

龙芯 2E 体系结构 之多媒体指令

CurrentVersion: 0.4

Date: 2007-04-04

Author: comcat <jiankemeng@gmail.com>



版本历史

版本状态	作者	参与者	起止日期	备注
0.1	comcat		06-12-06	完成草稿
0.2	comcat		06-12-20	正式版发布
0.3	comcat		07-01-22	修正 3 处描述错误
0.4	comcat		07-04-02	修正 biadd 和 fadd 的描述
0.5	comcat		07-04-04	修正 pshufh 的描述

A. 概述

龙芯 2E 多媒体指令支持基于字节、半字、字以及双字整数的并行操作。这个与 Intel 的 MMX 很类似。事实上龙芯的多媒体指令吸取了 MMX、SSE 的优秀思想，并作了一些扩充。可惜目前龙芯仅支持对整数的并行操作，不支持 SIMD 浮点模型，期待龙芯的后序系列在多媒体方面能有大幅的提高。

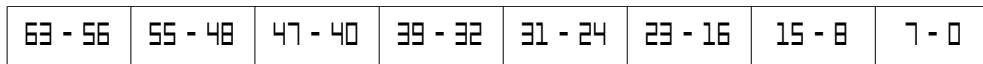
另龙芯实现了双单精度浮点指令 (Paired-single, PS), 可以同时处理 2 个单精度数的浮点操作，庶几可以弥补一些遗憾。

龙芯 2E 多媒体指令直接使用 FPU 的浮点寄存器 (64 位, 32 个) 作为多媒体寄存器。

龙芯 2E 多媒体指令的操作对象为 64 位，可以分割成 8×8 , 16×4 , 32×2 , 64×1 。如果每个单元为 8 位 (称为 Packed byte) 则有 8 个这样的单元；若每个单元为 32 位 (Packed word) 则有 2 个这样的单元。而每条指令可以同时处理这几个单元进行操作 (SIMD)。

这个在图像处理中很有用，可以同时处理 8 个像素 (8 位色, RGB, 每像素一个字节)。

Packed Byte



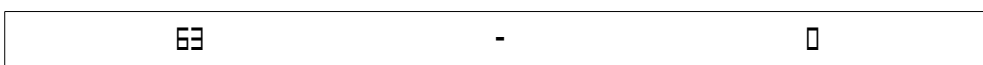
Packed Half-word



Packed Word



Packed Doubleword



注：龙芯体系结构下定义每个字(word)为 32 位，注意与 x86 的差别。

B. 龙芯多媒体指令介绍

注意：因为目前尚无公开的龙芯多媒体指令文档，以下属我分析，未经一一验证，大部分经验证、测试的指令会指出。

本文所有测试程序置于 <http://people.openrays.org/~comcat/>

先给个指令列表吧，该列表从龙芯开发者为汇编器 `as` 提供的补丁中取出。

```
punpcklbh/punpcklhw/punpcklwd    -- 解包指令
punpckhbh/punpckhhw/punpckhwd

packssbh/packsswh                -- 打包指令
packushb

pcmpgtb/pcmpgth/pcmpgtw          -- 比较赋值指令
pcmpeqb/pcmpeqh/pcmpeqw
pmaxsh
pmaxub
pminsh
pminub

paddb/paddh/paddw/paddd          -- 加法指令
paddsb/paddsh
paddusb/paddush

psubb/psubh/psubw/psubd          -- 减法指令
psubsb/psubsh
psubusb/psubush

pmullh                            -- 乘和乘加指令
pmulhh
pmuluw
pmulhuh
pmaddhw

pandn                              -- 逻辑指令

psrlh/psrlw                       -- 移位指令
psrah/psraw
psllh/psllw

pavgb/pavgh                       绿色
pshufh                             绿色
```

```
pmovmskb
pextrh

pinsrh_0/pinsrh_1/pinsrh_2/pinsrh_3

pasubub          -- 减取绝对值指令
biadd

for              -- 逻辑指令
fxor
fnor
fand

fsrl            -- 移位指令
fdsrl
fsra
fdsra
fsll
fdsll

fadd            -- 加减法指令
faddu
fdadd
fsub
fsubu
fsub

fseq            -- 比较赋值指令
fseq1
fslt
fsltu
fsle
fsleu
```

上述指令中，黑色的是可以对应于 **MMX** 指令，故而可以参考 **MMX** 的文档，绿色的对应 **SSE** 指令，红色的则为龙芯自定义的指令。

可以看到多媒体指令的后缀一般为：

b -- Byte 8 位
h -- Half-word 16 位
w -- Word 32 位
d -- Double-word 64 位
u -- Unsigned
s -- Saturation 表示饱和算法，参见本文档对 packssb 的描述。
有时 s 也表示 Signed

了解上述缩写的含义，有助于分析、记忆。

本文档称通用寄存器为 GPR，多媒体寄存器为 MMR，浮点寄存器为 FPR，浮点控制寄存器为 FCR。

龙芯体系结构下，多媒体指令由 FPU 执行，亦使用 FPU 的浮点寄存器作为多媒体寄存器，而在整个体系中 FPU 是作为 1 号协处理器出现，因此 MMR 与 GPR 之间的数据传送需要使用 dmtc1, dmfc1 这类指令。而 MMR 与内存之间的数据交换则要使用 ldc1, sdc1 这类指令。注意，用于 GPR 的指令不能施加于 MMR 上，如果要用的话，先要把 MMR 中的数据移到 GPR 中，运算完了再移回，对于一些常用的操作这个会影响效率，因而龙芯 2E 引入了可以直接施加于 MMR 上的常用操作(上文中的红色指令)，来提高多媒体的性能。

以下所述 DEST, SRC2, SRC1 均为 64 位多媒体寄存器

1. punpcklbh/punpcklhw/punpcklwd (Unpack Low data)

对包数据进行取低位解包

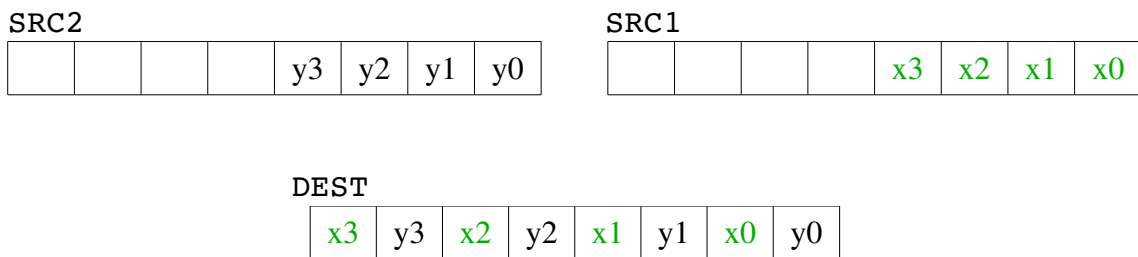
```

punpcklbh DEST, SRC2, SRC1    (Byte to Half-word)
punpcklhw DEST, SRC2, SRC1    (Half-word to Word)
punpcklwd DEST, SRC2, SRC1    (Word to Double-word)

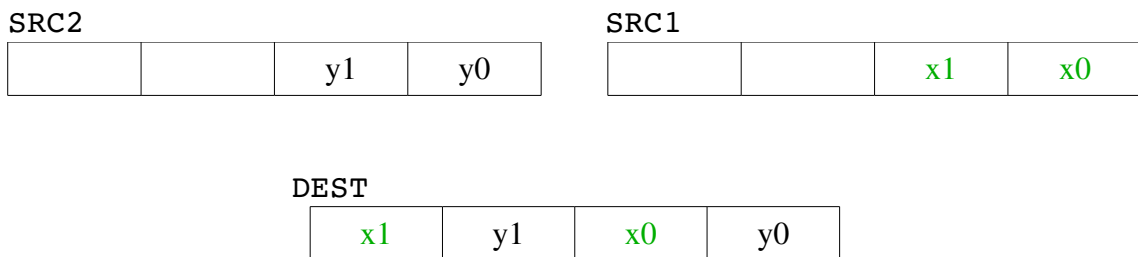
```

DEST, SRC2, SRC1 均为 64 位浮点寄存器

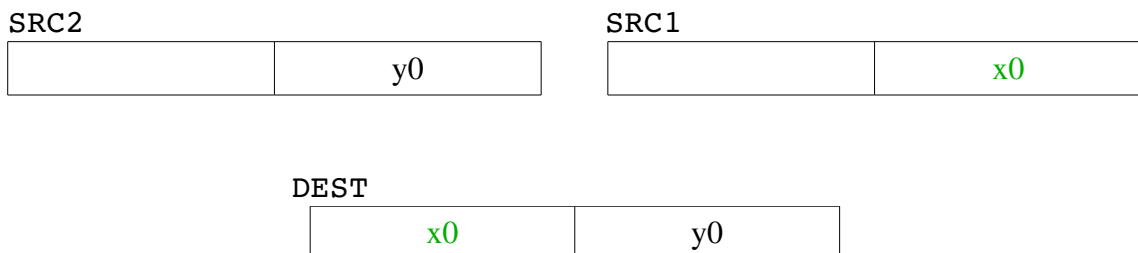
punpcklbh DEST, SRC2, SRC1 示意图:



punpcklhw DEST, SRC2, SRC1 示意图:



punpcklwd DEST, SRC2, SRC1 示意图:



Examples:

如将 f2 中的 8 位扩张到 16 位:

f2 值为 0x0000 0000 aa55 bbaa; f0 值为 0x0000 0000 0000 0000

```
punpcklbh $f2, $f2, $f0
```

执行后 f0 的值为: 0x00aa 0055 00bb 00aa

```
src1 data is: 0000, 0000, 0000, 0000
```

```
src2 data is: 0102, 001f, 85fd, aa82
```

```
punpcklbh src2, src1 result is: 0085, 00fd, 00aa, 0082
```

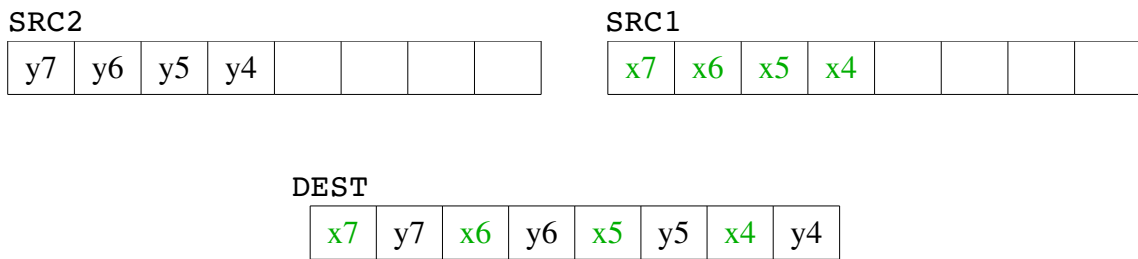
```
punpcklhw src2, src1 result is: 0000, 85fd, 0000, aa82
```

2. punpckhbh/punpckhwd/punpckhhw - Unpack High data

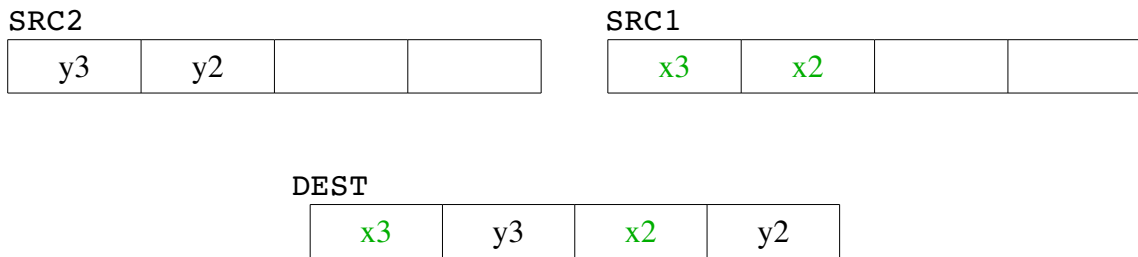
对包数据进行取高位解包

punpckhbh DEST, SRC2, SRC1 (Byte to Half-word)
 punpckhhw DEST, SRC2, SRC1 (Half-word to Word)
 punpckhwd DEST, SRC2, SRC1 (Word to Double-word)

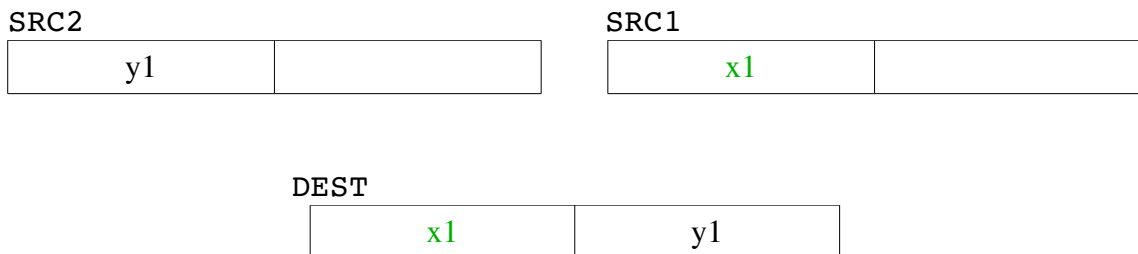
punpckhbh DEST, SRC2, SRC1 示意图:



punpckhhw DEST, SRC2, SRC1 示意图:



punpckhwd DEST, SRC2, SRC1 示意图:

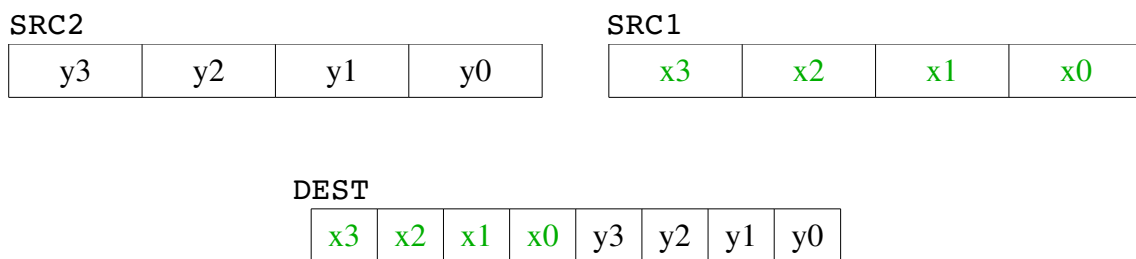


3. packsshb/packsswh - Pack with Signed Saturation

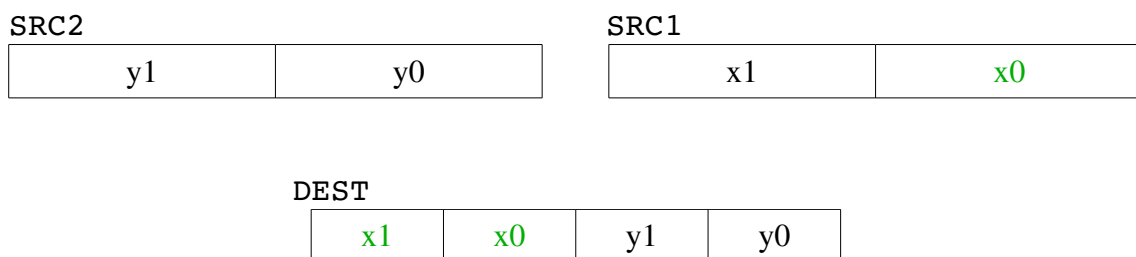
对有符号数据使用饱和算法打包

packsshb DEST, SRC2, SRC1 (Half-word to Byte)
packsswh DEST, SRC2, SRC1 (Word to Half-word)

packsshb DEST, SRC2, SRC1 示意图:



packsswh DEST, SRC2, SRC1 示意图:



packsshb 将 16 位打包入 8 位中，使用 8 位有符号饱和算法

packsswh 将 32 位打包入 16 位中，使用 16 位有符号饱和算法

关于饱和算法(Saturation Algorithm):

在计算机系统中参与运算的单元，字长固定，若运算结果超过固定字长则会“溢出”，如 8 位机上，1000 0000 与 1100 0000 相加，其结果的第 9 位进位就会舍去。在多媒体处理中，采用这种方法处理溢出是不行的。比如对一副 RGB 24 位图像进行加亮运算，每个颜色值的运算结果必须在 0~255 之间，如果一个像素的颜色值为 0xffffff，每种色彩再加 1 的话，按通常处理溢出的方法，结果为 0x000000，就是这个像素的颜色一下子从白色变成了黑色，这是与我们想要的结果是不相称的。故而我们会用如下语句来处理溢出的情况：

```
if(G > 255)
    G = 255;
else if(G < 0)
    G = 0;
```

对 R, B 的处理与此同。

在多媒体信息的处理中，需要大量用到这种算法，即：对产生溢出的运算将累积至固定单元可容纳的最大（或最小）值。

如上面的 packsshb 指令，就会使用 8 位有符号饱和，其范围为：-128~127 之间，亦有无符号饱和，如上面的颜色运算就会用到 8 位无符号饱和，其范围为 0~255。即当结果超过上限（上溢）时取固定单元能表示的最大值，当结果超过下限（下溢）时取固定单元能表示的最小值。

使用这样的多媒体指令，就无需那一堆判断语句，大大提高处理的效率。

Examples:

```
src1: 0x0000 7eff 00ff ff01
src2: 0x0102 001f 85fd aa82
```

packsshb dest, src2, src1 的结果就为： 0x007f 7f80 7f1f 8080

注意系统中使用补码表示有符号整数，即 0xff 为 -1，0x80 为 -128，0x7f 为 127

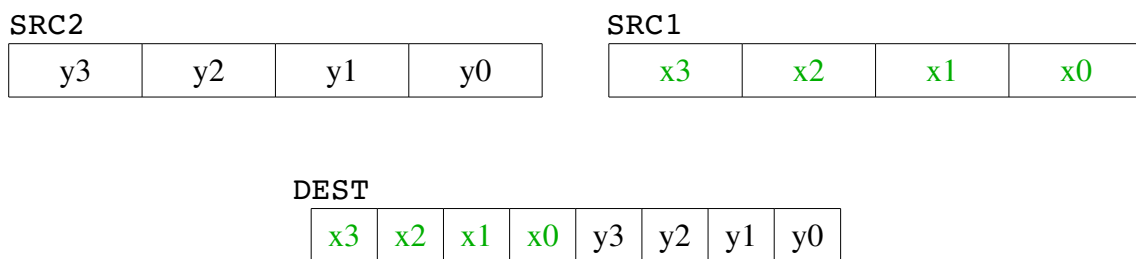
4. packushb - Pack with Unsigned Saturation

对无符号数据使用饱和算法打包

`packushb DEST, SRC2, SRC1` (Half-word to Byte)

将 16 位打包入 8 位中，使用 8 位无符号饱和算法（即饱和范围为：0~255）

`packushb DEST, SRC2, SRC1` 示意图：

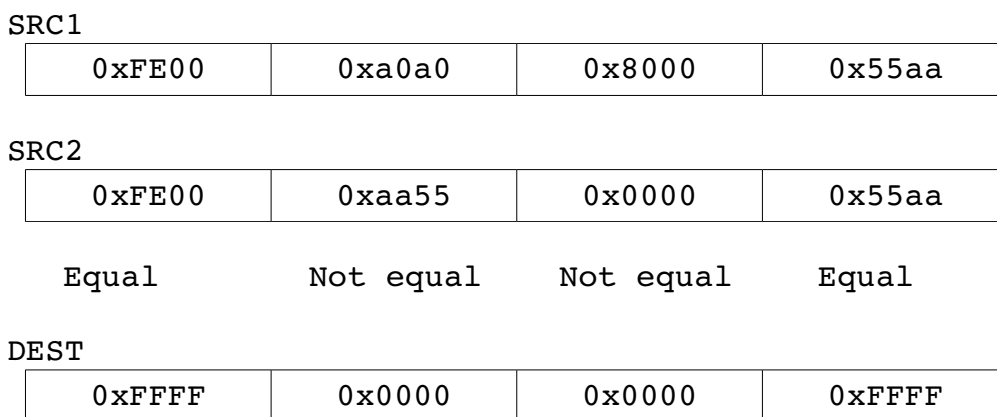


5. pcmpeqb/pcmpeqh/pcmpeqw - Compare Packed Data for Equal

对包数据进行比较，相等则包内的所有位置 1，否则置 0

```
pcmpeqb DEST, SRC2, SRC1    (Packed Byte)
pcmpeqh DEST, SRC2, SRC1    (Packed Half-word)
pcmpeqw DEST, SRC2, SRC1    (Packed Word)
```

pcmpeqh DEST, SRC2, SRC1 操作示意图:



使用该指令，很容易计算一幅图像的屏蔽图。即：颜色相同的地方为 1，不同的地方为 0

Tips:

将寄存器 `f0` 的值置为 `0xffff ffff ffff ffff`

```
pcmpeqw $f0, $f0, $f0
```

Example:

设有两个图，**A** 图是人，**B** 图是场景，现要将人写进这个场景中，如果使用一般的指令则会有如下过程：

```
if (图 A 当前像素颜色 is 屏蔽色) then
    图 D <-- 图 B 对应的像素颜色
else
    图 D <-- 图 A 对应的像素颜色
```

屏蔽色显示为透明。

这样就得一个像素一个像素的算，还有大量的判断，运行效率会很低，因为一旦 CPU 转移预测失败，流水线后面的指令都得清空（除了位于转移延迟槽中的指令）。

如果使用多媒体指令来实现，可以使用 `pcmpeqb` 同时将 **A** 图的 8 个像素（8 位色）与屏蔽色比较，是屏蔽色的像素全置 1，否则置 0，这样就获得了图 **A** 的屏蔽图，将其和图 **B** 作“与”运算就将场景要写入图 **A** 的像素提取出来了。这种处理方式避免了大量的条件判断语句，大大提高了运算效率。

6. pcmpgtb/pcmpgth/pcmpgtw - Compare Packed Signed Integers for Greater Than

对有符号包数据进行比较，大于则包内的所有位置 1，否则置 0

```
pcmpgtb DEST, SRC2, SRC1    (Packed Byte)
pcmpgth DEST, SRC2, SRC1    (Packed Half-word)
pcmpgtw DEST, SRC2, SRC1    (Packed Word)
```

pcmpgth DEST, SRC2, SRC1 操作示意图:

SRC1			
0xFE00	0xa0a0	0x8000	0x55aa
SRC2			
0xFE00	0xaa55	0x0000	0x55aa
Not Greater	Greater	Greater	Not Greater
DEST			
0x0000	0xFFFF	0xFFFF	0x0000

Examples:

```
src1: 0x0000 7eff 00ff ff01
src2: 0x0102 001f 85fd aa82
```

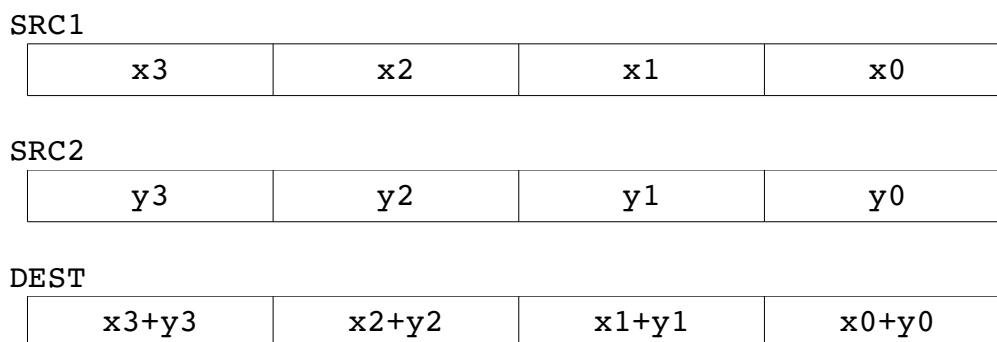
pcmpgth dest, src2, src1 结果为: 0xffff 0000 0000 0000

7. paddb/paddh/paddw/paddd - Add Packed Integers

对包数据同时相加

```
paddb DEST, SRC2, SRC1 (Packed Byte)
paddh DEST, SRC2, SRC1 (Packed Half-word)
paddw DEST, SRC2, SRC1 (Packed Word)
paddd DEST, SRC2, SRC1 (Packed Double-word)
```

paddh DEST, SRC2, SRC1 操作示意图（以半字 16 位为一个包数据）：



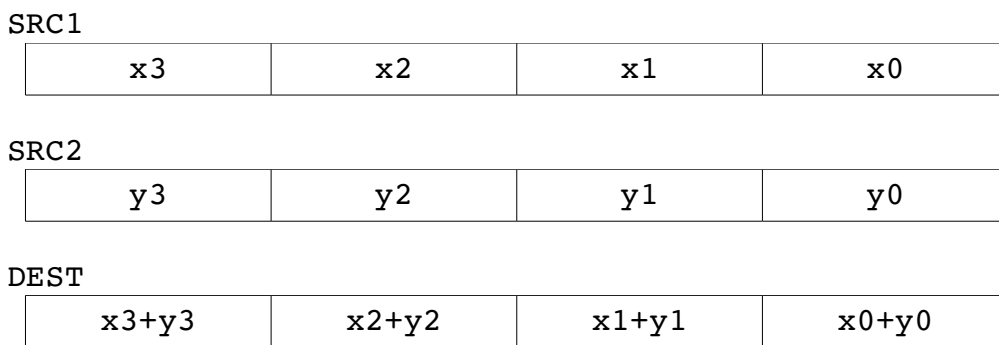
其余指令，操作类似，只是包数据的位长不一而已。

8. paddsb/paddsh - Add Packed Signed Integers with Signed Saturation

对有符号包数据进行有符号饱和相加

```
paddsb DEST, SRC2, SRC1
paddsh DEST, SRC2, SRC1
```

paddsh DEST, SRC2, SRC1 操作示意图（以半字 16 位为一个包数据）：



每个 16 位的包数据的结果范围为： $-2^{15} \sim 2^{15}-1$ ，结果上溢取最大值，下溢取最小值。

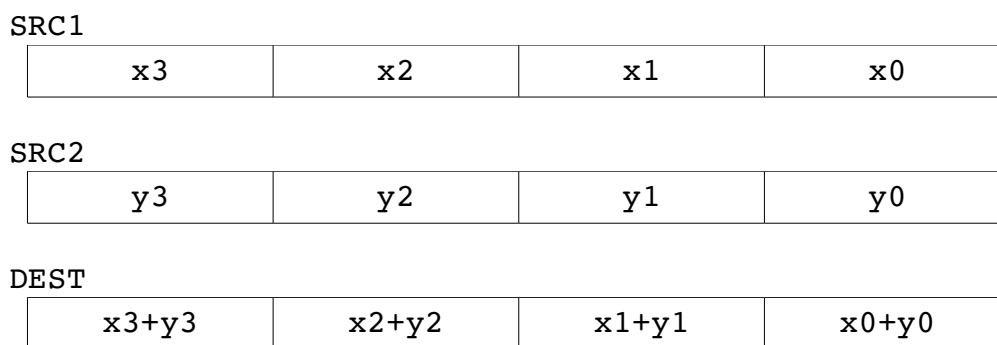
paddsb 操作类似，只是包数据位长为 8 位，结果范围为： $-2^7 \sim 2^7-1$

10. paddusb/paddush - Add Packed Unsigned Integers with Unsigned Saturation

对无符号包数据进行无符号饱和相加

```
paddusb DEST, SRC2, SRC1
paddush DEST, SRC2, SRC1
```

`paddush DEST, SRC2, SRC1` 操作示意图（以半字 16 位为一个包数据）：



每个 16 位的包数据的结果范围为：0 ~ $2^{16}-1$ ，结果上溢取最大值，下溢取最小值。

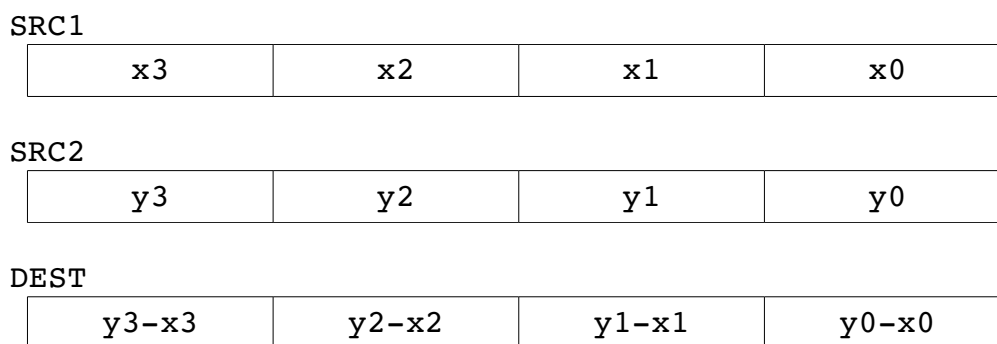
`paddusb` 操作类似，只是包数据位长为 8 位，结果范围为：0 ~ 2^8-1

11. psubb/psubh/psubw/psubd - Subtract Packed Integers

对包数据同时相减

```
psubb DEST, SRC2, SRC1 (Packed Byte)
psubh DEST, SRC2, SRC1 (Packed Half-word)
psubw DEST, SRC2, SRC1 (Packed Word)
psubd DEST, SRC2, SRC1 (Packed Double-word)
```

psubh DEST, SRC2, SRC1 操作示意图（以半字 16 位为一个包数据）：



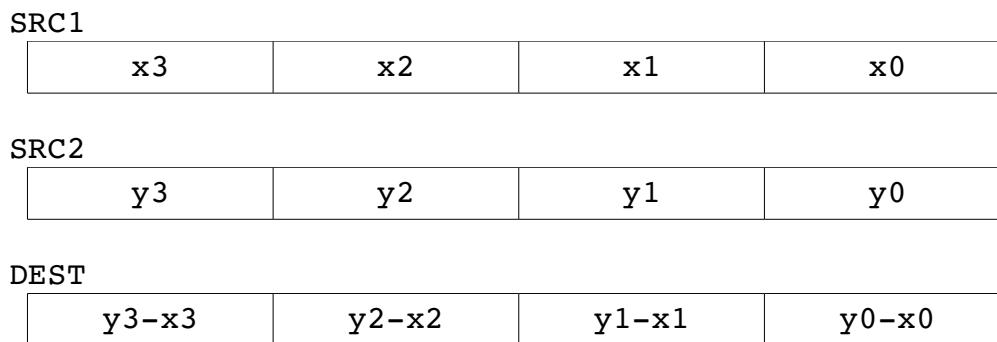
其余指令，操作类似，只是包数据的位长不一而已。

12. psubsb/psubsh - Subtract Packed Signed Integers with Signed Saturation

对有符号包数据进行有符号饱和相减

```
psubsb DEST, SRC2, SRC1
psubsh DEST, SRC2, SRC1
```

psubsh DEST, SRC2, SRC1 操作示意图（以半字 16 位为一个包数据）：



每个 16 位的包数据的结果范围为： $-2^{15} \sim 2^{15}-1$ ，结果上溢取最大值，下溢取最小值。

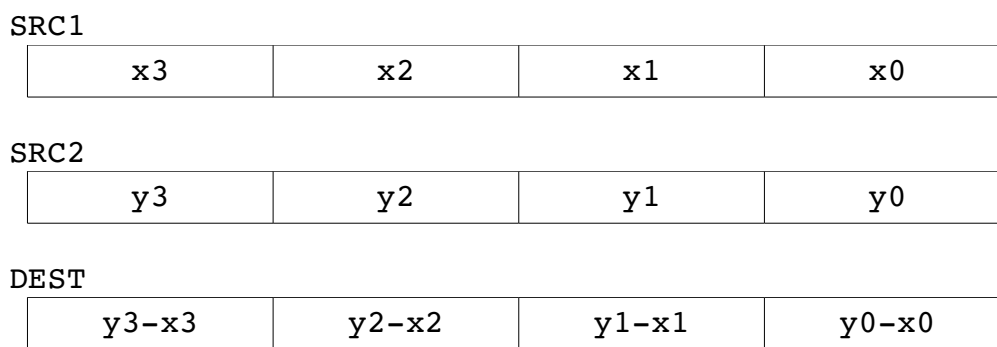
psubsb 操作类似，只是包数据位长为 8 位，结果范围为： $-2^7 \sim 2^7-1$

13. psubusb/psubush - Subtract Packed Unsigned Integers with Unsigned Saturation

对无符号包数据进行无符号饱和相减

```
psubusb DEST, SRC2, SRC1    (Packed Byte)
psubush DEST, SRC2, SRC1    (Packed Half-word)
```

psubush DEST, SRC2, SRC1 操作示意图（以半字 16 位为一个包数据）：



每个 16 位的包数据的结果范围为：0 ~ $2^{16}-1$ ，结果上溢取最大值，下溢取最小值。

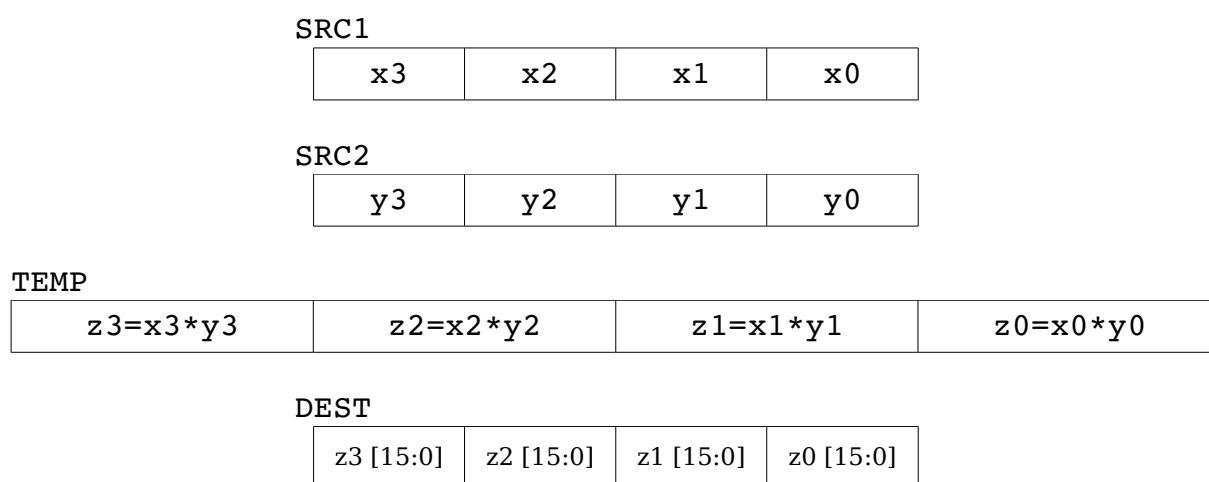
psubusb 操作类似，只是包数据位长为 8 位，结果范围为：0 ~ 2^8-1

14. pmullh - Multiply Packed Signed Integers and Store Low Result

对有符号半字包数据进行乘运算，取结果的低位

pmullh DEST, SRC2, SRC1 (Packed Half-word)

操作示意图（以半字 16 位为一个包数据）：



TEMP0[31:0] = SRC2[15:0] * SRC1[15:0] (有符号相乘)

TEMP1[31:0] = SRC2[31:16] * SRC1[31:16]

TEMP2[31:0] = SRC2[47:32] * SRC1[47:32]

TEMP3[31:0] = SRC2[63:48] * SRC1[63:48]

DEST[15:0] = TEMP0[15:0]

DEST[31:16] = TEMP1[15:0]

DEST[47:32] = TEMP2[15:0]

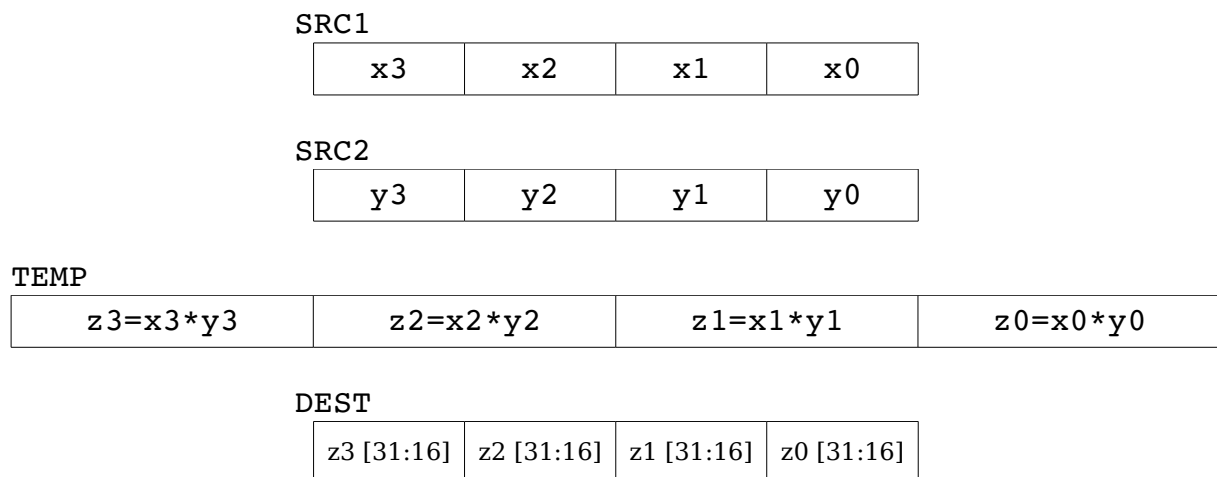
DEST[63:48] = TEMP3[15:0]

15. pmulhh - Multiply Packed Signed Integers and Store High Result

对有符号半字包数据进行乘运算，取结果的高位

pmulhh DEST, SRC2, SRC1 (Packed Half-word)

操作示意图（以半字 16 位为一个包数据）：



TEMP0[31:0] = SRC2[15:0] * SRC1[15:0] (有符号相乘)
 TEMP1[31:0] = SRC2[31:16] * SRC1[31:16]
 TEMP2[31:0] = SRC2[47:32] * SRC1[47:32]
 TEMP3[31:0] = SRC2[63:48] * SRC1[63:48]

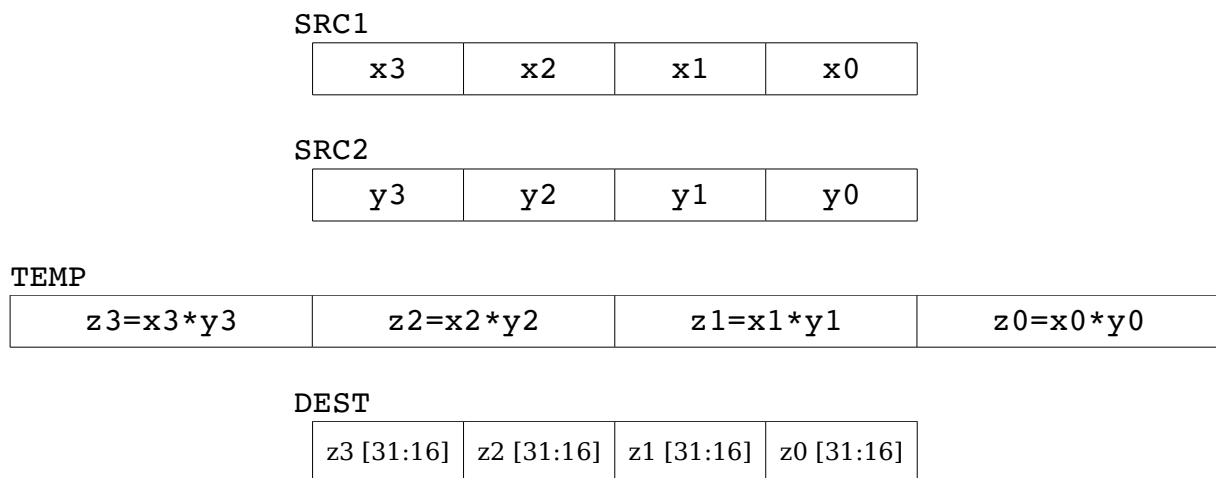
DEST[15:0] = TEMP0[31:16]
 DEST[31:16] = TEMP1[31:16]
 DEST[47:32] = TEMP2[31:16]
 DEST[63:48] = TEMP3[31:16]

16. pmulhuh - Multiply Packed Unsigned Integers and Store High Result

对无符号半字包数据进行乘运算，取结果的高位

`pmulhuh DEST, SRC2, SRC1` (Packed Half-word)

操作示意图（以半字 16 位为一个包数据）：



$\text{TEMP0}[31:0] = \text{SRC2}[15:0] * \text{SRC1}[15:0]$ (无符号相乘)
 $\text{TEMP1}[31:0] = \text{SRC2}[31:16] * \text{SRC1}[31:16]$
 $\text{TEMP2}[31:0] = \text{SRC2}[47:32] * \text{SRC1}[47:32]$
 $\text{TEMP3}[31:0] = \text{SRC2}[63:48] * \text{SRC1}[63:48]$

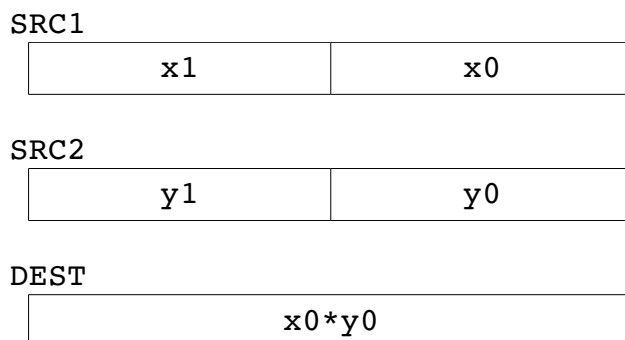
$\text{DEST}[15:0] = \text{TEMP0}[31:16]$
 $\text{DEST}[31:16] = \text{TEMP1}[31:16]$
 $\text{DEST}[47:32] = \text{TEMP2}[31:16]$
 $\text{DEST}[63:48] = \text{TEMP3}[31:16]$

17. pmuluw - Multiply Packed Unsigned Word Integers

对无符号字包数据进行乘运算

```
pmuluw DEST, SRC2, SRC1
```

操作示意图（以字 32 位为一个包数据）：



$$\text{DEST}[63:0] = \text{SRC2}[31:0] * \text{SRC1}[31:0]$$

Example:

```
src1 : 0x8000 00ff 0000 0010
src2 : 0x8000 001f 0000 0002
```

```
pmuluw dest, src2, src1 结果为: 0x0000 0000 0000 0020
```

```
src1 : 0x0000 0000 8000 0000
src2 : 0x0000 0000 0000 0002
```

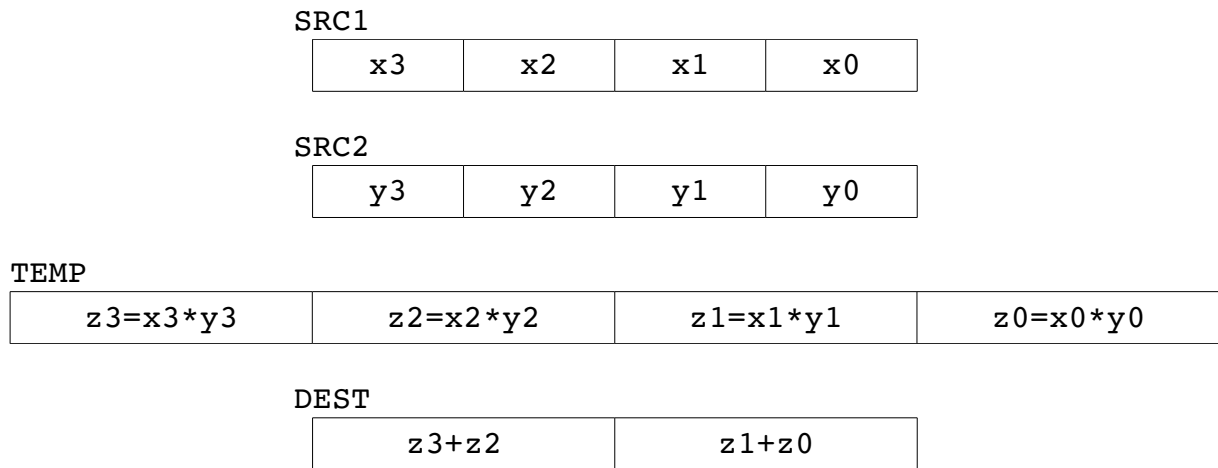
```
pmuluw dest, src2, src1 结果为: 0x0000 0001 0000 0000
```

18. pmaddhw - Multiply and Added Packed Integers

对有符号半字包数据做乘运算，结果暂存于 32 位字包中，后对暂存的字包数据进行两两加运算

```
pmaddhw DEST, SRC2, SRC1
```

操作示意图（以半字 16 位为一个包数据）：



```
DEST[31:0] = ( SRC2[31:16]*SRC1[31:16] + SRC2[15:0]*SRC1[15:0] )
DEST[63:32] = ( SRC2[63:48]*SRC1[63:48] + SRC2[47:32]*SRC1[47:32] )
```

Example:

```
src1 : 0x0007 0005 0003 0001
src2 : 0x0008 0006 0004 0002
```

```
pmaddhw dest, src2, src1 结果为: 0x0000 0056 0000 000e
```

19. pandn - Logical AND NOT

按位与或

```
pandn DEST, SRC2, SRC1
```

操作:

```
DEST = ((NOT SRC2) AND SRC1)
```

Example:

```
src1: 0x0000 7eff 00ff ff01
```

```
src2: 0x0102 001f 85fd aa82
```

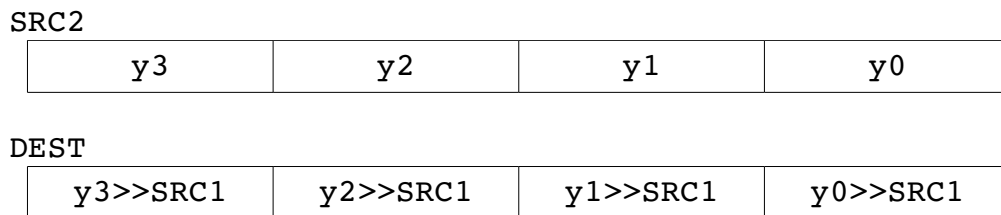
```
pandn dest, src2, src1 的结果为: 0000, 7ee0, 0002, 5501
```

20. psrlh/psrlw - Shift Packed Data Right Logical

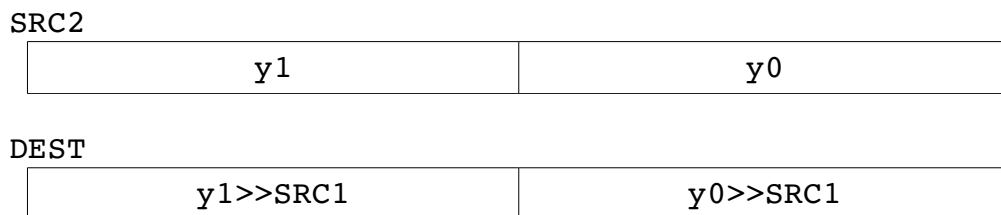
对包数据分别逻辑右移

```
psrlh DEST, SRC2, SRC1
psrlw DEST, SRC2, SRC1
```

psrlh DEST, SRC2, SRC1 操作示意图:



psrlw DEST, SRC2, SRC1 操作示意图:



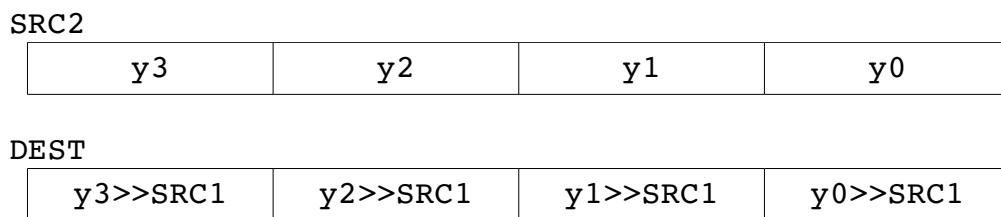
逻辑右移，左边补零。

21. psrah/psraw - Shift Packed Data Right Arithmetic

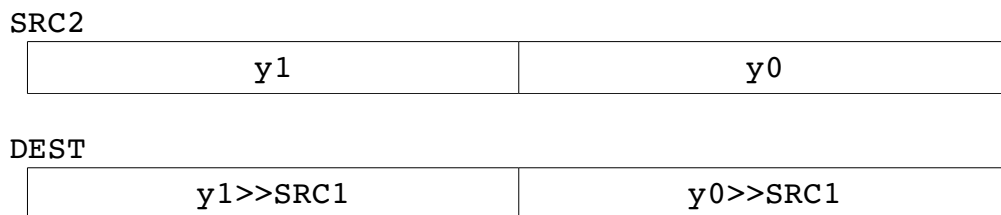
对包数据分别算术右移

```
psrah DEST, SRC2, SRC1
psraw DEST, SRC2, SRC1
```

psrah DEST, SRC2, SRC1 操作示意图:



psraw DEST, SRC2, SRC1 操作示意图:



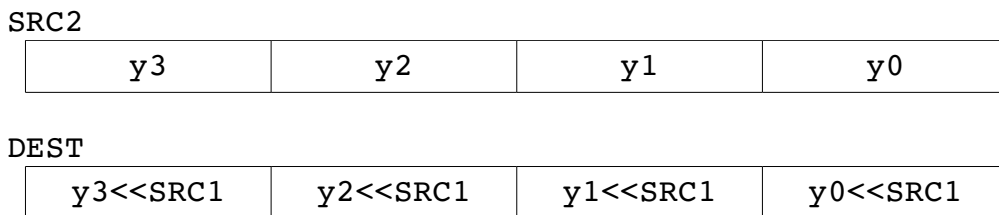
算术右移，负数左边补 1，正数左边补 0

22. psllh/psllw - Shift Packed Data Left Logical

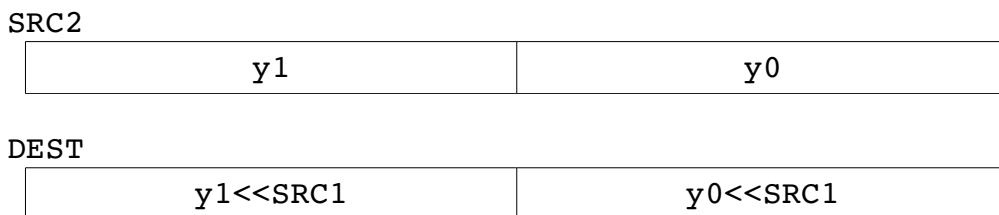
对包数据分别逻辑左移

```
psllh DEST, SRC2, SRC1
psllw DEST, SRC2, SRC1
```

psllh DEST, SRC2, SRC1 操作示意图:



psllw DEST, SRC2, SRC1 操作示意图:



逻辑左移，右边补零。

23. pmaxsh - Maximum of Packed Signed Half-word Integers

对有符号半字包数据进行比较，取较大的值

pmaxsh DEST, SRC2, SRC1

SRC1

x3	x2	x1	x0
----	----	----	----

SRC2

y3	y2	y1	y0
----	----	----	----

DEST

max(y3,x3)	max(y2,x2)	max(y1,x1)	max(y0,x0)
------------	------------	------------	------------

操作:

```
IF SRC2[15:0] > SRC1[15:0] THEN
  DEST[15:0] = SRC2[15:0]
ELSE
  DEST[15:0] = SRC1[15:0]
FI
```

(*repeat operation for 2nd and 3rd half-words in SRC1 and SRC2)

```
IF SRC2[63:48] > SRC1[63:48] THEN
  DEST[63:48] = SRC2[63:48]
ELSE
  DEST[63:48] = SRC1[63:48]
FI
```

24. pmaxub - Maximum of Packed Unsigned Byte Integers

对无符号字节包数据进行比较，取较大的值

```
pmaxub DEST, SRC2, SRC1
```

操作:

```
IF SRC2[7:0] > SRC1[7:0] THEN
    DEST[7:0] = SRC2[7:0]
ELSE
    DEST[7:0] = SRC1[7:0]
FI
```

(*repeat operation for 2nd through 7th bytes in SRC1 and SRC2)

```
IF SRC2[63:56] > SRC1[63:56] THEN
    DEST[63:56] = SRC2[63:56]
ELSE
    DEST[63:56] = SRC1[63:56]
FI
```

25. pminsh - Minimum of Packed Signed Half-word Integers

对有符号半字包数据进行比较，取较小的值

pminsh DEST, SRC2, SRC1

SRC1

x3	x2	x1	x0
----	----	----	----

SRC2

y3	y2	y1	y0
----	----	----	----

DEST

min(y3,x3)	min(y2,x2)	min(y1,x1)	min(y0,x0)
------------	------------	------------	------------

操作:

```
IF SRC2[15:0] < SRC1[15:0] THEN
  DEST[15:0] = SRC2[15:0]
ELSE
  DEST[15:0] = SRC1[15:0]
FI
```

(*repeat operation for 2nd and 3rd half-words in SRC1 and SRC2)

```
IF SRC2[63:48] < SRC1[63:48] THEN
  DEST[63:48] = SRC2[63:48]
ELSE
  DEST[63:48] = SRC1[63:48]
FI
```

26. pminub - Minimum of Packed Unsigned Byte Integers

对无符号字节包数据进行比较，取较小的值

```
pminub DEST, SRC2, SRC1
```

操作:

```
IF SRC2[7:0] < SRC1[7:0] THEN
    DEST[7:0] = SRC2[7:0]
ELSE
    DEST[7:0] = SRC1[7:0]
FI
```

(*repeat operation for 2nd through 7th bytes in SRC1 and SRC2)

```
IF SRC2[63:56] < SRC1[63:56] THEN
    DEST[63:56] = SRC2[63:56]
ELSE
    DEST[63:56] = SRC1[63:56]
FI
```

27. pavgb/pavgh - Average Packed Integers

对无符号包数据求平均值

```
pavgb DEST, SRC2, SRC1    (Packed Byte)
pavgh DEST, SRC2, SRC1    (Packed Half-word)
```

pavgb 操作:

```
DEST[7:0] = (SRC2[7:0] + SRC1[7:0] + 1) >> 1
```

(*repeat operation for 2nd through 7th bytes in SRC1 and SRC2)

```
DEST[63:56] = (SRC2[63:56] + SRC1[63:56] + 1) >> 1
```

pavgh 操作:

```
DEST[15:0] = (SRC2[15:0] + SRC1[15:0] + 1) >> 1
```

(*repeat operation for 2nd and 3rd bytes in SRC1 and SRC2)

```
DEST[63:48] = (SRC2[63:48] + SRC1[63:48] + 1) >> 1
```

Example:

```
src1 : 0x0100 ffff 00ff 0001
src2 : 0x0102 feff 00fd ffff
```

```
pavgb dest, src2, src1 结果为: 0x01 01 ff ff 00 fe 80 80
```

```
pavgh dest, src2, src1 结果为: 0x0101 ff7f 00fe 8000
```

28. pshufh - Shuffle Packed Half-words

对半字包数据进行交换

```
pshufh DEST, SRC2, SRC1 (Packed Half-word)
```

操作:

```
DEST[15:0] = (SRC2 >> (SRC1[1:0] * 16))[15:0]
DEST[31:16] = (SRC2 >> (SRC1[3:2] * 16))[15:0]
DEST[47:32] = (SRC2 >> (SRC1[5:4] * 16))[15:0]
DEST[63:48] = (SRC2 >> (SRC1[7:6] * 16))[15:0]
```

Example:

```
src1 : 0xf4(11 11 01 00)
src2 : 0x0100 ffff 00ff aa55
```

```
pshufh dest, src2, src1 结果为: 0x0100 0100 00ff aa55
```

```
src1 : 0x00(00 00 00 00)
src2 : 0xaadd
```

```
pshufh dest, src2, src1 结果为: 0xaadd aadd aadd aadd
```

Tips:

I. 将 f0 的值传给 f2

```
li      $13, 0xe4      (0xe4: 11 10 01 00)
dmtc1   $13, $f4
pshufh  $f2, $f0, $f4
```

II. 交换 f2 的高位字和低位字

```
li      $13, 0x4e      (0x4e: 01 00 11 10)
dmtc1   $13, $f0
pshufh  $f2, $f2, $f0
```

III. 将 f2 置为 0xaa55 aa55 aa55 aa55

```
fxor    $f0, $f0, $f0
li      $13, 0xaa55
dmtc1   $13, $f2
pshufh  $f2, $f2, $f0
```

IV. 将 f2 置为 0xFF00 FF00 FF00 FF00

```
pcmpeqw $f0, $f0, $f0 # $f0 = 0xffff ffff ffff ffff
fxor    $f4, $f4, $f4
punpcklbh $f2, $f4, $f0
```

V. 将 f4 置为 0x 00FF 00FF 00FF 00FF:

```
fxor    $f0, $f0, $f0 # 0x 0
pcmpeqw $f2, $f2, $f2 # 0x FFFF FFFF FFFF FFFF
punpcklbh $f4, $f2, $f0 # 0x 00FF 00FF 00FF 00FF
```

29. **pmovmskb - Move Byte Mask**

`pmovmskb DEST, SRC` (Packed Byte)

`SRC`, `DEST` 均为 64 位多媒体寄存器

操作：

```
DEST[0] = SRC[7]
DEST[1] = SRC[15]
DEST[2] = SRC[23]
DEST[3] = SRC[31]
DEST[4] = SRC[39]
DEST[5] = SRC[47]
DEST[6] = SRC[55]
DEST[7] = SRC[63]
```

```
DEST[63:8] = 0x0000 0000 0000 00
```

Example:

```
src : 0x0100 ffff 00ff aa55
```

```
pmovmskb dest, src 结果为: 0x0000 0000 0000 0036
```

30. **pextrh** - Extract Half-word

```
pextrh DEST, SRC2, SRC1
```

操作:

```
DEST[15:0] = (SRC2 >> (SRC1[1:0]*16))[15:0]
```

```
DEST[63:16] = 0x0000 0000 0000
```

Example:

```
src1 : 0x2
```

```
src2 : 0x0100 aa55 ffff 00ff
```

```
pextrh dest, src2, src1 结果为: 0x0000 0000 0000 aa55
```

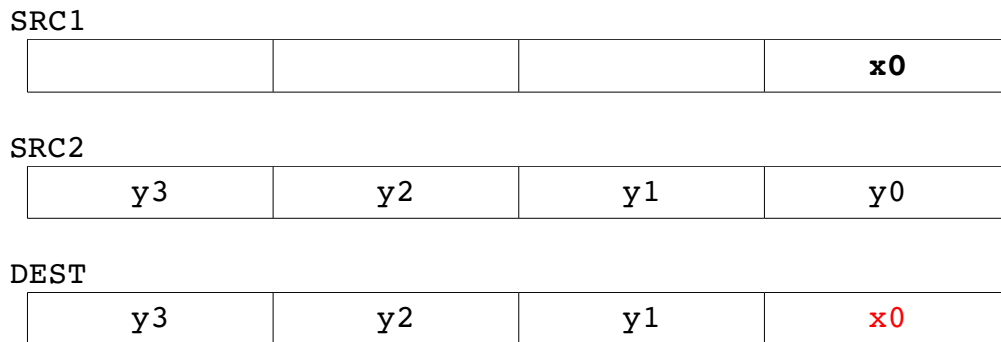
31. pinsrh_0/pinsrh_1/pinsrh_2/pinsrh_3 - Insert Half-word

```

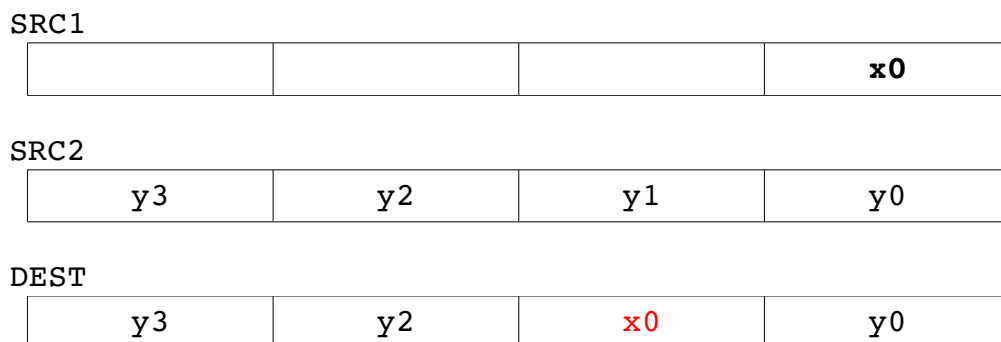
pinsrh_0 DEST, SRC2, SRC1    (Packed Half-word)
pinsrh_1 DEST, SRC2, SRC1    (Packed Half-word)
pinsrh_2 DEST, SRC2, SRC1    (Packed Half-word)
pinsrh_3 DEST, SRC2, SRC1    (Packed Half-word)

```

pinsrh_0 DEST, SRC2, SRC1 操作示意图:



pinsrh_1 DEST, SRC2, SRC1 操作示意图:



Example:

```
src1 : 0x55aa
```

```
src2 : 0x0100 ffff 00ff ff01
```

```
pinsrh_0 dest, src2, src1 结果为: 0x0100 ffff 00ff 55aa
```

```
pinsrh_1 dest, src2, src1 结果为: 0x0100 ffff 55aa ff01
```

```
pinsrh_2 dest, src2, src1 结果为: 0x0100 55aa 00ff ff01
```

```
pinsrh_3 dest, src2, src1 结果为: 0x55aa ffff 00ff ff01
```

以下为龙芯自定义指令

32. pasubub

对无符号字节包数据做减法运算，结果取绝对值

```
pasubub DEST, SRC2, SRC1
```

操作：

```
DEST[7:0] = ABS( SRC2[7:0] - SRC1[7:0] )    -- 结果取绝对值
```

```
(*repeat operation for 2nd through 7th bytes in SRC1 and SRC2)
```

```
DEST[63:56] = ABS( SRC2[63:56] - SRC1[63:56] )
```

Example:

```
src1 : 0x0002 8000 0003 0001
```

```
src2 : 0x0102 4000 0002 0002
```

```
pasubub dest, src2, src1 结果为: 0100 4000 0001 0001
```

33. **biadd**

```
biadd DEST, SRC
```

操作:

```
TEMP = SRC[63:56] + .. + SRC[23:16] + SRC[15:8] + SRC[7:0]
DEST[15:0] = TEMP
DEST[63:16] = 0x0000 0000 0000 00
```

Example:

```
src : 0x0000 4810 1020 0008
biadd dest, src 结果为: 0x0000 0000 0000 0090
```

```
src : 0x00c0 8000 0000 0004
biadd dest, src 结果为: 0x0000 0000 0000 0144
```

```
src : 0x0100 ffff 00ff aa55
biadd dest, src 结果为: 0x0000 0000 0000 03fd
```

```
src : 0xffff ffff ffff ffff
biadd dest, src 结果为: 0x0000 0000 0000 07f8 (0xff * 8 = 0x7f8)
```

可以看到该指令运算结果的最大值为 0x7f8

注: 一般 pasubub 与 biadd 合用, 等同于 x86 的 psadbh:

```
pasubub $f0, $f2, $f4
biadd $f6, $f0
```

以下指令严格意义上来说，不能称作多媒体指令，只是为了方便操作位于多媒体寄存器中的数据而引入。实际上和操作通用寄存器中的数据指令相似。

34. **for**

按位与

```
for DEST, SRC2, SRC1
```

35. **fxor**

按位异或

```
fxor DEST, SRC2, SRC1
```

36. **fnor**

按位或非

```
fnor DEST, SRC2, SRC1
```

37. **fand**

按位与

```
fand DEST, SRC2, SRC1
```

Example:

```
src1 : 0x8102 fff0 00fd ff04
```

```
src2 : 0x0100 ffff 00ff ff01
```

```
for dest, src2, src1 结果为: 0x8102 ffff 00ff ff05
```

```
fxor dest, src2, src1 结果为: 0x8002 000f 0002 0005
```

```
fnor dest, src2, src1 结果为: 0x7efd 0000 ff00 00fa
```

```
fand dest, src2, src1 结果为: 0x0100 fff0 00fd ff00
```

38. fsrl

字数据逻辑右移，高位字填充 0

```
fsrl DEST, SRC2, SRC1
```

Example:

```
src1 : 0x0100 ffff 85ff ff01  
src3 : 0x8
```

```
fsrl dest, src1, src3 结果为: 0x0000 0000 0085 ffff
```

39. fdsrl

双字逻辑右移

```
fdsrl DEST, SRC2, SRC1
```

Example:

```
src2 : 0x8102 0ff0 00fd ff04  
src3 : 0x8
```

```
fdsrl dest, src2, src3 结果为: 0x0081 020f f000 fdff
```

40. **fsra**

低位字数据算术右移，高位字填充低位字的最高位(符号位)

```
fsra DEST, SRC2, SRC1
```

Example:

```
src1 : 0x0100 ffff 85ff ff01  
src3 : 0x8
```

```
fsra dest, src1, src3 结果为: 0xffff ffff ff85 ffff
```

```
若 src1 : 0x0100 ffff 75ff ff01  
则 fsra dest, src1, src3 结果为: 0x0000 0000 0075 ffff
```

41. **fdsra**

双字算术右移

```
fdsra DEST, SRC2, SRC1
```

Example:

```
src1 : 0x0100 ffff 85ff ff01  
src3 : 0x8
```

```
fdsra dest, src2, src3 结果为: 0xff81 020f f000 fdff
```

42. **fsll**

低位字数据逻辑左移，高位字填充低位字移位后的最高位

```
fsll DEST, SRC2, SRC1
```

Example:

```
src1 : 0x0100 ffff 754f ff01  
src2 : 0x8102 0ff0 00a7 ff04  
src3 : 0x8
```

```
fsll dest, src1, src3 结果为: 0x0000 0000 4fff 0100  
fsll dest, src2, src3 结果为: 0xffff ffff a7ff 0400
```

若 src3 : 0xc, 则:

```
fsll dest, src1, src3 结果为: 0xffff ffff fff0 1000  
fsll dest, src2, src3 结果为: 0x0000 0000 7ff0 4000
```

43. **fdsll**

双字逻辑左移

```
fdsll DEST, SRC2, SRC1
```

Example:

```
src1 : 0x0100 ffff 75af ff01  
src2 : 0x8102 0ff0 00ad ff04  
src3 : 0x8
```

```
fdsll dest, src1, src3 结果为: 0x00ff ff75 afff 0100  
fdsll dest, src2, src3 结果为: 0x020f f000 adff 0400
```

44. **fadd**

低位(31:0)字数据加

```
fadd DEST, SRC2, SRC1
```

指令有使用要特别注意, 见下页 **Example** 的描述

操作:

```
DEST[31:0] = SRC2[31:0] + SRC1[31:0]
```

45. **faddu**

无符号低位字数据加, 高位字填充相加结果的最高位

```
faddu DEST, SRC2, SRC1
```

操作:

```
DEST[31:0] = SRC2[31:0] + SRC1[31:0]  
DEST[63:32] = DEST[31] ... DEST[31] # 高位字填充相加结果的最高位
```

46. **fdadd**

双字数据加

```
fdadd DEST, SRC2, SRC1
```

操作:

```
DEST[63:0] = SRC2[63:0] - SRC1[63:0]
```

Example:

```

src1 : 0x0000 00ff ffff fffe
src2 : 0x0000 0001 0000 0004

fadd dest, src2, src1 结果为: 0x0000 0000 0000 0002
faddu dest, src2, src1 结果为: 0x0000 0000 0000 0002 # 溢出
fdadd dest, src2, src1 结果为: 0x0000 0101 0000 0002

-----

src1 : 0x0000 0000 8fff fffe
src2 : 0x0000 0000 0000 0001

faddu dest, src2, src1 结果为: 0xffff ffff 8fff ffff
fdadd dest, src2, src1 结果为: 0x0000 0000 8fff ffff

```

注意:

若 `src1 = 0x0000 0000 8fff fffe`, `src2 = 0x0000 0000 0000 0001`
fadd dest, src2, src1 则出现 **Floating point exception**, 进反复测试
 只要运算结果为负就会出现该异常, 按理应该高位字填充相加结果的最高位才是。

测试程序参见附录 C.

这个是体系结构的一个约定了。`fadd` 只对低 32 位执行加运算, 然而操作数是位于 64 位多媒体寄存器 (MMR) 中的。如果要使用 `fadd` 的话, 则寄存器中的数据必须符号扩展, 否则程序会异常退出。这个只对运行于 32 位模式下的龙芯 2E 而言, 64 位模式下的龙芯 2E 就不会有该问题。

如上面的测试程序中操作数 `0x0000 0000 ffff fffe(-2)` 要扩展为 `0xffff ffff ffff fffe` 方可无事。

47. fsub

有符号低位字数据减

```
fsub DEST, SRC2, SRC1
```

操作:

```
DEST[31:0] = SRC2[31:0] - SRC1[31:0]  
DEST[63:32] = DEST[31] ... DEST[31] # 高位字填充相减结果的最高位
```

48. fsubu

无符号低位字数据减

```
fsubu DEST, SRC2, SRC1
```

操作:

```
DEST[31:0] = SRC2[31:0] - SRC1[31:0]  
DEST[63:32] = DEST[31] ... DEST[31] # 高位字填充相减结果的最高位
```

49. fdsb

有符号双字数据减

```
fdsb DEST, SRC2, SRC1
```

操作:

```
DEST[63:0] = SRC2[63:0] - SRC1[63:0]
```

Example:

```
src1 : 0x0000 0000 0000 0002
src2 : 0x0000 0000 ffff fffe
```

```
fsub dest, src2, src1 结果为: 0xffff ffff ffff fffc
fsubu dest, src2, src1 结果为: 0xffff ffff ffff fffc
fddsub dest, src2, src1 结果为: 0x0000 0000 ffff fffc
```

```
src1 : 0x0000 0000 ffff fffe
src2 : 0x0000 0000 0000 0001
```

```
fsub dest, src2, src1 结果为: 0x0000 0000 0000 0003
fsubu dest, src2, src1 结果为: 0x0000 0000 0000 0003
fddsub dest, src2, src1 结果为: 0xffff ffff 0000 0003
```

50. fseq

相等则置浮点控制寄存器的条件位(CC0, FCR31[23]) 为 1, 否则置条件位为 0

```
fseq DEST, SRC
```

操作:

```
IF DEST[63:0] == SRC[63:0] THEN
    FCR31[23] = 1
ELSE
    FCR31[23] = 0
FI
```

Example:

```
src : 0x0000 0202 0201 8040
dest : 0x0000 0000 0201 8040

fseq dest, src FCR31 is: 0x0000 0000

-----

src : 0x0000 0000 0201 8040
dest : 0x0000 0000 0201 8040

fseq dest, src FCR31 is: 0x0080 0000
```

51. fseq1 (后缀为数字 1)

```
fseq1 DEST, SRC
```

简单测试了一下，该指令与 `fseq` 没有明显的差别，发现的兄弟提醒一下我。

52. fsltu

无符号数据比较，小于则置浮点控制寄存器的条件位(CC0, FCR31[23]) 为 1，否则置条件位为 0

```
fsltu DEST, SRC
```

操作：

```
IF DEST[63:0] < SRC[63:0]      # 无符号数据比较
    FCR31[23] = 1;
ELSE
    FCR31[23] = 0;
```

53. fslt

有符号数据比较，小于则置浮点控制寄存器的条件位(CC0, FCR31[23]) 为 1，否则置条件位为 0

```
fslt DEST, SRC
```

操作：

```
IF DEST[63:0] < SRC[63:0]      # 有符号数据比较
    FCR31[23] = 1;
ELSE
    FCR31[23] = 0;
```

Example:

```
src : 0x0000 0000 0000 8042
```

```
dest : 0x8000 0000 0000 8040 # 有符号为负数, 无符号为正数
```

```
fslt dest, src FCR31 is: 0x0080 0000 # 有符号比较 dest 小, 置 1
```

```
fsltu dest, src FCR31 is: 0x0000 0000 # 无符号比较 dest 大, 置 0
```

54. fsleu

无符号数据比较，小于等于则置浮点控制寄存器的条件位(CC0, FCR31[23]) 为 1，否则置条件位为 0

```
fsleu DEST, SRC
```

操作：

```
IF DEST[63:0] <= SRC[63:0]      # 无符号数据比较
    FCR31[23] = 1;
ELSE
    FCR31[23] = 0;
```

55. fsle

```
fsle DEST, SRC
```

有符号数据比较，小于等于则置浮点控制寄存器的条件位(CC0, FCR31[23]) 为 1，否则置条件位为 0，

操作：

```
IF DEST[63:0] < SRC[63:0]      # 有符号数据比较
    FCR31[23] = 1;
ELSE
    FCR31[23] = 0;
```

Example:

```
src : 0x0000 0000 0000 8042
dest : 0x8000 0000 0000 8040
```

```
fsle dest, src FCR31 is: 0x0080 0000 # 有符号比较 dest 小, 置1
fsleu dest, src FCR31 is: 0x0000 0000 # 无符号比较 dest 大, 置0
```

```
src : 0x8000 0000 0000 8042
dest : 0x8000 0000 0000 8042
```

```
fsle dest, src FCR31 is: 0x0080 0000 # 有符号比较,相等, 置1
fsleu dest, src FCR31 is: 0x0080 0000 # 无符号比较,相等, 置1
```

```
src : 0x0000 0000 0000 8042
dest : 0x0000 0000 0000 8042
```

```
fsle dest, src FCR31 is: 0x0080 0000
fsleu dest, src FCR31 is: 0x0080 0000
```

Reference :

- [1] Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2B, Intel Corp, 2006
- [2] 梁肇新, 编程高手箴言 [M]. 北京: 电子工业出版社, 2003: 368~393.
- [3] 龙芯 2E 用户手册 [S].
- [4] Khang Nguyen, Assembly Language Tips & Tricks for the Intel Pentium 4 Processor, Intel Corp.

附录 A: 测试例程 -- pmaddhw

```
#include <stdio.h>

unsigned short data[] = {
    0x0, 0x0, 0x0, 0x0,      /* 该 64 位用来保存运算结果 */
    0x1, 0x3, 0x5, 0x7,      /* src1 = 0x0007 0005 0003 0001 */
    0x2, 0x4, 0x6, 0x8      /* src2 = 0x0008 0006 0004 0002 */
};

inline void pmaddhw(unsigned short * const row)
{
    asm volatile
    (
        ".set mips3\n\t"
        ".set noreorder\n\t"
        "ldc1 $f0, %1\n\t"      /* 从内存加载 64 位数据到 f0 */
        "ldc1 $f2, %2\n\t"      /* f2 <-- SRC2 */
        "pmaddhw $f2, $f2, $f0\n\t" /* pmaddhw dest, src2, src1 */
        "sdc1 $f2, %0\n\t"      /* 存储计算结果 */
        ".set reorder\n\t"
        ".set mips0\n\t"
        : "=m"(*row)
        : "m"(*(row+4)), "m"(*(row+8))
        : "$f0", "$f2", "memory"
    );
}
```

```
int main()
{
    printf("src1: 0x%04x %04x %04x %04x\n\n", data[7], data[6], data[5],
data[4]);
    printf("src2: 0x%04x %04x %04x %04x\n", data[11], data[10], data[9],
data[8]);
    pmaddhw(data);
    printf("pmaddhw dest, src2, src1 结果为: 0x%04x %04x %04x %04x\n\n",
data[3], data[2], data[1], data[0]);
}
```

附录 B: 测试例程 -- **fslt/fsltu**

```
#include <stdio.h>

unsigned short data[] = {
    0x0, 0x0, 0x0, 0x0,
    0x8042, 0x0000, 0x0000, 0x0000,
    0x8040, 0x0000, 0x0000, 0x8000
};

void fslt(unsigned short * const row);

int main()
{
    printf("-----\n");
    printf(" Testing Godson2 Multi-Media Instruction  fslt/fsltu\n");
    printf("-----\n\n");
    printf("src : 0x%04x %04x %04x %04x\n", data[7], data[6], data[5],
data[4]);
    printf("dest : 0x%04x %04x %04x %04x\n\n", data[11], data[10], data[9],
data[8]);

    fslt(data);

    printf("fslt dest, src FCR31 is: 0x%04x %04x\n\n", data[1], data[0]);

    printf("fsltu dest, src FCR31 is: 0x%04x %04x\n\n", data[3], data[2]);
}
```

```
void fslt(unsigned short * const row)
{
    asm volatile
    (
        ".set mips3\n\t"
        ".set noreorder\n\t"

        "ldc1 $f0, %2\n\t"
        "ldc1 $f2, %3\n\t"

        "fslt $f2, $f0\n\t"

        "xor $13, $13, $13\n\t"
        "cfc1 $13, $31\n\t"          /* 取浮点控制寄存器的值到通用寄存器 */
        "sw $13, %0\n\t"          /* 将通用寄存器的值写到内存, 用于显示 */

        "fsltu $f2, $f0\n\t"

        "xor $13, $13, $13\n\t"
        "cfc1 $13, $31\n\t"
        "sw $13, %1\n\t"

        ".set reorder\n\t"
        ".set mips0\n\t"
        : "=m"(*row), "=m"(*(row+2))
        : "m"(*(row+4)), "m"(*(row+8))
        : "$f0", "$f2", "$f4", "memory"
    );
}
```

附录 C: 测试例程 -- **fadd**

```
#include <stdio.h>

unsigned short data[] = {
    0x0, 0x0, 0x0, 0x0,
    0xFFFE, 0xFFFF, 0x0000, 0x0000,
    0x0001, 0x0000, 0x0000, 0x0000
};

inline void fadd(unsigned short * const row)
{
    asm volatile
    (
        ".set mips3\n\t"
        ".set noreorder\n\t"
        "ldc1 $f2, %1\n\t"
        "ldc1 $f8, %2\n\t"

        "fadd $f8, $f8, $f2\n\t"

        "sdc1 $f8, %0\n\t"

        ".set reorder\n\t"
        ".set mips0\n\t"
        : "=m"(*row)
        : "m"(*(row+4)), "m"(*(row+8))
        : "$f2", "$f4", "$8", "memory"
    );
}
```

```
int main()
{
    printf("-----\n");
    printf("  Testing Godson2 Instruction fadd\n");
    printf("-----\n\n");
    printf("src1 is: 0x%04x %04x %04x %04x\n", data[7], data[6], data[5],
data[4]);
    printf("src2 is: 0x%04x %04x %04x %04x\n", data[11], data[10], data[9],
data[8]);

    fadd(data);

    printf("fadd dest, src2, src1 result is: 0x%04x %04x %04x %04x\n",
data[3], data[2], data[1], data[0]);
}
```

预期做 `0x0000 0000 ffff fffe(-2)` 与 `0x0000 0000 0000 0001(+1)` 的 `fadd` 运算, 预期输出应该为 `0xffff ffff ffff ffff`, 但运行会报 `Floating point exception`, 并退出。