greater than (2^8 - 1) the result saturates to (2^8 - 1). The result is placed into the corresponding byte element of **v**D.

Other registers altered:

- SAT

Figure 6-92 shows the usage of the **vpkuhus** instruction. Each of the eight elements in the vectors, **v**A, and **v**B, is 16 bits long. Each of the sixteen elements in the vector **v**D, is 8 bits long.

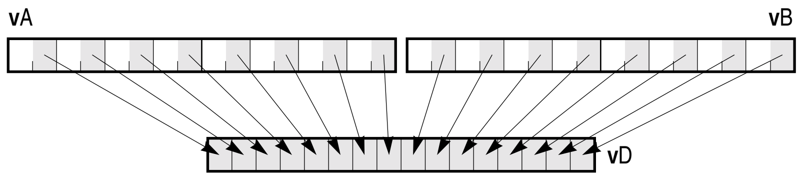


Figure 6-92. vpkuhus—Pack Sixteen Unsigned Integer Elements (16-Bit) to Sixteen Unsigned Integer Elements (8-Bit)

vpkuwum vpkuwum

Vector Pack Unsigned Word Unsigned Modulo

vpkuwum **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	78
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 63 by 16
    vD_{i:i+15} ← (vA)_{(i*2)+16:(i*2)+31}
    vD_{i+64:i+79} ← (vB)_{(i*2)+16:(i*2)+31}
end
```

Let the source vector be the concatenation of the contents of **v**A followed by the contents of **v**B.

The low-order half-word of each word element in the source vector is placed into the corresponding half-word element of **v**D.

Other registers altered:

- None

Figure 6-93 shows the usage of the **vpkuwum** instruction. Each of the four elements in the vectors, **v**A, and **v**B, is 32 bits long. Each of the eight elements in the vector **v**D, is 16 bits long.

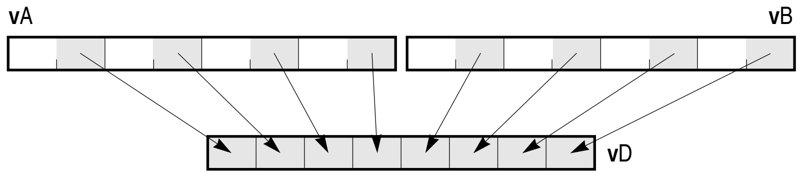


Figure 6-93. vpkuwum—Pack Eight Unsigned Integer Elements (32-Bit) to Eight Unsigned Integer Elements (16-Bit)

vpkuwus vpkuwus

Vector Pack Unsigned Word Unsigned Saturate

vpkuwus **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	206
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 63 by 16
    vD_{i:i+15} ← UItoUIsat((vA)_{i*2:(i*2)+31},16)
    vD_{i+64:i+79} ← UItoUIsat((vB)_{i*2:(i*2)+31},16)
end
```

Let the source vector be the concatenation of the contents of **v**A followed by the contents of **v**B.

Each unsigned integer word element in the source vector is converted to a 16-bit unsigned integer. If the value of the element is greater than (2^{16} - 1) the result saturates to (2^{16} - 1). The result is placed into the corresponding half-word element of **v**D.

Other registers altered:

- SAT

Figure 6-94 shows the usage of the **vpkuwus** instruction. Each of the four elements in the vectors, **v**A, and **v**B, is 32 bits long. Each of the eight elements in the vector **v**D, is 16 bits long.

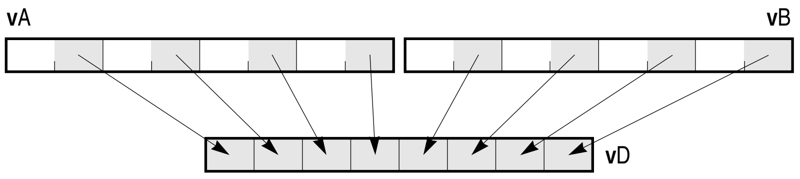


Figure 6-94. vpkuwum—Pack Eight Unsigned Integer Elements (32-Bit) to Eight Unsigned Integer Elements (16-Bit)

vrefp vrefp

Vector Reciprocal Estimate Floating Point

vrefp **v**D,**v**B Form: VX

04	**v**D	0_0000	**v**B	266
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 32
    x ← (vB)$_{i:i+31}$
    vD$_{i:i+31}$ ← 1/x
end
```

The single-precision floating-point estimate of the reciprocal of each single-precision floating-point element in **v**B is placed into the corresponding element of **v**D.

For results that are not a +0, -0, +∞, -∞, or QNaN, the estimate has a relative error in precision no greater than one part in 4096, that is:

$$\left| \frac{\text{estimate} - 1/x}{1/x} \right| \le \frac{1}{4096}$$

where *x* is the value of the element in **v**B. Note that the value placed into the element of **v**D may vary between implementations, and between different executions on the same implementation.

Operation with various special values of the element in **v**B is summarized below in Table 6-7.

Table 6-7. Special Values of the Element in vB

Value	Result
-∞	-0
-0	-∞
+0	+∞
+∞	+0
NaN	QNaN

If VSCR[NJ] = 1, every denormalized operand element is truncated to a 0 of the same sign before the operation is carried out, and each denormalized result element truncates to a 0 of the same sign.

Other registers altered:

- None

Figure 6-95 shows the usage of the **vrefp** instruction. Each of the four elements in the vectors **v**B and **v**D is 32 bits long.

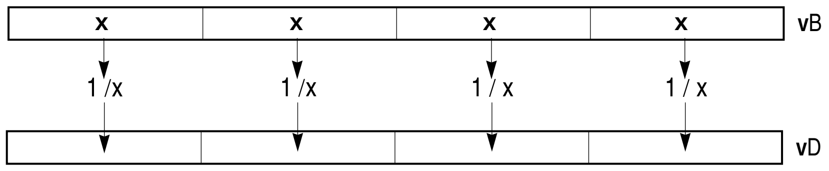


Figure 6-95. vrefp—Reciprocal Estimate of Four Floating-Point Elements (32-Bit)

vrfim vrfim

Vector Round to Floating-Point Integer toward Minus Infinity

vrfim **v**D,**v**B Form: VX

04	**v**D	0_0000	**v**B	714
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 32
    vD_{i:i+31} ← RndToFPInt32Floor((vB)_{i:i+31})
end
```

Each single-precision floating-point word element in **v**B is rounded to a single-precision floating-point integer, using the rounding mode Round toward -Infinity, and placed into the corresponding word element of **v**D.

Other registers altered:

- None

Figure 6-96 shows the usage of the **vrfim** instruction. Each of the four elements in the vectors **v**B and **v**D is 32 bits long.

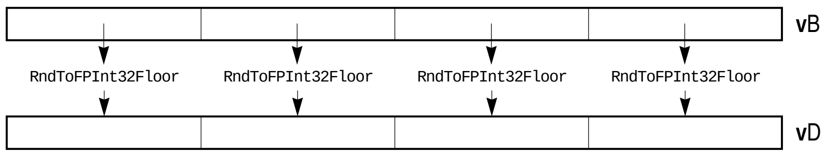


Figure 6-96. vrfim— Round to Minus Infinity of Four Floating-Point Integer Elements (32-Bit)

vrfin vrfin

Vector Round to Floating-Point Integer Nearest

vrfin **v**D,**v**B Form: VX

04	**v**D	0_0000	**v**B	522
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 32
    $\mathbf{v}D_{i:i+31} \leftarrow$ RndToFPInt32Near(($\mathbf{v}$B)$_{i:i+31}$)
end
```

Each single-precision floating-point word element in **v**B is rounded to a single-precision floating-point integer, using the rounding mode Round to Nearest, and placed into the corresponding word element of **v**D.

Note the result is independent of VSCR[NJ].

Other registers altered:

- None

Figure 6-97 shows the usage of the **vrfin** instruction. Each of the four elements in the vectors **v**B and **v**D is 32 bits long.

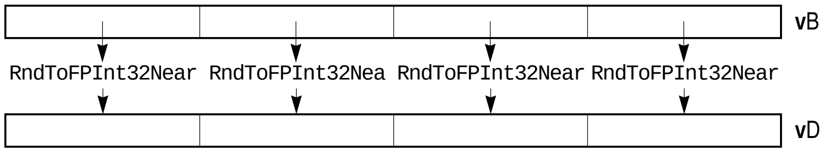


Figure 6-97. vrfin—Nearest Round to Nearest of Four Floating-Point Integer Elements (32-Bit)

vrfip vrfip

Vector Round to Floating-Point Integer toward Plus Infinity

vrfip **v**D,**v**B Form: VX

04	**v**D	0_0000	**v**B	650
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 32
    vD_{i:i+31} ← RndToFPInt32Ceil((vB)_{i:i+31})
end
```

Each single-precision floating-point word element in **v**B is rounded to a single-precision floating-point integer, using the rounding mode Round toward +Infinity, and placed into the corresponding word element of **v**D.

If VSCR[NJ] = 1, every denormalized operand element is truncated to 0 before the comparison is made.

Other registers altered:

- None

Figure 6-98 shows the usage of the **vrfip** instruction. Each of the four elements in the vectors **v**B and **v**D is 32 bits long.

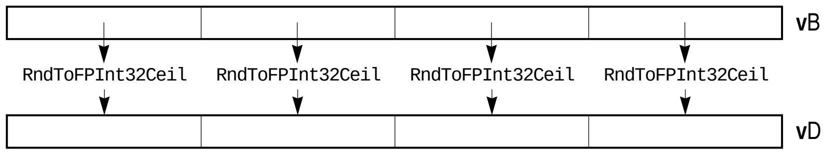


Figure 6-98. vrfip—Round to Plus Infinity of Four Floating-Point Integer Elements (32-Bit)

vrfiz vrfiz

Vector Round to Floating-Point Integer toward Zero

vrfiz **v**D,**v**B Form: VX

04	**v**D	0_0000	**v**B	586
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 32
    vD$_{i:i+31}$ ← RndToFPInt32Trunc((vB)$_{i:i+31}$)
end
```

Each single-precision floating-point word element in **v**B is rounded to a single-precision floating-point integer, using the rounding mode Round toward Zero, and placed into the corresponding word element of **v**D.

Note, the result is independent of VSCR[NJ].

Other registers altered:

- None

Figure 6-99 shows the usage of the **vrfiz** instruction. Each of the four elements in the vectors **v**B and **v**D is 32 bits long.

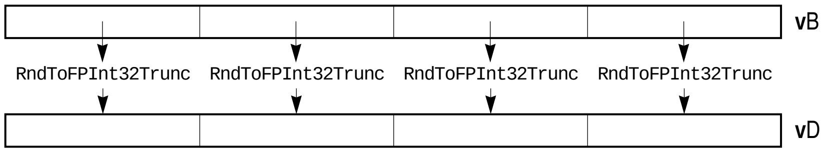


Figure 6-99. vrfiz—Round-to-Zero of Four Floating-Point Integer Elements (32-Bit)

vrlb vrlb

Vector Rotate Left Integer Byte

vrlb **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	4
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 8
    sh ← (vB)i+5:i+7
    vDi:i+7 ← ROTL((vA)i:i+7,sh)
end
```

Each element is a byte. Each element in **v**A is rotated left by the number of bits specified in the low-order 3 bits of the corresponding element in **v**B. The result is placed into the corresponding element of **v**D.

Other registers altered:

- None

Figure 6-100 shows the usage of the **vrlb** instruction. Each of the sixteen elements in the vectors, **v**A, **v**B, and **v**D, is 8 bits long.

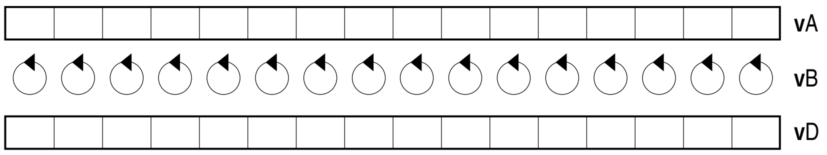


Figure 6-100. vrlb—Left Rotate of Sixteen Integer Elements (8-Bit)

vrlh vrlh

Vector Rotate Left Integer Half Word

vrlh **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	68
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 16
    sh ← (vB)$_{i+12:i+15}$
    vD$_{i:i+15}$ ← ROTL((vA)$_{i:i+15}$,sh)
end
```

Each element is a half word

Each element in **v**A is rotated left by the number of bits specified in the low-order 4 bits of the corresponding element in **v**B. The result is placed into the corresponding element of **v**D.

Other registers altered:

- None

Figure 6-101 shows the usage of the **vrlh** instruction. Each of the eight elements in the vectors, **v**A, **v**B, and **v**D, is 16 bits long.

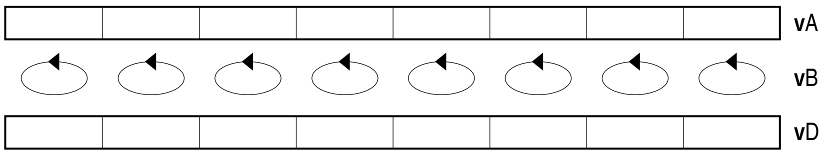


Figure 6-101. vrlh—Left Rotate of Eight Integer Elements (16-Bit)

vrlw vrlw

Vector Rotate Left Integer Word

vrlw **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	132
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 32
    sh ← (vB)_{i+27:i+31}
    vD_{i:i+31} ← ROTL((vA)_{i:i+31},sh)
end
```

Each element is a word. Each element in **v**A is rotated left by the number of bits specified in the low-order 5 bits of the corresponding element in **v**B. The result is placed into the corresponding element of **v**D.

Other registers altered:

- None

Figure 6-102 shows the usage of the **vrlw** instruction. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits long.

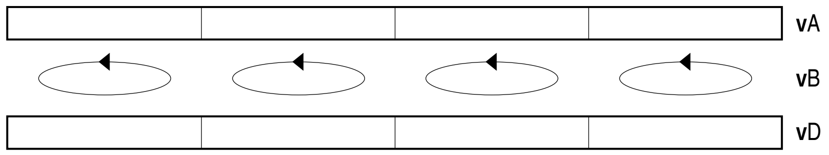


Figure 6-102. vrlw—Left Rotate of Four Integer Elements (32-Bit)

vrsqrtefp vrsqrtefp

Vector Reciprocal Square Root Estimate Floating Point

vrsqrtefp **v**D,**v**B Form: VX

04	**v**D	0_0000	**v**B	330
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 32
    x ← (vB)$_{i:i+31}$
    vD$_{i:i+31}$ ← 1 $\div_{fp}$ ($\sqrt{}_{fp}$(x))
end
```

The single-precision estimate of the reciprocal of the square root of each single-precision element in **v**B is placed into the corresponding word element of **v**D. The estimate has a relative error in precision no greater than one part in 4096, as explained below:

$$\left|\frac{\text{estimate} - 1/\sqrt{x}}{1/\sqrt{x}}\right| \le \frac{1}{4096}$$

where *x* is the value of the element in **v**B. Note that the value placed into the element of **v**D may vary between implementations and between different executions on the same implementation. Operation with various special values of the element in **v**B is summarized below in Table 6-8.

Table 6-8. Special Values of the Element in vB

Value	Result	Value	Result
-∞	QNaN	+0	+∞
less than 0	QNaN	+∞	+0
-0	-∞	NaN	QNaN

Other registers altered:

- None

Figure 6-103 shows the usage of the **vrsqrtefp** instruction. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits long.

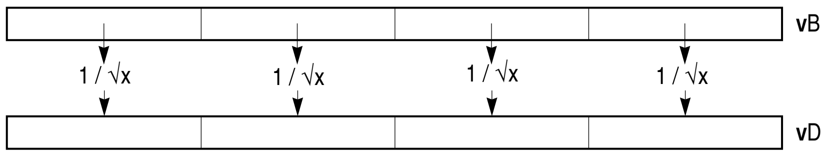


Figure 6-103. vrsqrtefp—Reciprocal Square Root Estimate of Four Floating-Point Elements (32-Bit)

vsel vsel

Vector Conditional Select

vsel **v**D,**v**A,**v**B,**v**C Form: VA

04	**v**D	**v**A	**v**B	**v**C	42
0 5	6 10	11 15	16 20	21 25	26 31

```
do i=0 to 127

    if (vC)_i=0 then vD_i ← (vA)_i
     else vD_i ← (vB)_i

end
```

For each bit in **v**C that contains the value 0, the corresponding bit in **v**A is placed into the corresponding bit of **v**D. For each bit in **v**C that contains the value 1, the corresponding bit in **v**B is placed into the corresponding bit of **v**D.

Other registers altered:

- None

Figure 6-104 shows the usage of the **vsel** instruction. Each of the vectors, **v**A, **v**B, **v**C, and **v**D, is 128 bits long.

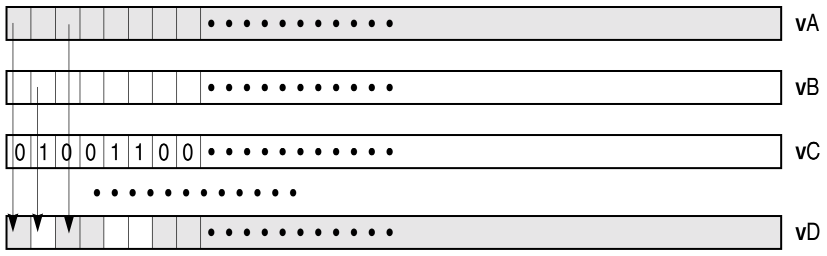


Figure 6-104. vsel—Bitwise Conditional Select of Vector Contents(128-bit)

vsl vsl

Vector Shift Left

vsl **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	452
0 5	6 10	11 15	16 20	21 31

```
sh ← (vB)125:127
t ← 1
do i = 0 to 127 by 8
    t ← t & ((vB)i+5:i+7 = sh)
    if t = 1 then vD ← (vA) <<ui sh
    else vD ← undefined
end
```

The contents of **v**A are shifted left by the number of bits specified in **v**B[125–127]. Bits shifted out of bit 0 are lost. Zeros are supplied to the vacated bits on the right. The result is placed into **v**D.

The contents of the low-order three bits of all byte elements in **v**B must be identical to **v**B[125–127]; otherwise the value placed into **v**D is undefined.

Other registers altered:

- None

Figure 6-105 shows the usage of the **vsl** instruction.

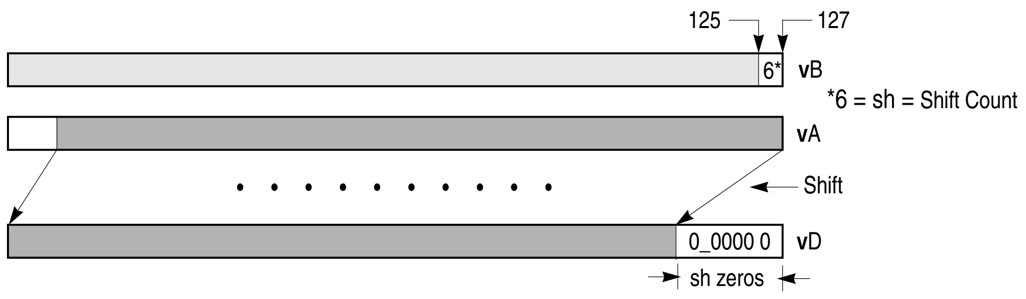


Figure 6-105. vsl—Shift Bits Left in Vector (128-Bit)

vslb vslb

Vector Shift Left Integer Byte

vslb **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	260
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 8
    sh ← (vB)$_{i+5):i+7}$
    vD$_{i:i+7}$ ← (vA)$_{i:i+7}$ <<$_{ui}$ sh
end
```

Each element is a byte. Each element in **v**A is shifted left by the number of bits specified in the low-order 3 bits of the corresponding element in **v**B. Bits shifted out of bit 0 of the element are lost. Zeros are supplied to the vacated bits on the right. The result is placed into the corresponding element of **v**D.

Other registers altered:

- None

Figure 6-106 shows the usage of the **vslb** instruction. Each of the sixteen elements in the vectors, **v**A, **v**B, and **v**D, is 8 bits long.

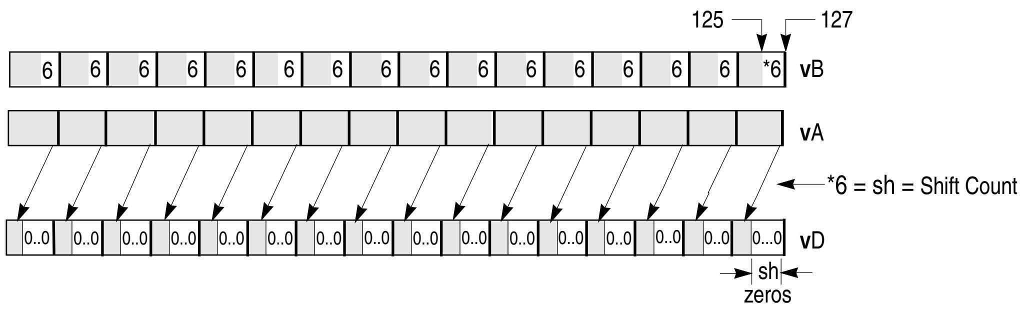


Figure 6-106. vslb—Shift Bits Left in Sixteen Integer Elements (8-Bit)

vsldoi vsldoi

Vector Shift Left Double by Octet Immediate

vsldoi **v**D, **v**A, **v**B, SHB Form: VA

04	**v**D	**v**A	**v**B	0	SH	44
0 5	6 10	11 15	16 20	21	22 25	26 31

```
vD ← ((vA) ‖ (vB)) <<ui (SHB ‖ 0b000)
```

Let the source vector be the concatenation of the contents of **v**A followed by the contents of **v**B. Bytes SHB:SHB+15 of the source vector are placed into **v**D.

Other registers altered:

- None

Figure 6-107 shows the usage of the **vsldoi** instruction. Each of the sixteen elements in the vectors, **v**A, **v**B, and **v**D, is 8 bits long.

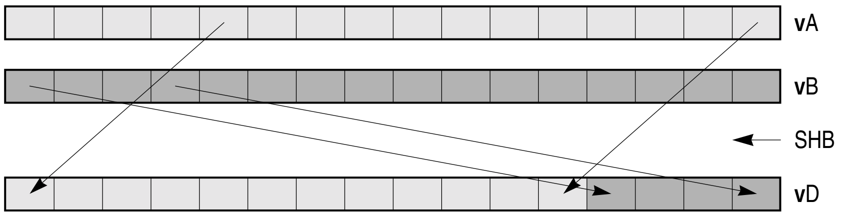


Figure 6-107. vsldoi—Shift Left by Bytes Specified

vslh vslh

Vector Shift Left Integer Half Word

vslh **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	324
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 16
    sh ← $(\mathbf{v}B)_{i+12:i+15}$
    $\mathbf{v}D_{i:i+15}$ ← $(\mathbf{v}A)_{i:i+15}$ $<<_{ui}$ sh
end
```

Each element is a half word. Each element in **v**A is shifted left by the number of bits specified in the low-order 4 bits of the corresponding element in **v**B. Bits shifted out of bit 0 of the element are lost. Zeros are supplied to the vacated bits on the right. The result is placed into the corresponding element of **v**D.

Other registers altered:

- None

Figure 6-108 shows the usage of the **vslh** instruction. Each of the eight elements in the vectors, **v**A, **v**B, and **v**D, is 16 bits long.

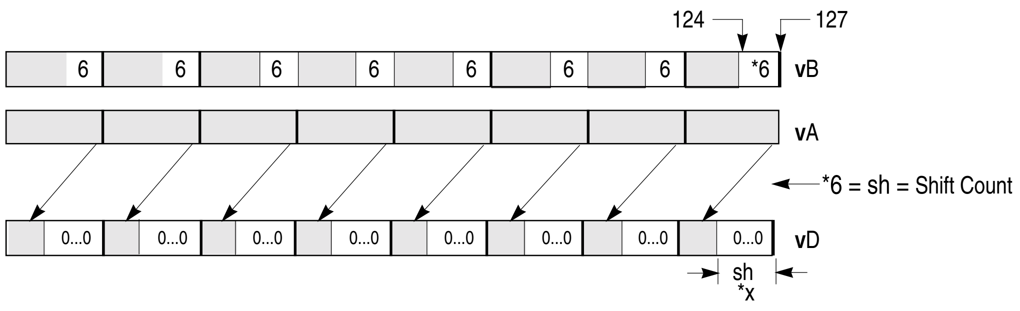


Figure 6-108. vslh—Shift Bits Left in Eight Integer Elements (16-Bit)

vslo vslo

Vector Shift Left by Octet

vslo **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	1036
0 5	6 10	11 15	16 20	21 31

```
shb ← (vB)121:124
vD  ← (vA) <<ui (shb || 0b000)
```

The contents of **v**A are shifted left by the number of bytes specified in **v**B[121–124]. Bytes shifted out of byte 0 are lost. Zeros are supplied to the vacated bytes on the right. The result is placed into **v**D.

Other registers altered:

- None

Figure 6-109 shows the usage of the **vslo** instruction.

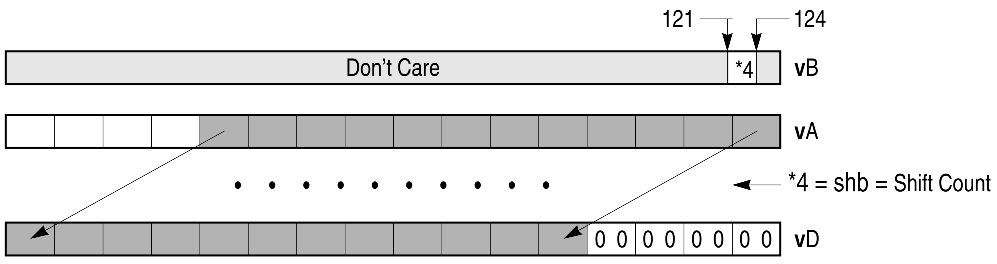


Figure 6-109. vslo—Left Byte Shift of Vector (128-Bit)

vslw vslw

Vector Shift Left Integer Word

vslw **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	388
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 32
    sh ← (vB)$_{i+27:i+31}$
    vD$_{i:i+31}$ ← (vA)$_{i:i+31}$ <<$_{ui}$ sh
end
```

Each element is a word. Each element in **v**A is shifted left by the number of bits specified in the low-order 5 bits of the corresponding element in **v**B. Bits shifted out of bit 0 of the element are lost. Zeros are supplied to the vacated bits on the right. The result is placed into the corresponding element of **v**D.

Other registers altered:

- None

Figure 6-110 shows the usage of the **vslw** instruction. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits long.

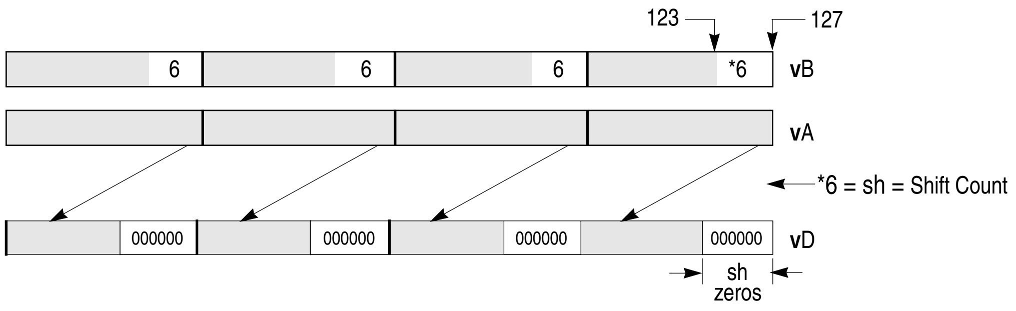


Figure 6-110. vslw—Shift Bits Left in Four Integer Elements (32-Bit)

vspltb vspltb

Vector Splat Byte

vspltb **v**D,**v**B,UIMM Form: VX

04	**v**D	UIMM	**v**B	524
0 5	6 10	11 15	16 20	21 31

```
b ← UIMM*8
do i=0 to 127 by 8
    vD_i:i+7 ← (vB)_b:b+7
end
```

Each element of **vspltb** is a byte.

The contents of element UIMM in **v**B are replicated into each element of **v**D.

Other registers altered:

- None

Programming note: The vector splat instructions can be used in preparation for performing arithmetic for which one source vector is to consist of elements that all have the same value (for example, multiplying all elements of a vector register by a constant).

Figure 6-111 shows the usage of the **vspltb** instruction. Each of the sixteen elements in the vectors **v**B and **v**D is 8 bits long.

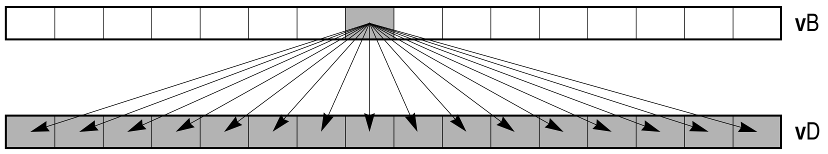


Figure 6-111. vspltb—Copy Contents to Sixteen Elements (8-Bit)

vsplth vsplth

Vector Splat Half Word

vsplth **v**D,**v**B,UIMM Form: VX

04	**v**D	UIMM	**v**B	588
0 5	6 10	11 15	16 20	21 31

```
b ← UIMM*16
do i=0 to 127 by 16
    vD_{i:i+15} ← (vB)_{b:b+15}
end
```

Each element of **vsplth** is a half word.

The contents of element UIMM in **v**B are replicated into each element of **v**D.

Other registers altered:

- None

Programming note: The vector splat instructions can be used in preparation for performing arithmetic for which one source vector is to consist of elements that all have the same value (for example, multiplying all elements of a vector register by a constant).

Figure 6-112 shows the usage of the **vsplth** instruction. Each of the eight elements in the vectors **v**B and **v**D is 16 bits long.

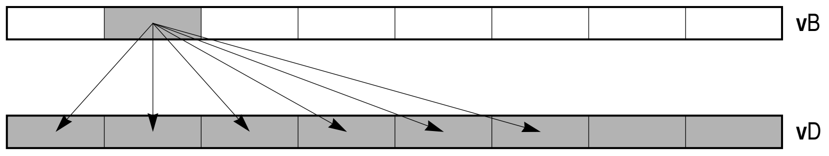


Figure 6-112. vsplth—Copy Contents to Eight Elements (16-Bit)

vspltisb vspltisb

Vector Splat Immediate Signed Byte

vspltisb **v**D,SIMM Form: VX

04	**v**D	SIMM	0000_0	780
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 8
    $vD_{i:i+7}$ ← SignExtend(SIMM,8)
end
```

Each element of **vspltisb** is a byte.

The value of the SIMM field, sign-extended to the length of the element, is replicated into each element of **v**D.

Other registers altered:

- None

Figure 6-113 shows the usage of the **vspltisb** instruction. Each of the sixteen elements in the vector, **v**D, is 8 bits long.

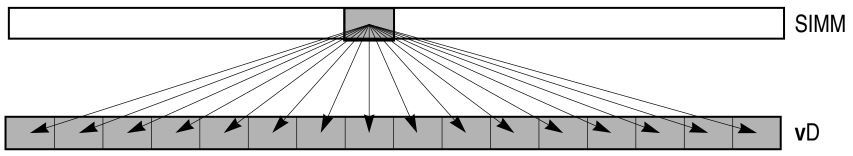


Figure 6-113. vspltisb—Copy Value into Sixteen Signed Integer Elements (8-Bit)

vspltish vspltish

Vector Splat Immediate Signed Half Word

vspltish **v**D,SIMM Form: VX

04	**v**D	SIMM	0000_0	844
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 16
    vD_{i:i+15} ← SignExtend(SIMM,16)
end
```

Each element of **vspltish** is a half word.

The value of the SIMM field, sign-extended to the length of the element, is replicated into each element of **v**D.

Other registers altered:

- None

Figure 6-114 shows the usage of the **vspltish** instruction. Each of the eight elements in the vectors, **v**A, **v**B, and **v**D, is 16 bits long.

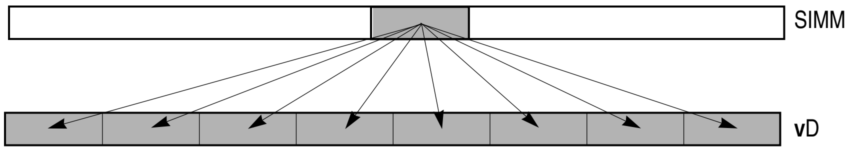


Figure 6-114. vspltish—Copy Value to Eight Signed Integer Elements (16-Bit)

vspltisw vspltisw

Vector Splat Immediate Signed Word

vspltisw **v**D,SIMM Form: VX

04	**v**D	SIMM	0000_0	908
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 32
    vD_{i:i+31} ← SignExtend(SIMM,32)
end
```

Each element of **vspltisw** is a word.

The value of the SIMM field, sign-extended to the length of the element, is replicated into each element of **v**D.

Other registers altered:

- None

Figure 6-115 shows the usage of the **vspltisw** instruction. Each of the four elements in the vector, and **v**D, is 32 bits long.

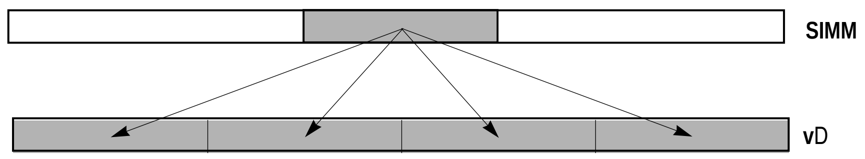


Figure 6-115. vspltisw—Copy Value to Four Signed Elements (32-Bit)

vspltw vspltw

Vector Splat Word

vspltw **v**D,**v**B,UIMM Form: VX

04	**v**D	UIMM	**v**B	652
0 5	6 10	11 15	16 20	21 31

```
b ← UIMM*32
do i=0 to 127 by 32
    vD_i:i+31 ← (vB)_b:b+31
end
```

Each element of **vspltw** is a word.

The contents of element UIMM in **v**B are replicated into each element of **v**D.

Other registers altered:

- None

Programming note: The Vector Splat instructions can be used in preparation for performing arithmetic for which one source vector is to consist of elements that all have the same value (for example, multiplying all elements of a Vector Register by a constant).

Figure 6-116 shows the usage of the **vspltw** instruction. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits long.

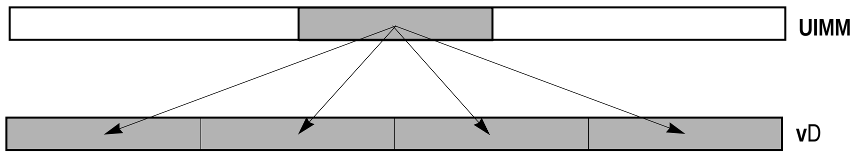


Figure 6-116. vspltw—Copy contents to Four Elements (32-Bit)

vsr vsr

Vector Shift Right

vsr **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	708
0 5	6 10	11 15	16 20	21 31

```
sh ← (vB)$_{125:127}$
t ← 1
do i = 0 to 127 by 8

    t ← t & ((vB)$_{i+5:i+7}$ = sh)
    if t = 1 then vD ← (vA) >>$_{ui}$ sh
    elsevD ← undefined

end
```

Let sh = **v**B[125–127]; sh is the shift count in bits (0≤sh≤7). The contents of **v**A are shifted right by sh bits. Bits shifted out of bit 127 are lost. Zeros are supplied to the vacated bits on the left. The result is placed into **v**D.

The contents of the low-order three bits of all byte elements in register **v**B must be identical to **v**B[125-127]; otherwise the value placed into register **v**D is undefined.

Other registers altered:

- None

Programming notes:

A pair of **vslo** and **vsl** or **vsro** and **vsr** instructions, specifying the same shift count register, can be used to shift the contents of a vector register left or right by the number of bits (0–127) specified in the shift count register. The following example shifts the contents of **v**X left by the number of bits specified in **v**Y and places the result into **v**Z.

```
vslo     VZ,VX,VY
vsl      VZ,VZ,VY
```

A double-register shift by a dynamically specified number of bits (0–127) can be performed in six instructions. The following example shifts (**v**W) || (**v**X) left by the number of bits specified in **v**Y and places the high-order 128 bits of the result into **v**Z.

```
vslo     t1,VW,VY  #shift high-order reg left
vsl      t1,t1,VY
vsububm  t3,V0,VY  #adjust shift count ((V0)=0)
vsro     t2,VX,t3  #shift low-order reg right
vsr      t2,t2,t3
vor      VZ,t1,t2  #merge to get final result
```

Figure 6-117 shows the usage of the **vsr** instruction. Each of the sixteen elements in the vectors, **v**A, **v**B, and **v**D, is 8 bits long.

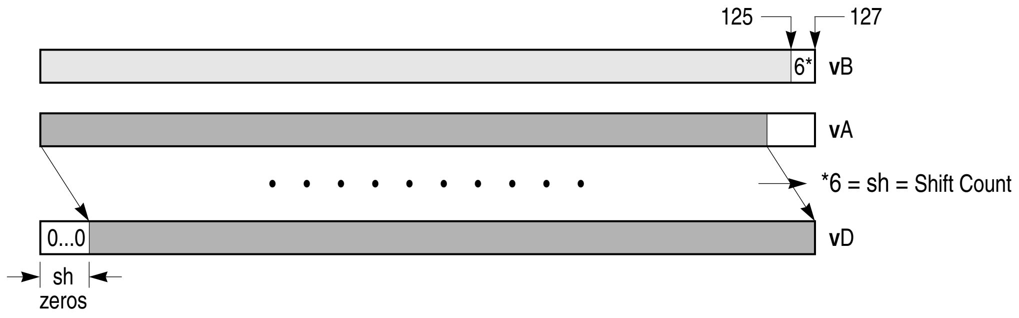


Figure 6-117. vsr—Shift Bits Right for Vectors (128-Bit)

vsrab vsrab

Vector Shift Right Algebraic Byte

vsrab **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	772
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 8
    sh ← (vB)$_{i+2:i+7}$
    vD$_{i:i+7}$ ← (vA)$_{i:i+7}$ >>$_{si}$ sh
end
```

Each element is a byte. Each element in **v**A is shifted right by the number of bits specified in the low-order 3 bits of the corresponding element in **v**B. Bits shifted out of bit n-1 of the element are lost. Bit 0 of the element is replicated to fill the vacated bits on the left. The result is placed into the corresponding element of **v**D.

Other registers altered:

- None

Figure 6-118 shows the usage of the **vsrab** instruction. Each of the sixteen elements in the vectors, **v**A, and **v**D, is 8 bits long.

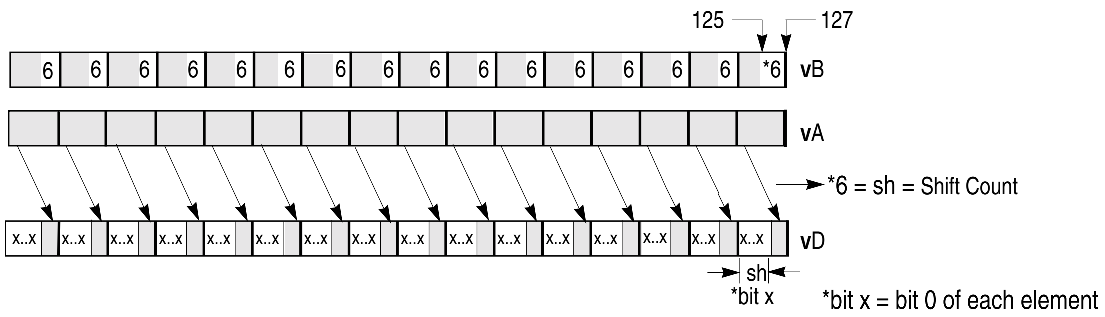


Figure 6-118. vsrab—Shift Bits Right in Sixteen Integer Elements (8-Bit)

vsrah vsrah

Vector Shift Right Algebraic Half Word

vsrah **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	836
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 16
    sh ← (vB)$_{i+12:i+15}$
    vD$_{i:i+15}$ ← (vA)$_{i:i+15}$ >>$_{si}$ sh
end
```

Each element is a half word. Each element in **v**A is shifted right by the number of bits specified in the low-order 4 bits of the corresponding element in **v**B. Bits shifted out of bit 15 of the element are lost. Bit 0 of the element is replicated to fill the vacated bits on the left. The result is placed into the corresponding element of **v**D.

Other registers altered:

- None

Figure 6-119 shows the usage of the **vsrah** instruction. Each of the eight elements in the vectors, **v**A, and **v**D, is 16 bits long.

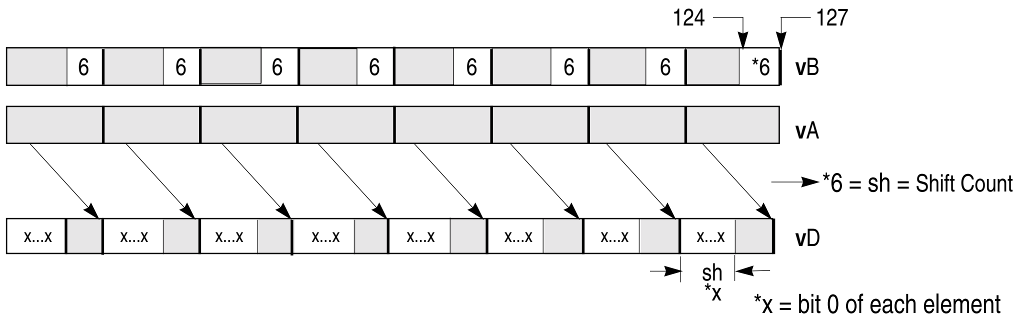


Figure 6-119. vsrah—Shift Bits Right for Eight Integer Elements (16-Bit)

vsraw vsraw

Vector Shift Right Algebraic Word

vsraw **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	900
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 32
    sh ← $(\mathbf{v}B)_{i+27:i+31}$
    $\mathbf{v}D_{i:i+31}$ ← $(\mathbf{v}A)_{i:i+31} >>_{si}$ sh
end
```

Each element is a word. Each element in **v**A is shifted right by the number of bits specified in the low-order 5 bits of the corresponding element in **v**B. Bits shifted out of bit 31 of the element are lost. Bit 0 of the element is replicated to fill the vacated bits on the left. The result is placed into the corresponding element of **v**D.

Other registers altered:

- None

Figure 6-120 shows the usage of the **vsraw** instruction. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits long.

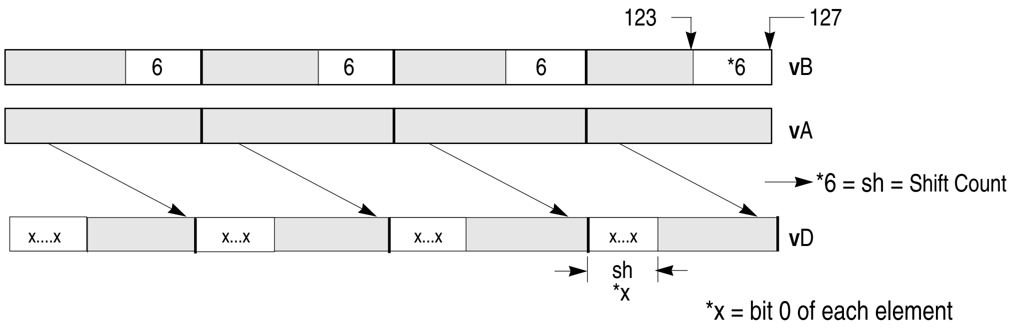


Figure 6-120. vsraw—Shift Bits Right in Four Integer Elements (32-Bit)

vsrb vsrb

Vector Shift Right Byte

vsrb **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	516
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 8
    sh ← (vB)_{i+5:i+7}
    vD_{i:i+7} ← (vA)_{i:i+7} >>_{ui} sh
end
```

Each element is a byte. Each element in **v**A is shifted right by the number of bits specified in the low-order 3 bits of the corresponding element in **v**B. Bits shifted out of bit 7 of the element are lost. Zeros are supplied to the vacated bits on the left. The result is placed into the corresponding element of **v**D.

Other registers altered:

- None

Figure 6-121 shows the usage of the **vsrb** instruction. Each of the sixteen elements in the vectors, **v**A, and **v**D, is 8 bits long.

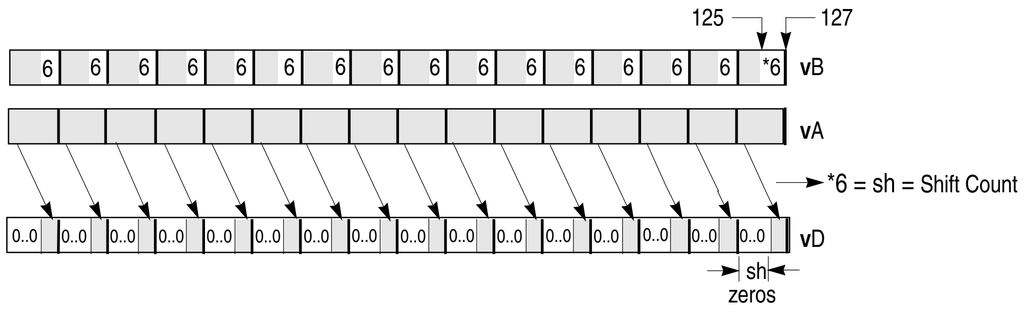


Figure 6-121. vsrb—Shift Bits Right in Sixteen Integer Elements (8-Bit)

vsrh vsrh

Vector Shift Right Half Word

vsrh **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	580
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 16
    sh ← (vB)$_{i+12:i+15}$
    vD$_{i:i+15}$ ← (vA)$_{i:i+15}$ >>$_{ui}$ sh
end
```

Each element is a half word. Each element in **v**A is shifted right by the number of bits specified in the low-order 4 bits of the corresponding element in **v**B. Bits shifted out of bit 15 of the element are lost. Zeros are supplied to the vacated bits on the left. The result is placed into the corresponding element of **v**D.

Other registers altered:

- None

Figure 6-122 shows the usage of the **vsrh** instruction. Each of the eight elements in the vectors, **v**A, and **v**D, is 16 bits long.

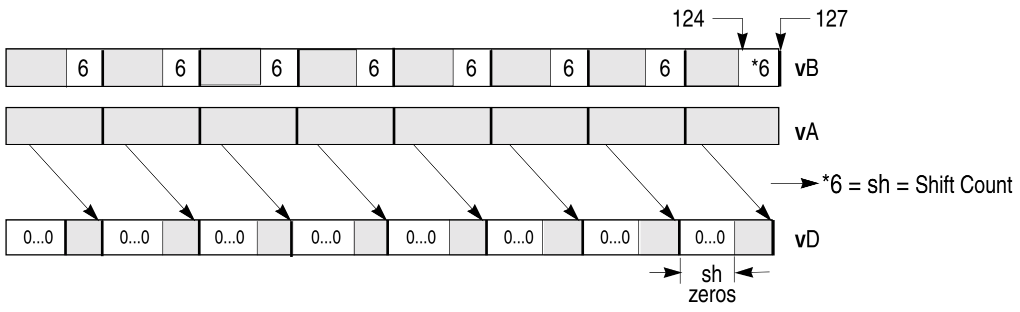


Figure 6-122. vsrh—Shift Bits Right for Eight Integer Elements (16-Bit)

vsro vsro

Vector Shift Right Octet

vsro **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	1100
0 5	6 10	11 15	16 20	21 31

```
shb ← (vB)121:124
vD  ← (vA) >>ui (shb || 0b000)
```

The contents of **v**A are shifted right by the number of bytes specified in **v**B[121–124]. Bytes shifted out of **v**A are lost. Zeros are supplied to the vacated bytes on the left. The result is placed into **v**D.

Other registers altered:

- None

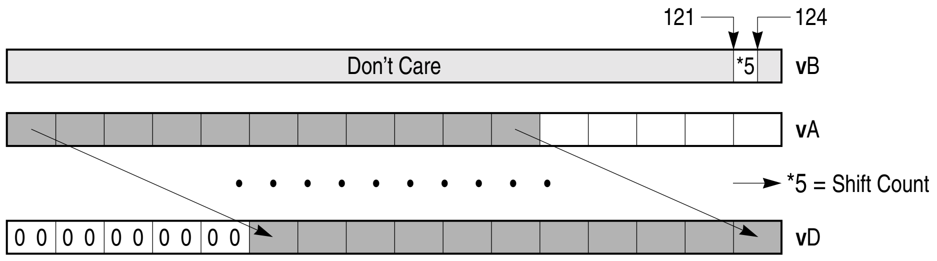


Figure 6-123. vsro—Vector Shift Right Octet

vsrw   vsrw

Vector Shift Right Word

vsrw   **v**D,**v**A,**v**B   Form: VX

04	**v**D	**v**A	**v**B	644
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 32

    sh ← (vB)_{i+(27):i+31}
    vD_{i:i+31} ← (vA)_{i:i+31} >>_{ui} sh

end
```

Each element is a word. Each element in **v**A is shifted right by the number of bits specified in the low-order 5 bits of the corresponding element in **v**B. Bits shifted out of bit 31 of the element are lost. Zeros are supplied to the vacated bits on the left. The result is placed into the corresponding element of **v**D.

Other registers altered:

- None

Figure 6-124 shows the usage of the **vsrw** instruction. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits long.

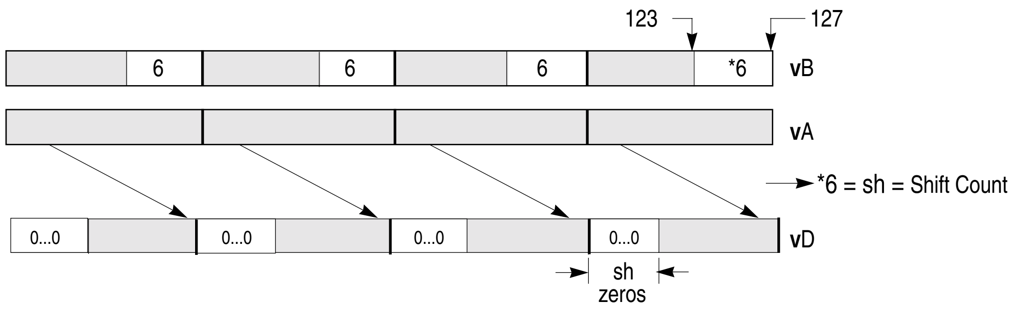


Figure 6-124. vsrw—Shift Bits Right in Four Integer Elements (32-Bit)

vsubcuw vsubcuw

Vector Subtract Carryout Unsigned Word

vsubcuw **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	1408
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 32
    $aop_{0:32}$ ← ZeroExtend(($\mathbf{v}A)_{i:i+31}$,33)
    $bop_{0:32}$ ← ZeroExtend(($\mathbf{v}B)_{i:i+31}$,33)
    $temp_{0:32}$ ← $aop_{0:32}$ $+_{int}$ ¬$bop_{0:32}$ $+_{int}$ 1
    $\mathbf{v}D_{i:i+31}$ ← ZeroExtend($temp_0$,32)
end
```

Each unsigned-integer word element in **v**B is subtracted from the corresponding unsigned-integer word element in **v**A. The complement of the borrow out of bit 0 of the 32-bit difference is zero-extended to 32 bits and placed into the corresponding word element of **v**D.

Other registers altered:

- None

Figure 6-125 shows the usage of the **vsubcuw** instruction. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits long.

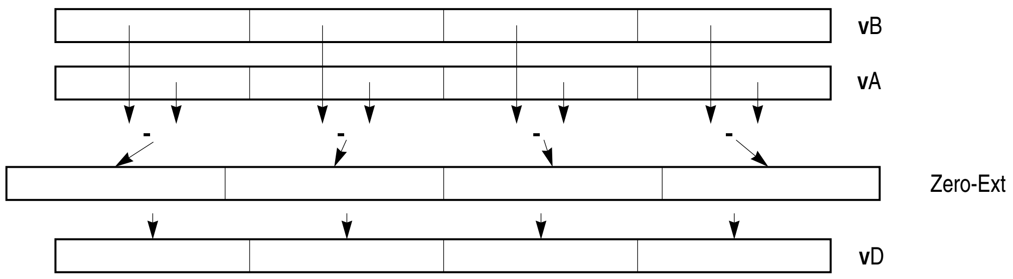


Figure 6-125. vsubcuw—Subtract Carryout of Four Unsigned Integer Elements (32-Bit)

vsubfp vsubfp

Vector Subtract Floating Point

vsubfp **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	74
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 32
    $vD_{i:i+31} \leftarrow$ RndToNearFP32(($vA$)$_{i:i+31}$ $-_{fp}$ ($vB$)$_{i:i+31}$)
end
```

Each single-precision floating-point word element in **v**B is subtracted from the corresponding single-precision floating-point word element in **v**A. The result is rounded to the nearest single-precision floating-point number and placed into the corresponding word element of **v**D.

If VSCR[NJ] = 1, every denormalized operand element is truncated to a 0 of the same sign before the operation is carried out, and each denormalized result element truncates to a 0 of the same sign.

Other registers altered:

- None

Figure 6-126 shows the usage of the **vsubfp** instruction. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits long.

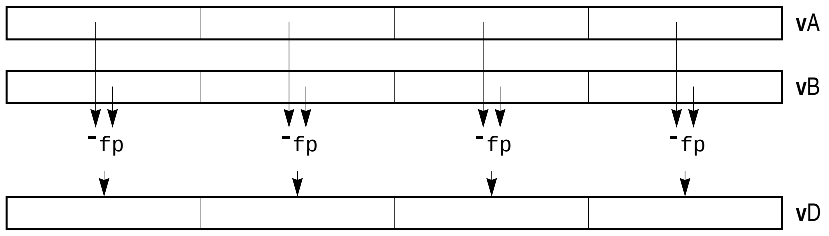


Figure 6-126. vsubfp—Subtract Four Floating Point Elements (32-Bit)

vsubsbs vsubsbs

Vector Subtract Signed Byte Saturate

vsubsbs **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	1792
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 8
    aop$_{0:8}$← SignExtend((vA)$_{i:i+7}$,9)
    bop$_{0:8}$← SignExtend((vB)$_{i:i+7}$,9)
    temp$_{0:8}$← aop$_{0:8}$ +$_{int}$ –bop$_{0:8}$ +$_{int}$ 1
    vD$_{i:i+7}$← SItoSIsat(temp$_{0:8}$,8)
end
```

Each element is a byte. Each signed-integer element in **v**B is subtracted from the corresponding signed-integer element in **v**A.

If the intermediate result is greater than (2^7-1) it saturates to (2^7-1) and if it is less than -2^7 it saturates to -2^7, where 8 is the length of the element.

The signed-integer result is placed into the corresponding element of **v**D.

Other registers altered:

- SAT

Figure 6-127 shows the usage of the **vsubsbs** instruction. Each of the sixteen elements in the vectors, **v**A, **v**B, and **v**D, is 8 bits long.

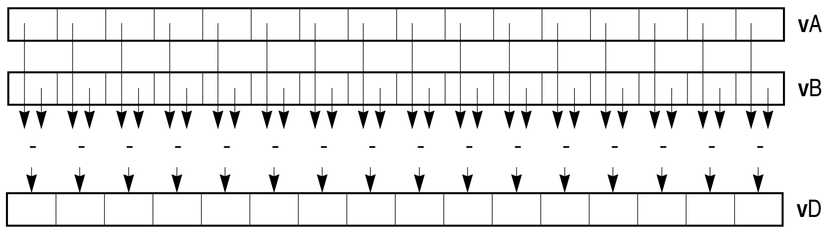


Figure 6-127. vsubsbs—Subtract Sixteen Signed Integer Elements (8-Bit)

vsubshs vsubshs

Vector Subtract Signed Half Word Saturate

vsubshs **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	1856
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 16
    aop$_{0:16}$← SignExtend((vA)$_{i:i+15}$,17)
    bop$_{0:16}$← SignExtend((vB)$_{i:i+15}$,17)
    temp$_{0:16}$← aop$_{0:16}$ +$_{int}$ -bop$_{0:16}$ +$_{int}$ 1
    vD$_{i:i+15}$← SItoSIsat(temp$_{0:16}$,16)
end
```

Each element is a half word. Each signed-integer element in **v**B is subtracted from the corresponding signed-integer element in **v**A.

If the intermediate result is greater than (2^{15}-1) it saturates to (2^{15}-1) and if it is less than -2^{15} it saturates to -2^{15}, where 16 is the length of the element.

The signed-integer result is placed into the corresponding element of **v**D.

Other registers altered:

- SAT

Figure 6-128 shows the usage of the **vsubshs** instruction. Each of the eight elements in the vectors, **v**A, **v**B, and **v**D, is 16 bits long.

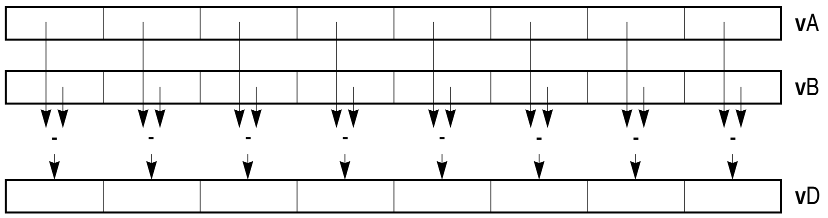


Figure 6-128. vsubshs—Subtract Eight Signed Integer Elements (16-Bit)

vsubsws vsubsws

Vector Subtract Signed Word Saturate

vsubsws **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	1920
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 32
    aop$_{0:32}$← SignExtend((vA)$_{i:i+31}$,33)
    bop$_{0:32}$← SignExtend((vB)$_{i:i+31}$,33)
    temp$_{0:32}$← aop$_{0:32}$ +$_{int}$ –bop$_{0:32}$ +$_{int}$ 1
    vD$_{i:i+31}$← SItoSIsat(temp$_{0:32}$,32)
end
```

Each element is a word. Each signed-integer element in **v**B is subtracted from the corresponding signed-integer element in **v**A.

If the intermediate result is greater than (2^{31}-1) it saturates to (2^{31}-1) and if it is less than -2^{31} it saturates to -2^{31}, where 32 is the length of the element.

The signed-integer result is placed into the corresponding element of **v**D.

Other registers altered:

- SAT

Figure 6-129 shows the usage of the **vsubsws** instruction. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits long.

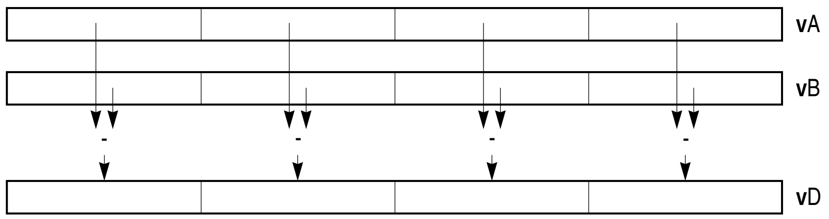


Figure 6-129. vsubsws—Subtract Four Signed Integer Elements (32-Bit)

vsububm vsububm

Vector Subtract Unsigned Byte Modulo

vsububm **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	1024
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 8
    vD$_{i:i+7}$ ← (vA)$_{i:i+7}$ +$_{int}$ ¬(vB)$_{i:i+7}$
end
```

Each element of **vsububm** is a byte.

Each integer element in **v**B is subtracted from the corresponding integer element in **v**A. The integer result is placed into the corresponding element of **v**D.

Other registers altered:

- None

Note the **vsububm** instruction can be used for unsigned or signed integers.

Figure 6-130 shows the usage of the **vsububm** instruction. Each of the sixteen elements in the vectors, **v**A, **v**B, and **v**D, is 8 bits long.

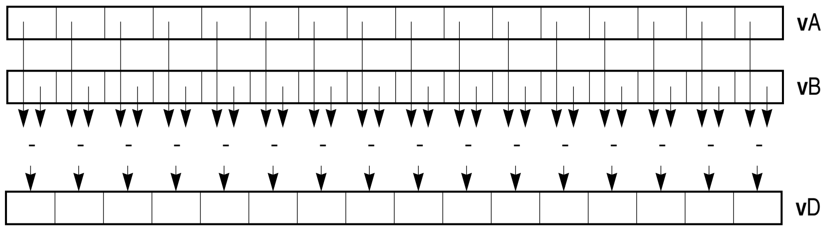


Figure 6-130. vsububm—Subtract Sixteen Integer Elements (8-Bit)

vsububs vsububs

Vector Subtract Unsigned Byte Saturate

vsububs **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	1536
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 8
    aop$_{0:8}$← ZeroExtend((vA)$_{i:i+7}$,9)
    bop$_{0:8}$← ZeroExtend((vB)$_{i:i+7}$,9)
    temp$_{0:8}$← aop$_{0:8}$ +$_{int}$ –bop$_{0:8}$ +$_{int}$ 1
    vD$_{i:i+7}$← SItoUIsat(temp$_{0:8}$,8)
end
```

Each element is a byte. Each unsigned-integer element in **v**B is subtracted from the corresponding unsigned-integer element in **v**A.

If the intermediate result is less than 0 it saturates to 0, where 8 is the length of the element. The unsigned-integer result is placed into the corresponding element of **v**D.

Other registers altered:

- SAT

Figure 6-131 shows the usage of the **vsububs** instruction. Each of the sixteen elements in the vectors, **v**A, **v**B, and **v**D, is 8 bits long.

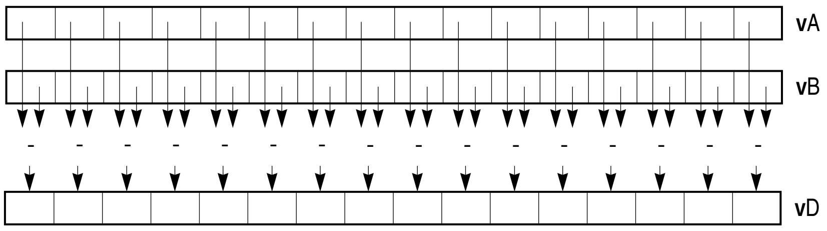


Figure 6-131. vsububs—Subtract Sixteen Unsigned Integer Elements (8-Bit)

vsubuhm vsubuhm

Vector Subtract Signed Half Word Modulo

vsubuhm **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	1088
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 16
    $vD_{i:i+15} \leftarrow (vA)_{i:i+15} +_{int} -(vB)_{i:i+15}$
end
```

Each element is a half word. Each integer element in **v**B is subtracted from the corresponding integer element in **v**A. The integer result is placed into the corresponding element of **v**D.

Other registers altered:

- None

Note the **vsubuhm** instruction can be used for unsigned or signed integers.

Figure 6-132 shows the usage of the **vsubuhm** instruction. Each of the eight elements in the vectors, **v**A, **v**B, and **v**D, is 16 bits long.

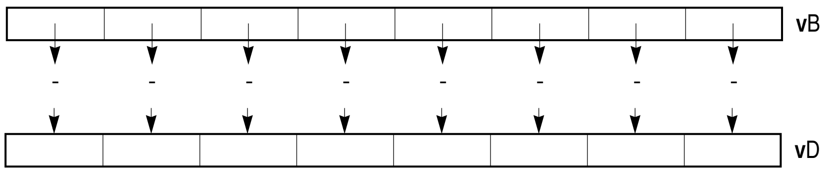


Figure 6-132. vsubuhm—Subtract Eight Integer Elements (16-Bit)

vsubuhs   vsubuhs

Vector Subtract Signed Half Word Saturate

vsubuhs   **v**D,**v**A,**v**B   Form: VX

04	**v**D	**v**A	**v**B	1600
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 16
    aop$_{0:16}$← ZeroExtend((vA)$_{i:i+15}$,17)
    bop$_{0:16}$← ZeroExtend((vB)$_{i:i+n:1}$,17)
    temp$_{0:16}$← aop$_{0:n}$ +$_{int}$ –bop$_{0:16}$ +$_{int}$ 1
    vD$_{i:i+15}$← SItoUIsat(temp$_{0:16}$,16)
end
```

Each element is a half word. Each unsigned-integer element in **v**B is subtracted from the corresponding unsigned-integer element in **v**A.

If the intermediate result is less than 0 it saturates to 0, where 16 is the length of the element. The unsigned-integer result is placed into the corresponding element of **v**D.

Other registers altered:

- SAT

Figure 6-133 shows the usage of the **vsubuhs** instruction. Each of the eight elements in the vectors, **v**A, **v**B, and **v**D, is 16 bits long.

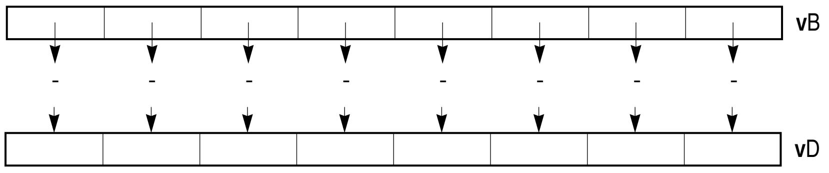


Figure 6-133. vsubuhs—Subtract Eight Signed Integer Elements (16-Bit)

vsubuwm vsubuwm

Vector Subtract Unsigned Word Modulo

vsubuwm **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	1152
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 32
    $vD_{i:i+31} \leftarrow (vA)_{i:i+31} +_{int} -(vB)_{i:i+31}$
end
```

Each element of **vsubuwm** is a word.

Each integer element in **v**B is subtracted from the corresponding integer element in **v**A. The integer result is placed into the corresponding element of **v**D.

Other registers altered:

- None

Note the **vsubuwm** instruction can be used for unsigned or signed integers.

Figure 6-134 shows the usage of the **vsubuwm** instruction. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits long.

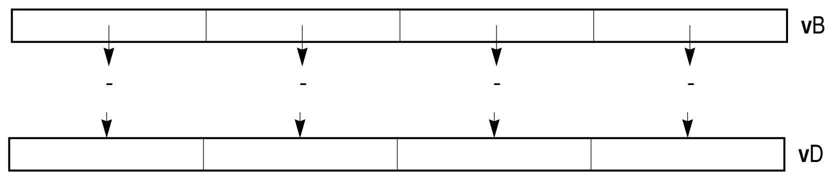


Figure 6-134. vsubuwm—Subtract Four Integer Elements (32-Bit)

vsubuws vsubuws

Vector Subtract Unsigned Word Saturate

vsubuws **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	1664
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 32
    $aop_{0:32}$ ← ZeroExtend(($vA)_{i:i+31}$,33)
    $bop_{0:32}$ ← ZeroExtend(($vB)_{i:i+31}$,33)
    $temp_{0:32}$ ← $aop_{0:32} +_{int} \neg bop_{0:32} +_{int} 1$
    $vD_{i:i+31}$ ← SItoUIsat($temp_{0:32}$,32)
end
```

Each element is a word. Each unsigned-integer element in **v**B is subtracted from the corresponding unsigned-integer element in **v**A.

If the intermediate result is less than 0 it saturates to 0, where 32 is the length of the element. The unsigned-integer result is placed into the corresponding element of **v**D.

Other registers altered:

- SAT

Figure 6-135 shows the usage of the **vsubuws** instruction. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits long.

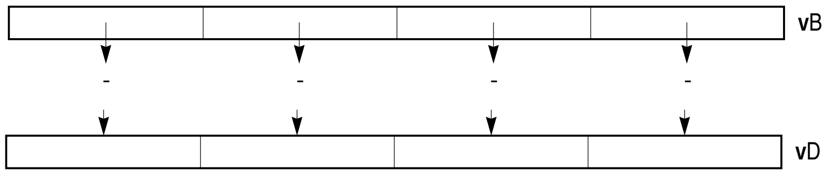


Figure 6-135. vsubuws—Subtract Four Signed Integer Elements (32-Bit)

vsumsws vsumsws

Vector Sum Across Signed Word Saturate

vsumsws **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	1928
0 5	6 10	11 15	16 20	21 31

```
temp$_{0:34}$ ← SignExtend((vB)$_{96:127}$,35)
do i=0 to 127 by 32
    temp$_{0:34}$ ← temp$_{0:34}$ +$_{int}$ SignExtend((vA)$_{i:i+31}$,35)
    vD ← $^{96}$0 || SItoSIsat(temp$_{0:34}$,32)
end
```

The signed-integer sum of the four signed-integer word elements in **v**A is added to the signed-integer word element in bits of **v**B[96-127]. If the intermediate result is greater than (2^{31}-1) it saturates to (2^{31}-1) and if it is less than -2^{31} it saturates to -2^{31}. The signed-integer result is placed into bits **v**D[96–127]. Bits **v**D[0–95] are cleared.

Other registers altered:

- SAT

Figure 6-136 shows the usage of the **vsumsws** instruction. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits long.

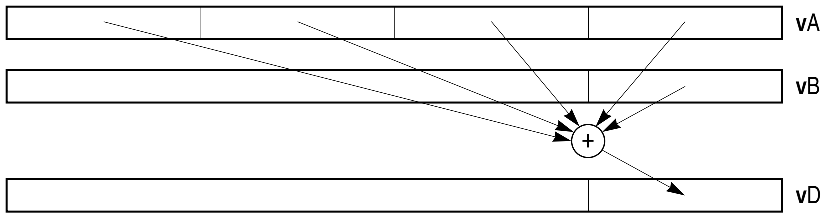


Figure 6-136. vsumsws—Sum Four Signed Integer Elements (32-Bit)

vsum2sws vsum2sws

Vector Sum Across Partial (1/2) Signed Word Saturate

vsum2sws **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	1672
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 64
    $temp_{0:33} \leftarrow$ SignExtend((vB)$_{i+32:i+63}$,34)
    do j=0 to 63 by 32
        $temp_{0:33} \leftarrow temp_{0:33} +_{int}$ SignExtend((vA)$_{i+j:i+j+31}$,34)
    end
    $vD_{i:i+63} \leftarrow {}^{32}0$ || SItoSIsat($temp_{0:33}$,32)
end
```

The signed-integer sum of the first two signed-integer word elements in register **v**A is added to the signed-integer word element in **v**B[32–63]. If the intermediate result is greater than (2^{31}-1) it saturates to (2^{31}-1) and if it is less than -2^{31} it saturates to -2^{31}. The signed-integer result is placed into **v**D[32–63]. The signed-integer sum of the last two signed-integer word elements in register **v**A is added to the signed-integer word element in **v**B[96-127]. If the intermediate result is greater than (2^{31}-1) it saturates to (2^{31}-1) and if it is less than -2^{31} it saturates to -2^{31}. The signed-integer result is placed into **v**D[96–127]. The register **v**D[0–31,64–95] are cleared to 0.

Other registers altered:

- SAT

Figure 6-137 shows the usage of the **vsum2sws** instruction. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits long.

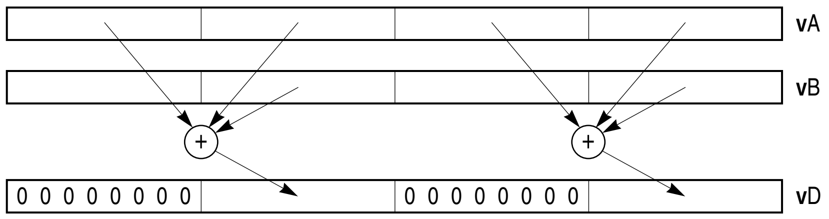


Figure 6-137. vsum2sws—Two Sums in the Four Signed Integer Elements (32-Bit)

vsum4sbs vsum4sbs

Vector Sum Across Partial (1/4) Signed Byte Saturate

vsum4sbs **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	1800
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 32
  temp_{0:32} ← SignExtend((vB)_{i:i+31},33)
  do j=0 to 31 by 8
        temp_{0:32} ← temp_{0:32} +_{int} SignExtend((vA)_{i+j:i+j+7},33)
    end
    vD_{i:i+31} ← SItoSIsat(temp_{0:32},32)
end
```

For each word element in **v**B the following operations are performed in the order shown.

- The signed-integer sum of the four signed-integer byte elements contained in the corresponding word element of register **v**A is added to the signed-integer word element in register **v**B.
- If the intermediate result is greater than (2^{31}-1) it saturates to (2^{31}-1) and if it is less than -2^{31} it saturates to -2^{31}.
- The signed-integer result is placed into the corresponding word element of **v**D.

Other registers altered:

- SAT

Figure 6-138 shows the usage of the **vsum4sbs** instruction. Each of the sixteen elements in the vector **v**A, is 8 bits long. Each of the four elements in the vectors **v**B and **v**D is 32 bits long.

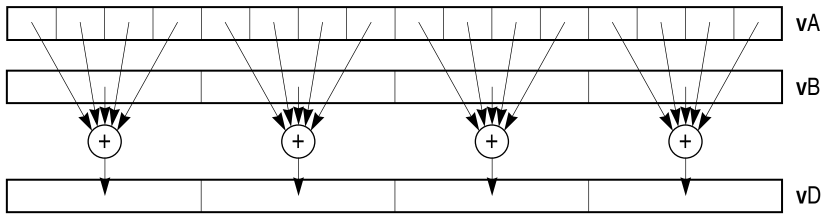


Figure 6-138. vsum4sbs—Four Sums in the Integer Elements (32-Bit)

vsum4shs vsum4shs

Vector Sum Across Partial (1/4) Signed Half Word Saturate

vsum4shs **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	1608
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 32
    $temp_{0:32}$ ← SignExtend((vB)$_{i:i+31}$,33)
    do j=0 to 31 by 16
        $temp_{0:32}$ ← $temp_{0:32}$ $+_{int}$ SignExtend((vA)$_{i+j:i+j+15}$,33)
    end
    vD$_{i:i+31}$ ← SItoSIsat($temp_{0:32}$,32)
end
```

For each word element in register **v**B the following operations are performed, in the order shown.

- The signed-integer sum of the two signed-integer halfword elements contained in the corresponding word element of register **v**A is added to the signed-integer word element in **v**B.
- If the intermediate result is greater than (2^{31}-1) it saturates to (2^{31}-1) and if it is less than -2^{31} it saturates to -2^{31}.
- The signed-integer result is placed into the corresponding word element of **v**D.

Other registers altered:

- SAT

Figure 6-139 shows the usage of the **vsum4shs** instruction. Each of the eight elements in the vector **v**A, is 16 bits long. Each of the four elements in the vectors **v**B and **v**D is 32 bits long.

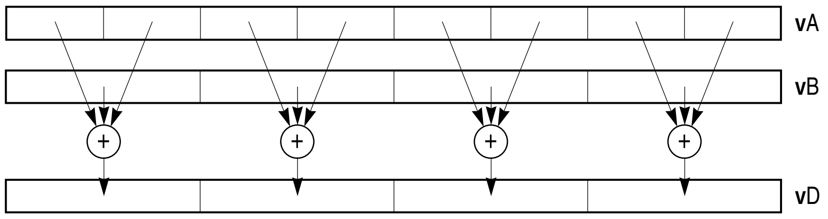


Figure 6-139. vsum4shs—Four Sums in the Integer Elements (32-Bit)

vsum4ubs vsum4ubs

Vector Sum Across Partial (1/4) Unsigned Byte Saturate

vsum4ubs **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	1544
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 32
    $temp_{0:32} \leftarrow$ ZeroExtend((vB)$_{i:i+31}$,33)
    do j=0 to 31 by 8
    $temp_{0:32} \leftarrow temp_{0:32} +_{int}$ ZeroExtend((vA)$_{i+j:i+j+7}$,33)
  end
    vD$_{i:i+31} \leftarrow$ UItoUIsat($temp_{0:32}$,32)
end
```

For each word element in **v**B the following operations are performed in the order shown.

- The unsigned-integer sum of the four unsigned-integer byte elements contained in the corresponding word element of register **v**A is added to the unsigned-integer word element in register **v**B.
- If the intermediate result is greater than (2^{32}-1) it saturates to (2^{32}-1).
- The unsigned-integer result is placed into the corresponding word element of **v**D.

Other registers altered:

- SAT

Figure 6-140 shows the usage of the **vsum4ubs** instruction. Each of the four elements in the vector **v**A, is 8 bits long. Each of the four elements in the vectors **v**B and **v**D is 32 bits long.

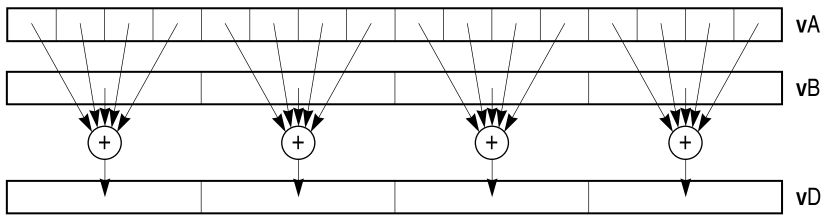


Figure 6-140. vsum4ubs—Four Sums in the Integer Elements (32-Bit)

vupkhpx vupkhpx

Vector Unpack High Pixel16

vupkhpx **v**D,**v**B Form: VX

04	**v**D	0_0000	**v**B	846
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 63 by 16
    vD_{i*2:(i*2)+7} ← SignExtend((vB)_i,8)
    vD_{(i*2)+8:(i*2)+15} ← ZeroExtend((vB)_{i+1:i+5},8)
    vD_{(i*2)+16:(i*2)+23} ← ZeroExtend((vB)_{i+6:i+10},8)
    vD_{(i*2)+24:(i*2)+31} ← ZeroExtend((vB)_{i+11:i+15},8)
end
```

Each halfword element in the high-order half of register **v**B is unpacked to produce a 32-bit value as described below and placed, in the same order, into the four words of **v**D.

A halfword is unpacked to 32 bits by concatenating, in order, the results of the following operations.

- sign-extend bit 0 of the halfword to 8 bits
- zero-extend bits 1–5 of the halfword to 8 bits
- zero-extend bits 6–10 of the halfword to 8 bits
- zero-extend bits 11–15 of the halfword to 8 bits

Other registers altered:

- None

The source and target elements can be considered to be 16-bit and 32-bit "pixels" respectively, having the formats described in the programming note for the Vector Pack Pixel instruction.

Figure 6-141 shows the usage of the **vupkhpx** instruction. Each of the eight elements in the vectors, **v**B, is 16 bits long. Each of the four elements in the vectors, **v**D, is 32 bits long.

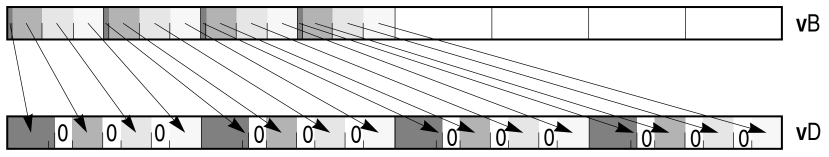


Figure 6-141. vupkhpx—Unpack High-Order Elements (16 bit) to Elements (32-Bit)

vupkhsb vupkhsb

Vector Unpack High Signed Byte

vupkhsb **v**D,**v**B Form: VX

04	**v**D	0_0000	**v**B	526
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 63 by 8
    vD_{i*2:(i*2)+15} ← SignExtend((vB)_{i:i+7},16)
end
```

Each signed integer byte element in the high-order half of register **v**B is sign-extended to produce a 16-bit signed integer and placed, in the same order, into the eight halfwords of register **v**D.

Other registers altered:

- None

Figure 6-142 shows the usage of the **vupkhsb** instruction. Each of the sixteen elements in the vectors, **v**B, is 8 bits long. Each of the eight elements in the vectors, **v**D, is 16 bits long.

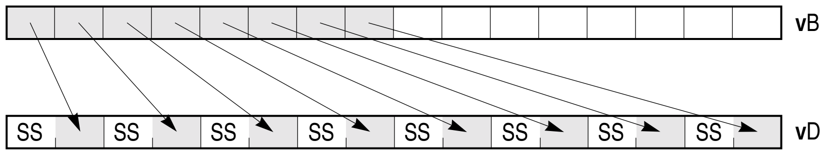


Figure 6-142. vupkhsb—Unpack HIgh-Order Signed Integer Elements (8-Bit) to Signed Integer Elements (16-Bit)

vupkhsh vupkhsh

Vector Unpack High Signed Half Word

vupkhsh **v**D,**v**B Form: VX

04	**v**D	0_0000	**v**B	590
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 63 by 16
    $\mathbf{v}D_{i*2:(i*2)+31} \leftarrow$ SignExtend(($\mathbf{v}B)_{i:i+15}$,32)
end
```

Each signed integer halfword element in the high-order half of register **v**B is sign-extended to produce a 32-bit signed integer and placed, in the same order, into the four words of register **v**D.

Other registers altered:

- None

Figure 6-143 shows the usage of the **vupkhsh** instruction. Each of the eight elements in the vectors **v**B and **v**D is 16 bits long.

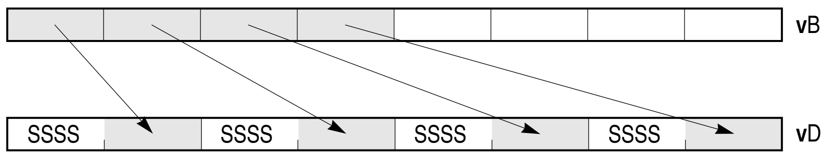


Figure 6-143. vupkhsh—Unpack Signed Integer Elements (16-Bit) to Signed Integer Elements (32-Bit)

vupklpx vupklpx

Vector Unpack Low Pixel16

vupklpx **v**D,**v**B Form: VX

04	**v**D	0_0000	**v**B	974
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 63 by 16
    $\mathbf{v}D_{i*2:(i*2)+7}$← SignExtend(($\mathbf{v}B$)$_{i+64}$,8)
    $\mathbf{v}D_{(i*2)+8:(i*2)+15}$← ZeroExtend(($\mathbf{v}B$)$_{i+65:i+69}$,8)
    $\mathbf{v}D_{(i*2)+16:(i*2)+23}$← ZeroExtend(($\mathbf{v}B$)$_{i+70:i+74}$,8)
    $\mathbf{v}D_{(i*2)+24:(i*2)+31}$← ZeroExtend(($\mathbf{v}B$)$_{i+75:i+79}$,8)
end
```

Each halfword element in the low-order half of register **v**B is unpacked to produce a 32-bit value as described below and placed, in the same order, into the four words of register **v**D.

A halfword is unpacked to 32 bits by concatenating, in order, the results of the following operations.

- sign-extend bit 0 of the halfword to 8 bits
- zero-extend bits 1–5 of the halfword to 8 bits
- zero-extend bits 6–10 of the halfword to 8 bits
- zero-extend bits 11–15 of the halfword to 8 bits

Other registers altered:

- None

Programming note: Notice that the unpacking done by the Vector Unpack Pixel instructions does not reverse the packing done by the Vector Pack Pixel instruction. Specifically, if a 16-bit pixel is unpacked to a 32-bit pixel which is then packed to a 16-bit pixel, the resulting 16-bit pixel will not, in general, be equal to the original 16-bit pixel (because, for each channel except the first, Vector Unpack Pixel inserts high-order bits while Vector Pack Pixel discards low-order bits).

Figure 6-144 shows the usage of the **vupklpx** instruction. Each of the eight elements in the vectors, **v**B, is 16 bits long. Each of the four elements in the vectors, **v**D, is 32 bits long.

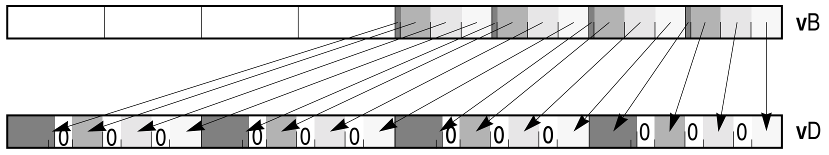


Figure 6-144. vupklpx—Unpack Low-order Elements (16-Bit) to Elements (32-Bit)

vupklsb vupklsb

Vector Unpack Low Signed Byte

vupklsb **v**D,**v**B Form: VX

04	**v**D	0_0000	**v**B	654
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 63 by 8
    vD_{i*2:(i*2)+15} ← SignExtend((vB)_{i+64:i+71},16)
end
```

Each signed integer byte element in the low-order half of register **v**B is sign-extended to produce a 16-bit signed integer and placed, in the same order, into the eight halfwords of register **v**D.

Other registers altered:

- None

Figure 6-145 shows the usage of the **vaddubs** instruction. Each of the sixteen elements in the vectors **v**B and **v**D is 8 bits long.

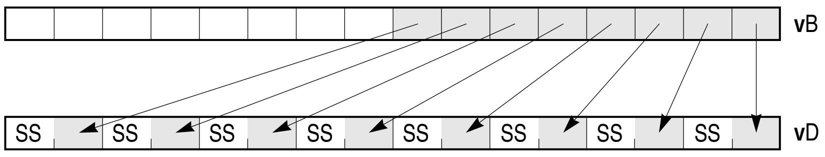


Figure 6-145. vupklsb—Unpack Low-Order Elements (8-Bit) to Elements (16-Bit)

vupklsh vupklsh

Vector Unpack Low Signed Half Word

vupklsh **v**D,**v**B Form: VX

04	**v**D	0_0000	**v**B	718
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 63 by 16
    $\mathbf{v}D_{i*2:(i*2)+31} \leftarrow$ SignExtend(($\mathbf{v}B)_{i+64:i+79}$,32)
end
```

Each signed integer half word element in the low-order half of register **v**B is sign-extended to produce a 32-bit signed integer and placed, in the same order, into the four words of register **v**D.

Other registers altered:

- None

Figure 6-146 shows the usage of the **vupklpx** instruction. Each of the eight elements in the vectors, **v**A, **v**B, and **v**D, is 16 bits long.

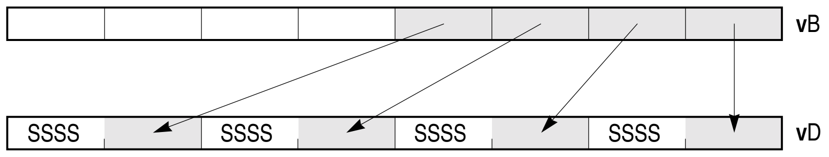


Figure 6-146. vupklsh—Unpack Low-Order Signed Integer Elements (16-Bit) to Signed Integer Elements (32-Bit)

vxor vxor

Vector Logical XOR

vxor **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	1220
0 5	6 10	11 15	16 20	21 31

```
vD ← (vA) ⊕ (vB)
```

The contents of **v**A are XORed with the contents of register **v**B and the result is placed into register **v**D.

Other registers altered:

- None

Figure 6-147 shows the usage of the **vxor** instruction.

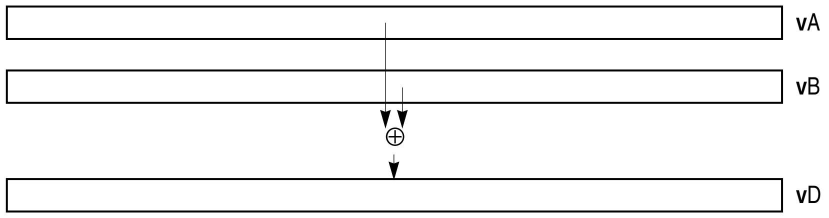


Figure 6-147. vxor—Bitwise XOR (128-Bit)