Godson MultiMedia Technology

1.1 OVERVIEW

The media extensions for the Godson Architecture were designed to enhance performance of advanced media and communication applications. The Godson MultiMedia technology provides a new level of performance to computer platforms by adding new instructions and defining new 64-bit data types, while preserving compatibility with software and operating systems developed for the Godson Architecture. The Godson MultiMedia technology introduces new general-purpose instructions. These instructions operate in parallel on multiple data elements packed into 64-bit quantities. They perform arithmetic and logical operations on the different data types. These instructions accelerate the performance of applications with compute-intensive algorithms that perform localized, recurring operations on small native data. This includes applications such as motion video, combined graphics with video, image processing, audio synthesis, speech synthesis and compression, telephony, video conferencing, 2D graphics, and 3D graphics.

The Godson MultiMedia instruction set has a simple and flexible software model with no new mode or operating-system visible state. The Godson MultiMedia instruction set is fully compatible with all Godson Architecture microprocessors. All existing software continues to run correctly, without modification, on microprocessors that incorporate the Godson MultiMedia technology, as well as in the presence of existing and new applications that incorporate this technology.

The Godson MultiMedia technology uses the Single Instruction, Multiple Data (SIMD) technique. This technique speeds up software performance by processing multiple data elements in parallel, using a single instruction. The Godson MultiMedia technology supports parallel operations on byte, halfword, and word data elements, and doubleword integer data type.

Modern media, communications, and graphics applications now include sophisticated algorithms that perform recurring operations on small data types. The Godson MultiMedia technology directly addresses the need of these applications. For example, most audio data is represented in 16-bit (halfword) quantities. The Godson MultiMedia instructions can operate on four of these words simultaneously with one instruction. Video and graphics information is commonly represented as palletized 8-bit (byte) quantities; one Godson MultiMedia instruction can operate on eight of these bytes simultaneously.

1.2 INSTRUCTION SYNTAX

Instructions vary by:

- Data type: packed bytes, packed halfwords, packed words or doublewords
- Signed - Unsigned numbers
- Wraparound - Saturate arithmetic

A typical Godson MultiMedia instruction has this syntax:

- Prefix: **P** for Packed
- Instruction operation: for example - ADD, CMP, or XOR
- Suffix:

  --**US** for Unsigned Saturation
  --**S** for Signed saturation
  --**B, H, W, D** for the data type: packed byte, packed halfword, packed word, or doubleword.

Instructions that have different input and output data elements have two data-type suffixes. For example, the conversion instruction converts from one data type to another. It has two suffixes: one for the original data type and the second for the converted data type.
This is an example of an instruction mnemonic syntax :

PADDUSW (Packed Add Unsigned with Saturation for Word)
P = Packed
ADD = the instruction operation
US = Unsigned Saturation
W = Word

1.3 SATURATION AND WRAPAROUND MODES

When performing integer arithmetic, an operation may result in an out-of-range condition, where the true result cannot be represented in the destination format. For example, when performing arithmetic on signed halfword integers, positive overflow can occur causing the true signed result is larger than 16 bits.

The Godson MultiMedia technology provides three ways of handling out-of-range conditions:

- Wraparound arithmetic.
- Signed saturation arithmetic.
- Unsigned saturation arithmetic.

With wraparound arithmetic, a true out-of-range result is truncated (that is, the carry or overflow bit is ignored and only the least significant bits of the result are returned to the destination). Wraparound arithmetic is suitable for applications that control the range of operands to prevent out-of-range results. If the range of operands is not controlled, however, wraparound arithmetic can lead to large errors. For example, adding two large signed numbers can cause positive overflow and produce a negative result.

With signed saturation arithmetic, out-of-range results are limited to the representable range of signed integers for the integer size being operated on. For example, if positive overflow occurs when operating on signed halfword integers, the result is "saturated" to 7FFFH, which is the largest positive integer that can be represented in 16 bits; if negative overflow occurs, the result is saturated to 8000H.

With unsigned saturation arithmetic, out-of-range results are limited to the representable range of unsigned integers for the integer size being operated on. So, positive overflow when operating on unsigned byte integers results in FFH being returned and negative overflow results in 00H being retuned.

Saturation arithmetic provides a more natural answer for many overflow situations. For example, in color calculations, saturation causes a color to remain pure black or pure white without allowing inversion. It also prevents wraparound artifacts from entering into computations, when range checking of source operands it not used.

Godson MultiMedia instructions do not indicate overflow or underflow occurrence by generating exceptions.

1.4 GODSON MULTIMEDIA INSTRUCTIONS

The Godson MultiMedia Technology defines 65 instructions(see Table 1-1). The instructions are grouped into the following functional categories:

- Arithmetic Instructions
- Comparison Instructions
- Conversion Instructions
- Logical Instructions
- Shift Instructions

Table 1-1 Godson MultiMedia Instruction Set Summary

OP / Fmt	ADD	SUB	MUL	DIV	ABS
13	Or	PASUBUB	Dsll	Dsrl	
14			PEXTRH		
15			PMADDHW	Dsra	

16					
17					
18	PAVGH	PCMPEQW	PSLLW	PSRLW	
19	PAVGB	PCMPGTW	PSLLH	PSRLH	
20	PMAXSH	PCMPEQH	PMULLH	PSRAW	BIADD
21	PMINSH	PCMPGTH	PMULHH	PSRAH	PMOVMASKB
22	PMAXUB	PCMPEQB	PMULUW	PUNPCKLWD	
23	PMINUB	PCMPGTB	PMULHUH	PUNPCKHWD	
24	PADDSH	PSUBSH	PSHUFH	PUNPCKLHW	
25	PADDUSH	PSUBUSH	PACKSSWH	PUNPCKHHW	
26	PADDH	PSUBH	PACKSSHB	PUNPCKLBH	
27	PADDW	PSUBW	PACKUSHB	PUNPCKHBH	
28	PADDSB	PSUBSB	Xor	PINSRH_0	
29	PADDUSB	PSUBUSB	Nor	PINSRH_1	
30	PADDB	PSUBB	And	PINSRH_2	
31	PADDD	PSUBD	PANDN	PINSRH_3	

PACKSSHB/PACKSSWH—Pack with Signed Saturation

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	PACKSSHB 11010	ft	fs	fd	MUL 000010
6	5	5	5	5	6

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	PACKSSWH 11001	ft	fs	fd	MUL 000010
6	5	5	5	5	6

Format:

PACKSSHB	fd,fs,ft
PACKSSWH	fd,fs,ft

Description:

Converts packed signed halfword integers into packed signed byte integers (PACKSSHB) or converts packed signed word integers into packed signed halfword integers (PACKSSWH), using saturation to handle overflow conditions. See Figure 3-5 for an example of the packing operation.

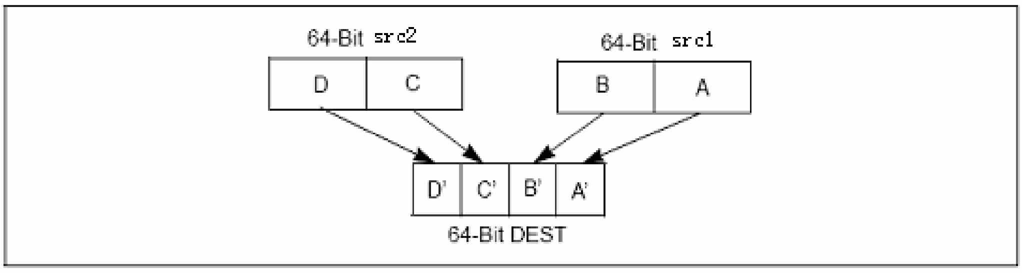

Figure 3-5. Operation of the PACKSSWH Instruction Using 64-bit Operands.

The PACKSSHB instruction converts 4 signed halfword integers from the first operand and 4 signed halfword integers from the second operand into 8 signed byte integers and stores the result in the destination operand. If a signed halfword integer value is beyond the range of a signed byte integer (that is, greater than 7FH for a positive integer or greater than 80H for a negative integer), the saturated signed byte integer value of 7FH or 80H, respectively, is stored in the destination.

The PACKSSWH instruction packs 2 signed words from the first operand and 2 signed words from the second operand into 4 signed halfwords in the destination operand (see Figure 3-5). If a signed word integer value is beyond the range of a signed halfword (that is, greater than 7FFFH for a positive integer or greater than 8000H for a negative integer), the saturated signed halfword integer value of 7FFFH or 8000H, respectively, is stored into the destination.

The PACKSSHB and PACKSSWH instructions operate on 64-bit operands.

Operation:

```
PACKSSHB
    fd[7..0]    ← SaturateSignedHalfwordToSignedByte fs[15..0];
    fd[15..8]   ← SaturateSignedHalfwordToSignedByte fs[31..16];
    fd[23..16]  ← SaturateSignedHalfwordToSignedByte fs[47..32];
    fd[31..24]  ← SaturateSignedHalfwordToSignedByte fs[63..48];
    fd[39..32]  ← SaturateSignedHalfwordToSignedByte ft[15..0];
    fd[47..40]  ← SaturateSignedHalfwordToSignedByte ft[31..16];
    fd[55..48]  ← SaturateSignedHalfwordToSignedByte ft[47..32];
    fd[63..56]  ← SaturateSignedHalfwordToSignedByte ft[63..48];

PACKSSWH
    fd[15..0]   ← SaturateSignedWordToSignedHalfWord fs[31..0];
    fd[31..16]  ← SaturateSignedWordToSignedHalfWord fs[63..32];
    fd[47..32]  ← SaturateSignedWordToSignedHalfWord ft[31..0];
    fd[63..48]  ← SaturateSignedWordToSignedHalfWord ft[63..32];
```

Exceptions:

None.

PACKUSHB—Pack with Unsigned Saturation

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	PACKUSHB 11011	ft	fs	fd	MUL 000010
6	5	5	5	5	6

Format:

PACKUSHB fd,fs,ft

Description:

Converts 4 signed halfword integers from the first operand and 4 signed halfword integers from the second operand into 8 unsigned byte integers and stores the result in the destination operand. (See Figure 3-5 for an example of the packing operation.) If a signed halfword integer value is beyond the range of an unsigned byte integer (that is, greater than FFH or less than 00H), the saturated unsigned byte integer value of FFH or 00H, respectively, is stored in the destination.
The PACKUSHB instruction operates on 64-bit operands.

Operation:

```
PACKUSHB
    fd[7..0]    ← SaturateSignedHalfwordToUnsignedByte fs[15..0];
    fd[15..8]   ← SaturateSignedHalfwordToUnsignedByte fs [31..16];
    fd[23..16]  ← SaturateSignedHalfwordToUnsignedByte fs [47..32];
    fd[31..24]  ← SaturateSignedHalfwordToUnsignedByte fs [63..48];
    fd[39..32]  ← SaturateSignedHalfwordToUnsignedByte ft[15..0];
    fd[47..40]  ← SaturateSignedHalfwordToUnsignedByte ft[31..16];
    fd[55..48]  ← SaturateSignedHalfwordToUnsignedByte ft[47..32];
    fd[63..56]  ← SaturateSignedHalfwordToUnsignedByte ft[63..48];
```

Exceptions:

None.

PADDB/PADDH/PADDW—Add Packed Integers

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	PADDB 11110	ft	fs	fd	ADD 000000
6	5	5	5	5	6

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	PADDH 11010	ft	fs	fd	ADD 000000
6	5	5	5	5	6

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	PADDW 11011	ft	fs	fd	ADD 000000
6	5	5	5	5	6

Format:

```
PADDB       fd,fs,ft
PADDH       fd,fs,ft
PADDW       fd,fs,ft
```

Description:

Performs a SIMD add of the packed integers from the first operand and the second operand, and stores the packed integer results in the destination operand. Overflow is handled with wraparound, as described in the following paragraphs.

These instructions operate on 64-bit operands.

The PADDB instruction adds packed byte integers. When an individual result is too large to be represented in 8 bits (overflow), the result is wrapped around and the low 8 bits are written to the destination operand (that is, the carry is ignored).

The PADDH instruction adds packed halfword integers. When an individual result is too large to be represented in 16 bits (overflow), the result is wrapped around and the low 16 bits are written to the destination operand.

The PADDW instruction adds packed word integers. When an individual result is too large to be represented in 32 bits (overflow), the result is wrapped around and the low 32 bits are written to the destination operand.

Note that the PADDB, PADDH, and PADDW instructions can operate on either unsigned or signed (two's complement notation) packed integers; however, it does not indicate overflow and/or a carry. To prevent undetected overflow conditions, software must control the ranges of values operated on.

Operation:

```
PADDB
    fd[7..0]    ←  fs[7..0] + ft[7..0];
    * repeat add operation for 2nd through 7th byte *;
    fd[63..56]  ←  fs[63..56] + ft[63..56];
PADDH
    fd[15..0]   ←  fs[15..0] + ft[15..0];
    * repeat add operation for 2nd and 3th halfword *;
    fd[63..48]  ←  fs[63..48] + ft[63..48];
PADDW
    fd[31..0]   ←  fs[31..0] + ft[31..0];
    fd[63..32]  ←  fs[63..32] + ft[63..32];
```

Exceptions:

None.

PADDD—Add Packed Doubleword Integers

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	PADDD 11111	ft	fs	fd	ADD 000000
6	5	5	5	5	6

Format:

PADDD fd,fs,ft

Description:

Adds the first operand to the second operand and stores the result in the destination operand. The source operand can be a doubleword integer stored in a 64-bit register. The destination operand can be a doubleword integer stored in a 64-bit register. When a doubleword result is too large to be represented in 64 bits (overflow), the result is wrapped around and the low 64 bits are written to the destination element (that is, the carry is ignored).

Note that the PADDD instruction can operate on either unsigned or signed (two’s complement notation) integers; however, it does not indicate overflow and/or a carry. To prevent undetected overflow conditions, software must control the ranges of the values operated on.

Operation:

```
PADDD
    fd[63..0] ← fs[63..0] + ft[63..0];
```

Exceptions:

None.

PADDSB/PADDSH—Add Packed Signed Integers with Signed Saturation

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	PADDSB 11100	ft	fs	fd	ADD 000000
6	5	5	5	5	6

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	PADDSH 11000	ft	fs	fd	ADD 000000
6	5	5	5	5	6

Format:

PADDSB	fd,fs,ft
PADDSH	fd,fs,ft

Description:

Performs a SIMD add of the packed signed integers from the first operand and the second operand, and stores the packed integer results in the destination operand. Overflow is handled with signed saturation, as described in the following paragraphs.
These instructions operate on 64-bit operands.
The PADDSB instruction adds packed signed byte integers. When an individual byte result is beyond the range of a signed byte integer (that is, greater than 7FH or less than 80H), the saturated value of 7FH or 80H, respectively, is written to the destination operand.
The PADDSH instruction adds packed signed halfword integers. When an individual halfword result is beyond the range of a signed halfword integer (that is, greater than 7FFFH or less than 8000H), the saturated value of 7FFFH or 8000H, respectively, is written to the destination operand.

Operations:

```
PADDSB
    fd[7..0]    ← SaturateToSignedByte(fs[7..0] + ft[7..0]) ;
    * repeat add operation for 2nd through 7th bytes *;
    fd[63..56]  ← SaturateToSignedByte(fs[63..56] + ft[63..56] );
PADDSH
    fd[15..0]   ←  SaturateToSignedHalfword(fs[15..0] + ft[15..0] );
    * repeat add operation for 2nd and 7th halfwords *;
    fd[63..48]  ←  SaturateToSignedHalfword(fs[63..48] + ft[63..48] );
```

Exceptions:

None.

PADDUSB/PADDUSH—Add Packed Unsigned Integers with Unsigned Saturation

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	PADDUSB 11101	ft	fs	fd	ADD 000000
6	5	5	5	5	6

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	PADDUSH 11001	ft	fs	fd	ADD 000000
6	5	5	5	5	6

Format:

PADDUSB	fd,fs,ft
PADDUSH	fd,fs,ft

Description:

Performs a SIMD add of the packed unsigned integers from the first operand and the second operand, and stores the packed integer results in the destination operand. Overflow is handled with unsigned saturation, as described in the following paragraphs.
These instructions operate on 64-bit operands.
The PADDUSB instruction adds packed unsigned byte integers. When an individual byte result is beyond the range of an unsigned byte integer (that is, greater than FFH), the saturated value of FFH is written to the destination operand.
The PADDUSH instruction adds packed unsigned halfword integers. When an individual halfword result is beyond the range of an unsigned halfword integer (that is, greater than FFFFH), the saturated value of FFFFH is written to the destination operand.

Operation:

PADDUSB

```
    fd[7..0]      ← SaturateToUnsignedByte(fs[7..0] + ft[7..0]) ;
    * repeat add operation for 2nd through 7th bytes *;
    fd[63..56]  ← SaturateToUnsignedByte(fs[63..56] + ft[63..56] );
PADDUSH
    fd[15..0]    ←  SaturateToUnsignedHalfword(fs[15..0] + ft[15..0] );
    * repeat add operation for 2nd and 3rd halfwords *;
    fd[63..48]  ←  SaturateToUnsignedHalfword(fs[63..48] + ft[63..48] );
```

Exceptions:

None.

PANDN—Logical AND NOT

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	PANDN 11111	ft	fs	fd	MUL 000010
6	5	5	5	5	6

Format:

PANDN fd,fs,ft

Description:

Performs a bitwise logical NOT of the first operand, then performs a bitwise logical AND of the second operand and the inverted destination operand. The result is stored in the destination operand. The source operand can be a 64-bit register. The destination operand can be a 64-bit register. Each bit of the result is set to 1 if the corresponding bit in the first operand is 0 and the corresponding bit in the second operand is 1; otherwise, it is set to 0.

Operation:

```
PANDN
    fd ← (NOT fs) AND ft;
```

Exceptions:

None.

PAVGB/PAVGH—Average Packed Integers

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	PAVGB 10011	ft	fs	fd	ADD 000000
6	5	5	5	5	6

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	PAVGH 10010	ft	fs	fd	ADD 000000
6	5	5	5	5	6

Format:

```
PAVGB        fd,fs,ft
PAVGH        fd,fs,ft
```

Description:

Performs a SIMD average of the packed unsigned integers from the first operand and the second operand, and stores the results in the destination operand. For each corresponding pair of data elements in the first and second operands, the elements are added together, a 1 is added to the temporary sum, and that result is shifted right one bit position. The source operand can be a 64-bit register. The destination operand can be a 64-bit register.
The PAVGB instruction operates on packed unsigned bytes and the PAVGH instruction operates on packed unsigned halfwords.

Operation:

```
PAVGB
    ft[7-0]     ← (fs[7..0] + ft[7..0] + 1) >> 1; * temp sum before shifting is 9 bits *
    * repeat operation performed for bytes 2 through 6 *;
    ft[63-56]  ← (fs[63..56] + ft[63..56] + 1) >> 1;
PAVGH
    ft[15-0]    ← (fs[15..0] + ft[15..0] + 1) >> 1; * temp sum before shifting is 17 bits *
```

```
* repeat operation performed for halfwords 2 and 3 *;
ft[63-48]  ← (fs[63..48] + ft[63..48] + 1) >> 1;
```

Exceptions:

None.

PCMPEQB/PCMPEQH/PCMPEQW— Compare Packed Data for Equal

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	PCMPEQB 10110	ft	fs	fd	SUB 000001
6	5	5	5	5	6

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	PCMPEQH 10100	ft	fs	fd	SUB 000001
6	5	5	5	5	6

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	PCMPEQW 10010	ft	fs	fd	SUB 000001
6	5	5	5	5	6

Format:

PCMPEQB	fd,fs,ft
PCMPEQH	fd,fs,ft
PCMPEQW	fd,fs,ft

Description:

Performs a SIMD compare for equality of the packed bytes, halfwords, or words in the first operand and the second operand. If a pair of data elements is equal, the corresponding data element in the destination operand is set to all 1s; otherwise, it is set to all 0s. The source operand can be a 64-bit register The destination operand can be a 64-bit register.
The PCMPEQB instruction compares the corresponding bytes in the first and second operands; the PCMPEQH instruction compares the corresponding halfwords in the first and second operands; and the PCMPEQW instruction compares the corresponding words in the first and second operands.

Operation:

```
PCMPEQB
    IF fs[7..0] = ft[7..0]
        THEN fd[7..0] ← FFH;
        ELSE fd[7..0] ← 0;
    * Continue comparison of 2nd through 7th bytes in fs and ft *
    IF fs[63..56] = ft[63..56]
        THEN fd[63..56] ← FFH;
        ELSE fd[63..56] ← 0;
PCMPEQH
    IF fs[15..0] = ft[15..0]
        THEN fd[15..0] ← FFFFH;
        ELSE fd[15..0] ← 0;
    * Continue comparison of 2nd and 3rd halfwords in fs and ft *
    IF fs[63..48] = ft[63..48]
        THEN fd[63..48] ← FFFFH;
        ELSE fd[63..48] ← 0;
PCMPEQW
    IF fs[31..0] = ft[31..0]
        THEN fd[31..0] ← FFFFFFFFH;
        ELSE fd[31..0] ← 0;
    IF fs[63..32] = ft[63..32]
        THEN fd[63..32] ← FFFFFFFFH;
        ELSE fd[63..32] ← 0;
```

Exceptions:

None.

PCMPGTB/PCMPGTH/PCMPGTW—Compare Packed Signed Integers for Greater Than

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	PCMPGTB 10111	ft	fs	fd	SUB 000001
6	5	5	5	5	6

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	PCMPGTH 10101	ft	fs	fd	SUB 000001
6	5	5	5	5	6

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	PCMPGTW 10011	ft	fs	fd	SUB 000001
6	5	5	5	5	6

Format:

```
PCMPGTB         fd,fs,ft
PCMPGTH         fd,fs,ft
PCMPGTW         fd,fs,ft
```

Description:

Performs a SIMD signed compare for the greater value of the packed byte, halfword, or word integers in the first operand and the second operand. If a data element in the first operand is greater than the corresponding date element in the second operand, the corresponding data element in the destination operand is set to all 1s; otherwise, it is set to all 0s. The source operand can be a 64-bit register. The destination operand can be a 64-bit register.

The PCMPGTB instruction compares the corresponding signed byte integers in the first and second operands; the PCMPGTH instruction compares the corresponding signed halfword integers in the first and second operands; and the PCMPGTW instruction compares the corresponding signed word integers in the first and second operands.

Operation:

```
PCMPGTB
    IF fs[7..0] > ft[7..0]
        THEN fd[7 0]  ← FFH;
        ELSE fd[7..0]  ← 0;
    * Continue comparison of 2nd through 7th bytes in fs and ft *
    IF fs[63..56] > ft[63..56]
        THEN fd[63..56]  ← FFH;
        ELSE fd[63..56]  ← 0;
PCMPGTH
    IF fs[15..0] > ft[15..0]
        THEN fd[15..0]  ← FFFFH;
        ELSE fd[15..0]  ← 0;
    * Continue comparison of 2nd and 3rd halfwords in fs and ft *
    IF fs[63..48] > ft[63..48]
        THEN fd[63..48]  ←  FFFFH;
        ELSE fd[63..48]  ← 0;
PCMPGTW
    IF fs[31..0] > ft[31..0]
        THEN fd[31..0]  ← FFFFFFFFH;
        ELSE fd[31..0]  ← 0;
    IF fs[63..32] > ft[63..32]
        THEN fd[63..32]  ← FFFFFFFFH;
        ELSE fd[63..32]  ← 0;
```

Exceptions:

None.

PEXTRH—Extract Halfword

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	PEXTRH 01110	ft	fs	fd	MUL 000010
6	5	5	5	5	6

Format:

PEXTRH fd,fs,ft

Description:

Copies the halfword in the first operand specified by the second operand to the destination operand. The high halfword of the destination operand is cleared (set to all 0s).

Operation:

```
PEXTRH
    SEL         ← ft AND 3H;
    TEMP        ← (fs >> (SEL * 16)) AND FFFFH;
    fd[15..0]   ← TEMP[15..0];
    fd[63..16]  ← 00000000H;
```

Exceptions:

None.

PINSRH—Insert Halfword

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	PINSRH_0 11100	ft	fs	fd	DIV 000011
6	5	5	5	5	6

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	PINSRH_1 11101	ft	fs	fd	DIV 000011
6	5	5	5	5	6

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	PINSRH_2 11110	ft	fs	fd	DIV 000011
6	5	5	5	5	6

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	PINSRH_3 11111	ft	fs	fd	DIV 000011
6	5	5	5	5	6

Format:

PINSRH_0	fd,fs,ft
PINSRH_1	fd,fs,ft
PINSRH_2	fd,fs,ft
PINSRH_3	fd,fs,ft

Description:

Copies a halfword from the second operand and inserts it in the first operand at the location specified with the number of the instruction name. (The other halfwords in the first register are left untouched.)

Operation:

```
PINSRH_0
    MASK $\leftarrow$ 000000000000FFFFH;
    fd $\leftarrow$ (fs AND NOT MASK) OR (((ft << (0 $*$ 16)) AND MASK);
PINSRH_1
    MASK $\leftarrow$ 00000000FFFF0000H;
    fd $\leftarrow$ (fs AND NOT MASK) OR (((ft << (1 $*$ 16)) AND MASK);
PINSRH_2
    MASK $\leftarrow$ 0000FFFF00000000H;
    fd $\leftarrow$ (fs AND NOT MASK) OR (((ft << (2 $*$ 16)) AND MASK);
PINSRH_3
    MASK $\leftarrow$ FFFF000000000000H;
    fd $\leftarrow$ (fs AND NOT MASK) OR (((ft << (3 $*$ 16)) AND MASK);
```

Exceptions:

None.

PMADDHW—Multiply and Add Packed Integers

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	PMADDHW 01111	ft	fs	fd	MUL 000010
6	5	5	5	5	6

Format:

PMADDHW fd,fs,ft

Description:

Multiplies the individual signed halfwords of the first operand by the corresponding signed halfwords of the second operand, producing temporary signed, word results. The adjacent word results are then summed and stored in the destination operand. For example, the corresponding low-order halfwords (15-0) and (31-16) in the first and second operands are multiplied by one another and the word results are added together and stored in the low word of the destination register (31-0). The same operation is performed on the other pairs of adjacent halfwords. (Figure 3-6 shows this operation when using 64-bit operands.) The source operands can be a 64-bit register. The destination operand can be a 64-bit register. The PMADDHW instruction wraps around only in one situation: when the 2 pairs of halfwords being operated on in a group are all 8000H. In this case, the result wraps around to 80000000H.

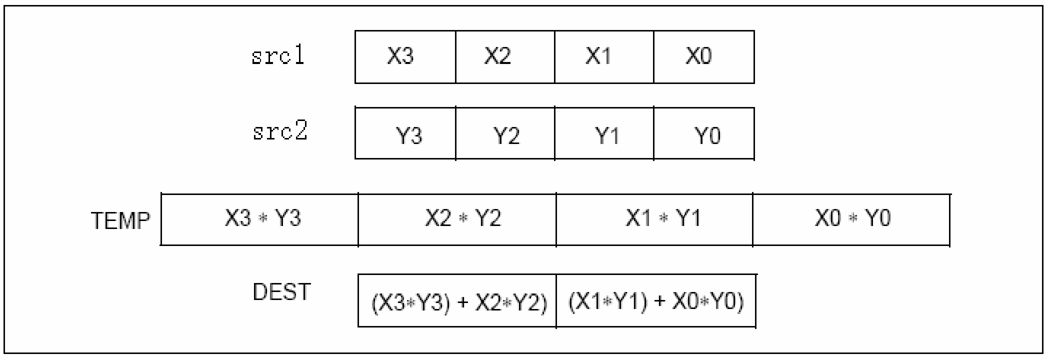

Figure 3-6. PMADDHW Execution Model Using 64-bit Operands

Operation:

```
PMADDHW
    fd[31..0]   ← (fs[15..0] * ft[15..0]) + (fs[31..16] * ft[31..16]);
    fd[63..32]  ← (fs[47..32] * ft[47..32]) + (fs[63..48] * ft[63..48]);
```

Exceptions:

None.

PMAXSH—Maximum of Packed Signed Halfword Integers

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	PMAXSH 10100	ft	fs	fd	ADD 000000
6	5	5	5	5	6

Format:

PMAXSH fd,fs,ft

Description:

Performs a SIMD compare of the packed signed halfword integers in the first operand and the second operand, and returns the maximum value for each pair of halfword integers to the destination operand. The source operands can be a 64-bi register. The destination operand can be a 64-bi register.

Operation:

```
PMAXSH
    IF (fs[15..0] > ft[15..0]) THEN
        fd[15..0] ← fs[15..0];
    ELSE
        fd[15..0] ← ft[15..0];
    FI
    * repeat operation for 2nd and 3rd halfwords in first and second operands *
    IF (fs[63..48] > ft[63..48]) THEN
        fd[63..48] ← fs[63..48];
    ELSE
        fd[63..48] ← ft[63..48];
    FI
```

Exceptions:

None.

PMAXUB—Maximum of Packed Unsigned Byte Integers

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	PMAXUB 10110	ft	fs	fd	ADD 000000
6	5	5	5	5	6

Format:

PMAXUB fd,fs,ft

Description:

Performs a SIMD compare of the packed unsigned byte integers in the first operand and the second operand, and returns the maximum value for each pair of byte integers to the destination operand. The source operands can be a 64-bit register. The destination operand can be a 64-bit register.

Operation:

```
PMAXUB
    IF (fs[7..0] > ft[7..0]) THEN
        fd[7..0] ← fs[7..0];
    ELSE
        fd[7..0] ← ft[7..0];
    FI
    * repeat operation for 2nd through 7th bytes in first and second operands *
    IF (fs[63..56] > ft[63..56]) THEN
        fd[63..56] ← fs[63..56];
    ELSE
        fd[63..56] ← ft[63..56];
    FI
```

Exceptions:

None.

PMINSH—Minimum of Packed Signed Halfword Integers

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	PMINSH 10101	ft	fs	fd	ADD 000000
6	5	5	5	5	6

Format:

PMINSH fd,fs,ft

Description:

Performs a SIMD compare of the packed signed halfword integers in the first operand and the second operand, and returns the minimum value for each pair of halfword integers to the destination operand. The source operands can be a 64-bit register. The destination operand can be a 64-bit register.

Operation:

```
PMINSH
    IF (fs[15..0] < ft[15..0]) THEN
        fd[15..0] ← fs[15..0];
    ELSE
        fd[15..0] ← ft[15..0];
    FI
    * repeat operation for 2nd and 3rd halfwords in first and second operands *
    IF (fs[63..48] < ft[63..48]) THEN
        fd[63..48] ← fs[63..48];
    ELSE
        fd[63..48] ← ft[63..48];
    FI
```

Exceptions:

None.

PMINUB—Minimum of Packed Unsigned Byte Integers

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	PMINUB 10111	ft	fs	fd	ADD 000000
6	5	5	5	5	6

Format:

PMINUB fd,fs,ft

Description:

Performs a SIMD compare of the packed unsigned byte integers in the first operand and the second operand, and returns the minimum value for each pair of byte integers to the destination operand. The source operands can be a 64-bit register. The destination operand can be a 64-bit register.

Operation:

```
PMINUB
    IF (fs[7..0] < ft[7..0]) THEN
        fd[7..0] ← fs[7..0];
    ELSE
        fd[7..0] ← ft[7..0];
    FI
    * repeat operation for 2nd through 7th bytes in first and second operands *
    IF (fs[63..56] < ft[63..56]) THEN
        fd[63..56] ← fs[63..56];
    ELSE
        fd[63..56] ← ft[63..56];
    FI
```

Exceptions:

None.

PMOVMSKB—Move Byte Mask

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	PMOVMSKB 10101	0 00000	fs	fd	ABS 000101
6	5	5	5	5	6

Format:

PMOVMSKB fd,fs

Description:

Creates a mask made up of the most significant bit of each byte of the first operand and stores the result in the low byte of the destination operand. The source operand is a 64-bit register. When operating on 64-bit operands, the byte mask is 8 bits.

Operation:

```
PMOVMSKB
    fd[0]  ← fs[7];
    fd[1]  ← fs[15];
    * repeat operation for bytes 2 through 6 *
    fd[7]  ← fs[63];
    fd[63..8]  ← 00000000000000H;
```

Exceptions:

None.

PMULHUH—Multiply Packed Unsigned Integers and Store High Result

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	PMULHUH 10111	ft	fs	fd	MUL 000010
6	5	5	5	5	6

Format:

PMULHUH fd,fs,ft

Description:

Performs a SIMD unsigned multiply of the packed unsigned halfword integers in the first operand and the second operand, and stores the high 16 bits of each 32-bit intermediate results in the destination operand. (Figure 3-7 shows this operation when using 64-bit operands.) The source operands can be a 64-bit register. The destination operand can be a 64-bit register.

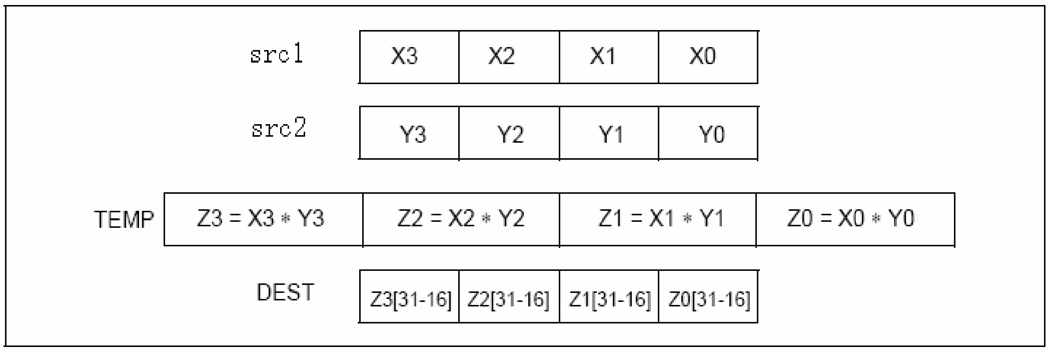

Figure 3-7. PMULHUH and PMULHH Instruction Operation Using 64-bit Operands

Operation:

PMULHUH

TEMP0[31..0] ← fs[15..0] ∗ ft[15..0]; * Unsigned multiplication *

```
TEMP1[31..0]  ← fs[31..16] * ft[31..16];
TEMP2[31..0]  ← fs[47..32] * ft[47..32];
TEMP3[31..0]  ← fs[63..48] * ft[63..48];
fd[15..0]     ← TEMP0[31..16];
fd[31..16]  ← TEMP1[31..16];
fd[47..32]  ← TEMP2[31..16];
fd[63..48]  ← TEMP3[31..16];
```

Exceptions:

None.

PMULHH—Multiply Packed Signed Integers and Store High Result

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	PMULHH 10101	ft	fs	fd	MUL 000010
6	5	5	5	5	6

Format:

PMULHH fd,fs,ft

Description:

Performs a SIMD signed multiply of the packed signed halfword integers in the first operand and the second operand, and stores the high 16 bits of each intermediate 32-bit result in the destination operand. (Figure 3-7 shows this operation when using 64-bit operands.) The source operands can be a 64-bit register. The destination operand can be a 64-bit register.

Operation:

```
PMULHH
    TEMP0[31..0]  ← fs[15..0] ∗ ft[15..0]; * Signed multiplication *
    TEMP1[31..0]  ← fs[31..16] ∗ ft[31..16];
    TEMP2[31..0]  ← fs[47..32] ∗ ft[47..32];
    TEMP3[31..0]  ← fs[63..48] ∗ ft[63..48];
    fd[15..0]    ← TEMP0[31..16];
    fd[31..16]  ← TEMP1[31..16];
    fd[47..32]  ← TEMP2[31..16];
    fd[63..48]  ← TEMP3[31..16];
```

Exceptions:

None.

PMULLH—Multiply Packed Signed Integers and Store Low Result

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	PMULLH 10100	ft	fs	fd	MUL 000010
6	5	5	5	5	6

Format:

PMULLH fd,fs,ft

Description:

Performs a SIMD signed multiply of the packed signed halfword integers in the first operand and the second operand, and stores the low 16 bits of each intermediate 32-bit result in the destination operand. (Figure 3-7 shows this operation when using 64-bit operands.) The source operand can be a 64-bit register. The destination operand can be a 64-bit register.

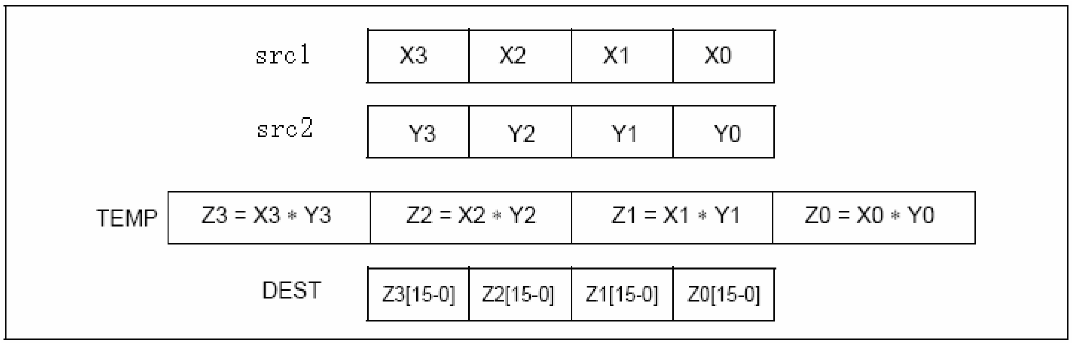

Figure 3-8. PMULLH Instruction Operation Using 64-bit Operands

Operation:

```
PMULLH
    TEMP0[31..0] ← fs[15..0] ∗ ft[15..0]; * Signed multiplication *
    TEMP1[31..0] ← fs[31..16] ∗ ft[31..16];
    TEMP2[31..0] ← fs[47..32] ∗ ft[47..32];
```

```
TEMP3[31..0]  ← fs[63..48] * ft[63..48];
fd[15..0]     ← TEMP0[15..0];
fd[31..16]  ← TEMP1[15..0];
fd[47..32]  ← TEMP2[15..0];
fd[63..48]  ← TEMP3[15..0];
```

Exceptions:

None.

PMULUW—Multiply Packed Unsignedword Integers

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	PMULUW 10110	ft	fs	fd	MUL 000010
6	5	5	5	5	6

Format:

PMULUW fd,fs,ft

Description:

Multiplies the first operand by the second operand and stores the result in the destination operand. The source operands can be a unsigned word integer stored in the low word of a 64-bit register. The result is an unsigned doubleword integer stored in the destination a 64-bit register. When a doubleword result is too large to be represented in 64 bits (overflow), the result is wrapped around and the low 64 bits are written to the destination element (that is, the carry is ignored).

Operation:

```
PMULUW
    fd[63..0] ← fs[31..0] * ft[31..0];
```

Exceptions:

None.

PSADBH—Compute Sum of Absolute Differences

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	PASUBUB 01101	ft	fs	fd	SUB 000001
6	5	5	5	5	6

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	BIADD 10100	0 00000	fs	fd	ABS 000101
6	5	5	5	5	6

Format:

PASUBUB	fd,fs,ft
BIADD	fd,fs

Description:

PSADBH instruction computes the absolute value of the difference of 8 unsigned byte integers from the first operand and from the second operand. These 8 differences are then summed to produce an unsigned halfword integer result that is stored in the destination operand. The source operand can be a 64-bit register. The destination operand can be a 64-bit register. Figure 3-9 shows the operation of the PSADBH instruction when using 64-bit operands. When operating on 64-bit operands, the halfword integer result is stored in the low halfword of the destination operand, and the remaining bytes in the destination operand are cleared to all 0s.

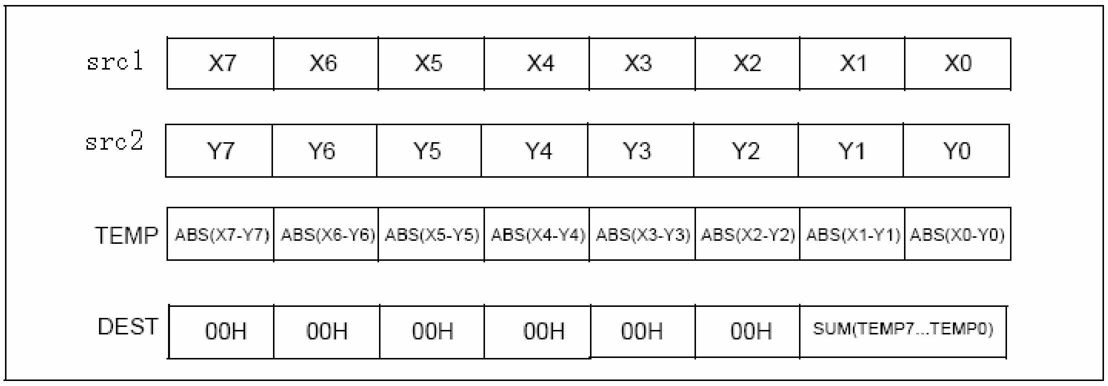

Figure 3-9. PSADBH Instruction Operation Using 64-bit Operands

Note: PSADBH instruction is divided into two instruction, PASUBUB and BIADD. PASUBUB instruction computes the absolute value of the difference of 8 unsigned byte integers from the first operand and from the second operand. BIADD computes the sum of 8 unsigned byte integers of the source operand.

Operation:

```
PASUBUB
    fd[7..0]  ← ABS(fs[7..0] – ft[7..0]);
    * repeat operation for bytes 2 through 6 *
    fd[63..56]  ← ABS(fs[63..56] – ft[63..56]);
BIADD
    fd[15..0]    ← SUM(fs[7..0]... fs[63..56]);
    fd[63..16]  ← 000000000000H;
```

Exceptions:

None.

PSHUFH—Shuffle Packed Halfwords

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	PSHUFH 11000	ft	fs	fd	MUL 000010
6	5	5	5	5	6

Format:

PSHUFH fd,fs,ft

Description:

Copies halfwords from the first operand and inserts them in the destination operand at halfword locations selected with the second operand(order operand). This operation is illustrated in Figure 3-10. For the PSHUFH instruction, each 2-bit field in the second operand selects the contents of one halfword location in the destination operand. The encodings of the second operand fields select halfwords from the first operand to be copied to the destination operand.

The first operand can be a 64-bit register. The destination operand is a 64-bit register. The order operand is a 64-bit register.

Note that this instruction permits a halfword in the first operand to be copied to more than one halfword location in the destination operand.

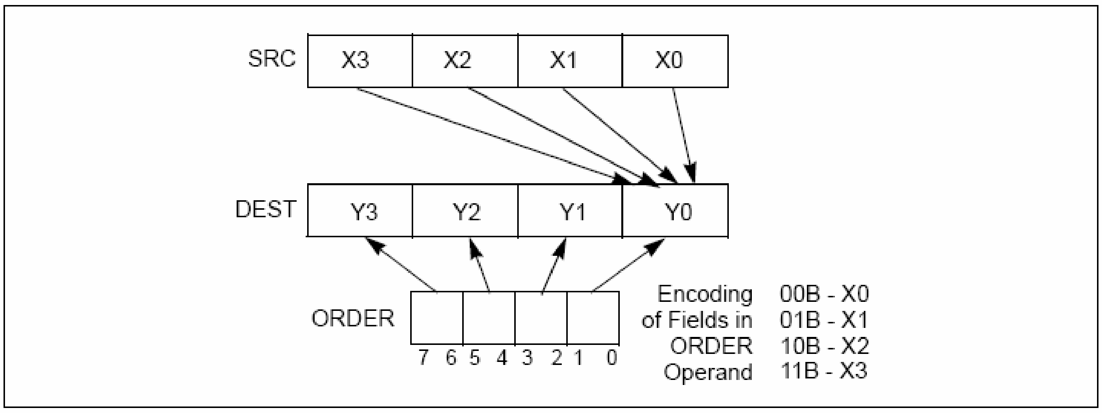

Figure 3-10. PSHUFH Instruction Operation

Operation:

PSHUFH

```
    fd[15..0]  ← (fs >> (ft[1..0] ∗ 16) )[15..0]
    fd[31..16] ← (fs >> (ft[3..2] ∗ 16) )[15..0]
    fd[47..32] ← (fs >> (ft[5..4] ∗ 16) )[15..0]
    fd[63..48] ← (fs >> (ft[7..6] ∗ 16) )[15..0]
```

Exceptions:

None.

PSLLH/PSLLW—Shift Packed Data Left Logical

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	PSLLH 10011	ft	fs	fd	MUL 000010
6	5	5	5	5	6

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	PSLLW 10010	ft	fs	fd	MUL 000010
6	5	5	5	5	6

Format:

PSLLH	fd,fs,ft
PSLLW	fd,fs,ft

Description:

Shifts the bits in the individual data elements (halfwords, words) in the first operand to the left by the number of bits specified in the second operand (count operand). As the bits in the data elements are shifted left, the empty low-order bits are cleared (set to 0). If the value specified by the count operand is greater than 15 (for halfwords), 31 (for words), then the destination operand is set to all 0s. (Figure 3-11 gives an example of shifting words in a 64-bit operand.)

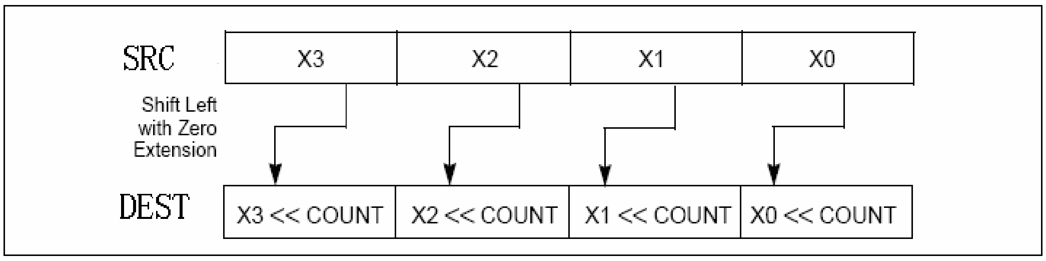

Figure 3-11. PSLLH, PSLLW Instruction Operation Using 64-bit Operand

The PSLLH instruction shifts each of the halfwords in the first operand to the left by the number of bits specified in the count operand; the PSLLW instruction shifts each of the words in the first operand.

Operation:

```
PSLLH
    IF (ft[6..0] > 15)
    THEN
        fd[64..0]   ← 0000000000000000H
    ELSE
        fd[15..0]   ← ZeroExtend(fs[15..0] << ft[6..0]);
        * repeat shift operation for 2nd and 3rd words *;
        fd[63..48]  ← ZeroExtend(fs[63..48] << ft[6..0]);
    FI;
PSLLW
    IF (ft[6..0] > 31)
    THEN
        fd[64..0]   ← 0000000000000000H
    ELSE
        fd[31..0]   ← ZeroExtend(fs[31..0] << ft[6..0]);
        fd[63..32]  ← ZeroExtend(fs[63..32] << ft[6..0]);
    FI;
```

Exceptions:

None.

PSRAH/PSRAW—Shift Packed Data Right Arithmetic

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	PSRAH 10101	ft	fs	fd	DIV 000011
6	5	5	5	5	6

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	PSRAW 10100	ft	fs	fd	DIV 000011
6	5	5	5	5	6

Format:

PSRAH	fd,fs,ft
PSRAW	fd,fs,ft

Description:

Shifts the bits in the individual data elements (halfwords or words) in the first operand to the right by the number of bits specified in the second operand (count operand). As the bits in the data elements are shifted right, the empty high-order bits are filled with the initial value of the sign bit of the data element. If the value specified by the count operand is greater than 15 (for halfwords) or 31 (for words), each destination data element is filled with the initial value of the sign bit of the element. (Figure 3-12 gives an example of shifting halfwords in a 64-bit operand.)

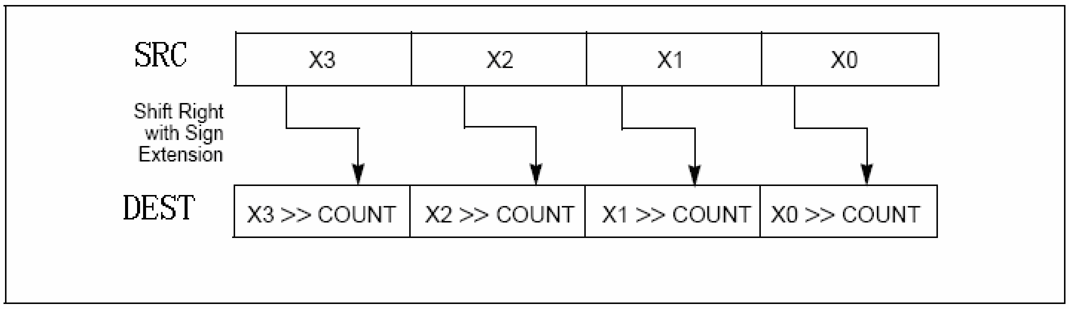

Figure 3-12. PSRAH and PSRAW Instruction Operation Using a 64-bit Operand

The PSRAH instruction shifts each of the halfwords in the first operand to the right by the number of bits specified in the count operand, and the PSRAW instruction shifts each of the words in the first operand.

Operation:

```
PSRAH
    IF (ft[6..0] > 15)
        THEN ft[6..0] ← 16;
    FI;
    fd[15..0]   ← SignExtend(fs[15..0] >> ft[6..0]);
    * repeat shift operation for 2nd and 3rd halfwords *;
    fd[63..48]  ← SignExtend(fs[63..48] >> ft[6..0]);
PSRAW
    IF (ft[6..0] > 31)
        THEN ft[6..0] ← 32;
    FI;
    fd[31..0]   ← SignExtend(fs[31..0] >> ft[6..0]);
    fd[63..32]  ← SignExtend(fs[63..32] >> ft[6..0]);
```

Exceptions:

None.

PSRLH/PSRLW—Shift Packed Data Right Logical

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	PSRLH 10011	ft	fs	fd	DIV 000011
6	5	5	5	5	6

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	PSRLW 10010	ft	fs	fd	DIV 000011
6	5	5	5	5	6

Format:

PSRLH	fd,fs,ft
PSRLW	fd,fs,ft

Description:

Shifts the bits in the individual data elements (halfwords, words) in the first operand to the right by the number of bits specified in the second operand (count operand). As the bits in the data elements are shifted right, the empty high-order bits are cleared (set to 0). If the value specified by the count operand is greater than 15 (for halfwords), 31 (for words), then the destination operand is set to all 0s. (Figure 3-13 gives an example of shifting halfwords in a 64-bit operand.)

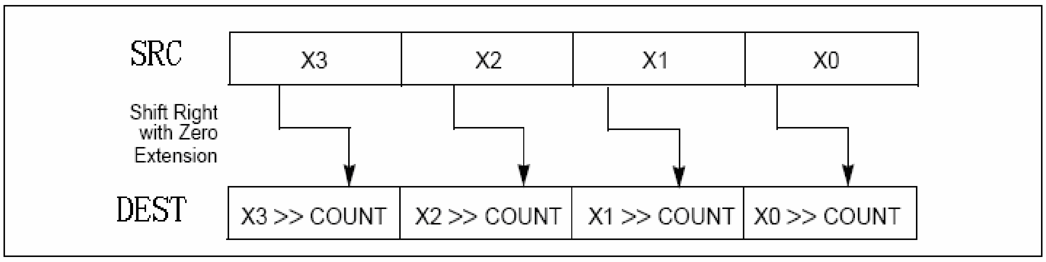

Figure 3-13. PSRLH, PSRLW Instruction Operation Using 64-bit Operand

The PSRLH instruction shifts each of the halfwords in the first operand to the right by the number of bits specified in the count operand; the PSRLW instruction shifts each of the words in the first operand.

Operation:

```
PSRLH
    IF (ft[6..0] > 15)
    THEN
        fd[64..0]   ← 0000000000000000H
    ELSE
        fd[15..0]   ← ZeroExtend(fs[15..0] >> ft[6..0]);
        * repeat shift operation for 2nd and 3rd halfwords *;
        fd[63..48]  ← ZeroExtend(fs[63..48] >> ft[6..0]);
    FI;
PSRLW
    IF (COUNT > 31)
    THEN
        fd[64..0]   ← 0000000000000000H
    ELSE
        fd[31..0]   ← ZeroExtend(fs[31..0] >> ft[6..0]);
        fd[63..32]  ← ZeroExtend(fs[63..32] >> ft[6..0]);
    FI;
```

Exceptions:

None.

PSUBB/PSUBH/PSUBW—Subtract Packed Integers

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	PSUBB 11110	ft	fs	fd	SUB 000001
6	5	5	5	5	6

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	PSUBH 11010	ft	fs	fd	SUB 000001
6	5	5	5	5	6

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	PSUBW 11011	ft	fs	fd	SUB 000001
6	5	5	5	5	6

Format:

```
PSUBB          fd,fs,ft
PSUBH          fd,fs,ft
PSUBW          fd,fs,ft
```

Description:

Performs a SIMD subtract of the packed integers of the second operand from the packed integers of the first operand, and stores the packed integer results in the destination operand. Overflow is handled with wraparound, as described in the following paragraphs. These instructions operate on 64-bit operands.

The PSUBB instruction subtracts packed byte integers. When an individual result is too large or too small to be represented in a byte, the result is wrapped around and the low 8 bits are written to the destination element.

The PSUBH instruction subtracts packed halfword integers. When an individual result is too large or too small to be represented in a halfword, the result is wrapped around and the low 16 bits are written to the destination element.

The PSUBW instruction subtracts packed word integers. When an individual result is too large or too small to be represented in a word, the result is wrapped around and the low 32 bits are written to the destination element.

Note that the PSUBB, PSUBW, and PSUBD instructions can operate on either unsigned or signed (two's complement notation) packed integers; however, it does not indicate overflow and/or a carry. To prevent undetected overflow conditions, software must control the ranges of values operated on.

Operation:

```
PSUBB
    fd[7..0]     ←  fs[7..0] – ft[7..0];
    * repeat subtract operation for 2nd through 7th byte *;
    fd[63..56]  ←  fs[63..56] – ft[63..56];
PSUBH
    fd[15..0]   ←  fs[15..0] – ft[15..0];
    * repeat subtract operation for 2nd and 3rd halfword *;
    fd[63..48]  ←  fs[63..48] – ft[63..48];
PSUBW
    fd[31..0]   ←  fs[31..0] – ft[31..0];
    fd[63..32]  ←  fs[63..32] – ft[63..32];
```

Exceptions:

None.

PSUBD—Subtract Packed Doubleword Integers

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	PSUBD 11111	ft	fs	fd	SUB 000001
6	5	5	5	5	6

Format:

PSUBD fd,fs,ft

Description:

Subtracts the second operand from the first operand and stores the result in the destination operand. When packed doubleword operands are used, a SIMD subtract is performed. When a doubleword result is too large to be represented in 64 bits (overflow), the result is wrapped around and the low 64 bits are written to the destination element (that is, the carry is ignored).

Note that the PSUBD instruction can operate on either unsigned or signed (two's complement notation) integers; however, it does not indicate overflow and/or a carry. To prevent undetected overflow conditions, software must control the ranges of the values operated on.

Operation:

```
PSUBD
    fd[63..0] ← fs[63..0] – ft[63..0];
```

Exceptions:

None.

PSUBSB/PSUBSH—Subtract Packed Signed Integers with Signed Saturation

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	PSUBSB 11100	ft	fs	fd	SUB 000001
6	5	5	5	5	6

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	PSUBSH 11000	ft	fs	fd	SUB 000001
6	5	5	5	5	6

Format:

```
PSUBSB          fd,fs,ft
PSUBSH          fd,fs,ft
```

Description:

Performs a SIMD subtract of the packed signed integers of the second operand from the packed signed integers of the first operand, and stores the packed integer results in the destination operand. Overflow is handled with signed saturation, as described in the following paragraphs. These instructions operate on 64-bit.

The PSUBSB instruction subtracts packed signed byte integers. When an individual byte result is beyond the range of a signed byte integer (that is, greater than 7FH or less than 80H), the saturated value of 7FH or 80H, respectively, is written to the destination operand.

The PSUBSH instruction subtracts packed signed halfword integers. When an individual halfword result is beyond the range of a signed halfword integer (that is, greater than 7FFFH or less than 8000H), the saturated value of 7FFFH or 8000H, respectively, is written to the destination operand.

Operation:

```
PSUBSB
    fd[7..0]    ← SaturateToSignedByte(fs[7..0] – ft[7..0]) ;
    * repeat subtract operation for 2nd through 7th bytes *;
    fd[63..56]  ← SaturateToSignedByte(fs[63..56] – ft[63..56] );
PSUBSH
    fd[15..0]   ←  SaturateToSignedHalfword(fs[15..0] – ft[15..0] );
    * repeat subtract operation for 2nd and 7th halfwords *;
    fd[63..48]  ←  SaturateToSignedHalfword(fs[63..48] – ft[63..48] );
```

Exceptions:

None.

PSUBUSB/PSUBUSH—Subtract Packed Unsigned Integers with Unsigned Saturation

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	PSUBUSB 11101	ft	fs	fd	SUB 000001
6	5	5	5	5	6

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	PSUBUSH 11001	ft	fs	fd	SUB 000001
6	5	5	5	5	6

Format:

```
PSUBUSB        fd,fs,ft
PSUBUSH        fd,fs,ft
```

Description:

Performs a SIMD subtract of the packed unsigned integers of thesecond operand from the packed unsigned integers of the first operand, and stores the packed unsigned integer results in the destination operand. Overflow is handled with unsigned saturation, as described in the following paragraphs. These instructions operate on 64-bit operands.

The PSUBUSB instruction subtracts packed unsigned byte integers. When an individual byte result is less than zero, the saturated value of 00H is written to the destination operand.

The PSUBUSH instruction subtracts packed unsigned halfword integers. When an individual halfword result is less than zero, the saturated value of 0000H is written to the destination operand.

Operation:

```
PSUBUSB
    fd[7..0]    ← SaturateToUnsignedByte(fs[7..0] – ft[7..0]) ;
    * repeat add operation for 2nd through 7th bytes *;
    fd[63..56]  ← SaturateToUnsignedByte(fs[63..56] – ft[63..56] );
PSUBUSH
    fd[15..0]   ← SaturateToUnsignedHalfword(fs[15..0] – ft[15..0] );
    * repeat add operation for 2nd and 3rd halfwords *;
    fd[63..48]  ← SaturateToUnsignedHalfword(fs[63..48] – ft[63..48] );
```

Exceptions:

None.

PUNPCKHBH/PUNPCKHHW/PUNPCKHWD—Unpack High Data

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	PUNPCKHBH 11011	ft	fs	fd	DIV 000011
6	5	5	5	5	6

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	PUNPCKHHW 11001	ft	fs	fd	DIV 000011
6	5	5	5	5	6

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	PUNPCKHWD 10111	ft	fs	fd	DIV 000011
6	5	5	5	5	6

Format:

```
PUNPCKHBH     fd,fs,ft
PUNPCKHHW     fd,fs,ft
PUNPCKHWD     fd,fs,ft
```

Description:

Unpacks and interleaves the high-order data elements (bytes,halfwords, words) of the first operand and second operand into the destination operand. (Figure 3-14 shows the unpack operation for bytes in 64-bit operands.). The low-order data elements are ignored.

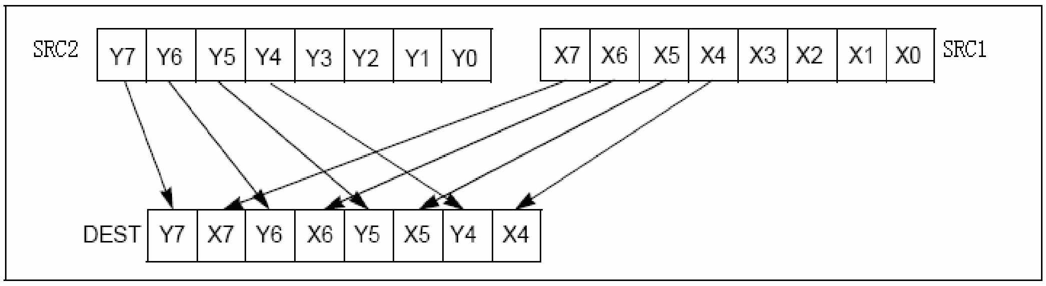

Figure 3-14. PUNPCKHBH Instruction Operation Using 64-bit Operands

The PUNPCKHBH instruction interleaves the high-order bytes of the first and second operands, the PUNPCKHHW instruction interleaves the high-order halfwords of the first and second operands, the PUNPCKHWD instruction interleaves the high-order word (or words) of first and second operands.

These instructions can be used to convert bytes to halfwords, halfwords to words, words to doublewords, respectively, by placing all 0s in the second operand. Here, if the second operand contains all 0s, the result (stored in the destination operand) contains zero extensions of the high-order data elements from the original value in the first operand. For example, with the PUNPCKHBH instruction the high-order bytes are zero extended (that is, unpacked into unsigned halfword integers), and with the PUNPCKHHW instruction, the high-order halfwords are zero extended (unpacked into unsigned word integers).

Operation:

```
PUNPCKHBH
    fd[7..0]    ← fs[39..32];
    fd[15..8]   ← ft[39..32];
    fd[23..16]  ← fs[47..40];
    fd[31..24]  ← ft[47..40];
    fd[39..32]  ← fs[55..48];
    fd[47..40]  ← ft[55..48];
    fd[55..48]  ← fs[63..56];
    fd[63..56]  ← ft[63..56];
PUNPCKHHW
    fd[15..0]   ← fs[47..32];
    fd[31..16]  ← ft[47..32];
    fd[47..32]  ← fs[63..48];
    fd[63..48]  ← ft[63..48];
PUNPCKHWD
    fd[31..0]   ← fs[63..32]
    fd[63..32]  ← ft[63..32];
```

Exceptions:

None.

PUNPCKLBH/PUNPCKLHW/PUNPCKLWD—Unpack Low Data

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	PUNPCKLBH 11010	ft	fs	fd	DIV 000011
6	5	5	5	5	6

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	PUNPCKLHW 11000	ft	fs	fd	DIV 000011
6	5	5	5	5	6

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	PUNPCKLWD 10110	ft	fs	fd	DIV 000011
6	5	5	5	5	6

Format:

PUNPCKLBH	fd,fs,ft
PUNPCKLHW	fd,fs,ft
PUNPCKLWD	fd,fs,ft

Description:

Unpacks and interleaves the low-order data elements (bytes, halfwords, words) of the first operand and second operand into the destination operand. (Figure 3-15 shows the unpack operation for bytes in 64-bit operands.). The high-order data elements are ignored.

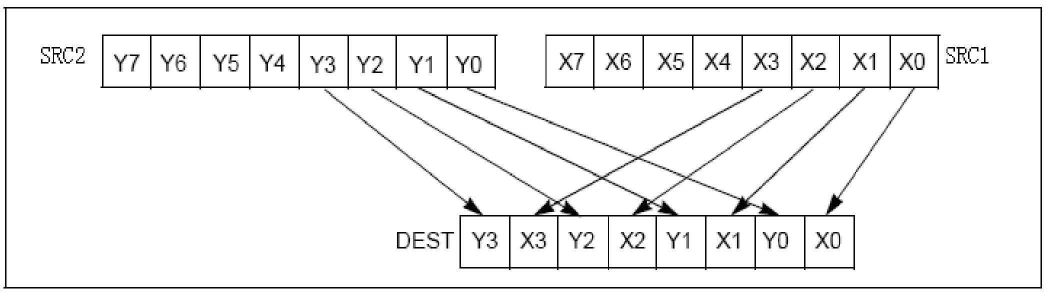

Figure 3-15. PUNPCKLBH Instruction Operation Using 64-bit Operands

The PUNPCKLBH instruction interleaves the low-order bytes of the first and second operands, the PUNPCKLHW instruction interleaves the low-order halfwords of the first and second operands, the PUNPCKLWD instruction interleaves the low-order word of the first and second operands.

These instructions can be used to convert bytes to halfwords, halfwords to words, words to doublewords, respectively, by placing all 0s in the secondoperand. Here, if the second operand contains all 0s, the result (stored in the destination operand) contains zero extensions of the high-order data elements from the original value in the first operand. For example, with the PUNPCKLBH instruction the high-order bytes are zero extended (that is, unpacked into unsigned halfword integers), and with the PUNPCKLHW instruction, the high-order halfwords are zero extended (unpacked into unsigned word integers).

Operation:

```
PUNPCKLBH
    fd[63..56]  ← ft[31..24];
    fd[55..48]  ← fs[31..24];
    fd[47..40]  ← ft[23..16];
    fd[39..32]  ← fs[23..16];
    fd[31..24]  ← ft[15..8];
    fd[23..16]  ← fs[15..8];
    fd[15..8]   ← ft[7..0];
    fd[7..0]    ← fs [7..0];
PUNPCKLHW
    fd[63..48]  ← ft[31..16];
    fd[47..32]  ← fs[31..16];
    fd[31..16]  ← ft[15..0];
    fd[15..0]   ← fs[15..0];
PUNPCKLWD
    fd[63..32]  ← ft[31..0];
    fd[31..0]   ← fs[31..0];
```

Exceptions:

None.