SPARC Architecture Essentials

Version: 0.4
Date: 2009-05-27
Author: Jack Tan <jiankemeng@gmail.com>

版本历史

版本状态	作 者	参与者	起止日期	备注
0.2	Jack Tan		09-05-15	完成草稿
0.3	Jack Tan		09-05-20	重排版
0.4	Jack Tan		09-05-27	增加中断异常部分

1. GPR 和 ABI

1.1 GPR

1.1.0 SPARC V9 为 64 位，兼容 32 位；SPARC V8 也是 64 位兼容 32 位；而广泛应用于航天器上的 SPARC V7 为 32 位，很精简，天上飞不是闹着玩的，简单、可靠才是王道

1.1.1 SPARC 的 GPR (General Purpose Register) 较特殊，其引入寄存器窗口 (Register Windows) 的概念，每个窗口的 GPR 数为 24（可以有 3 ~ 32 个窗口，任一时刻只有一个窗口可见），再加上一组（8 个）全局 GPR （Global GPR，有多组，任一时刻仅一组可见，V9 有 2 组，UltraSPARC 则有 2 ~ 15 组），则任一时刻可见的 GPR 数为 32

可以看到 SPARC 的 GPR 有很多，引入的目的意在提高过程调用的效率。

一般来讲，进入一个子函数首先需要借助栈来保存一些寄存器，而后返回时需要恢复之，参数的保存和恢复需要很多访存操作，这个会增加很多延迟。若 CPU 内有很多组冗余的寄存器，则子函数调用时，直接将当前窗口指针指向下一个，子函数就直接使用新的一组寄存器，就无需一堆的访存指令了。这个思想很好，进入 实用还得有些细节问题要考虑：

I. 子函数和父函数间参数和返回值怎么传递？
II. 由于物理的限制，可用寄存器组不可能无限大，当调用链达到一定的深度后，寄存器组不够用怎么办？

1.1.2 任一时刻可见之 32 个 GPR 编号

24 个窗口寄存器编号为 r8 ~ r31，8 个一组。

r8 ~ r15 又名为 o0 ~ o7，为输出组 (out)；
r16 ~ r23 又名为 l0 ~ l7，为本地组 (Local)；
r24 ~ r31 又名为 i0 ~ i7，为输入组 (In)。
8 个全局寄存器编号为 r0 ~ r7 又名为 g0 ~ g7

1.2 ABI

SPARC 的 ABI 没那么多版本，这个是程序员的福音，psABI：

Type	Name			Usage
in		%i7	%r31	return address – 8 †
	%fp,	%i6	%r30	frame pointer †
		%i5	%r29	incoming param 5 †
		%i4	%r28	incoming param 4 †
		%i3	%r27	incoming param 3 †
		%i2	%r26	incoming param 2 †
		%i1	%r25	incoming param 1 †
		%i0	%r24	incoming param 0, † outgoing return value
local		%l7	%r23	local 7 †
		%l6	%r22	local 6 †
		%l5	%r21	local 5 †
		%l4	%r20	local 4 †
		%l3	%r19	local 3 †
		%l2	%r18	local 2 †
		%l1	%r17	local 1 †
		%l0	%r16	local 0 †
out		%o7	%r15	address of call instruction, ‡ temporary value
	%sp,	%o6	%r14	stack pointer †
		%o5	%r13	outgoing param 5 ‡
		%o4	%r12	outgoing param 4 ‡
		%o3	%r11	outgoing param 3 ‡
		%o2	%r10	outgoing param 2 ‡
		%o1	%r9	outgoing param 1 ‡
		%o0	%r8	outgoing param 0, ‡ incoming return value

r24 ~ r29 用作前 6 个输入参数，同时 r24 (i0) 又用作返回值

r30 用作帧指针 fp

r31 置 （函数返回地址 - 8）

r8 ~ r13 用作 6 个输出参数，同时 r8 (o0) 又作第一个输出参数

r14 用作栈指针 sp

r15 置 CALL 指令的地址

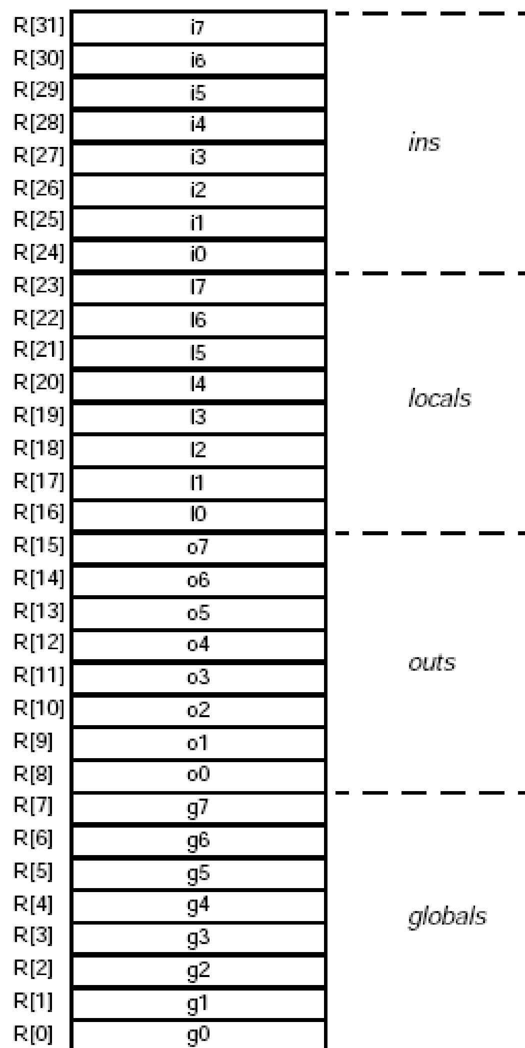
Type	Name		Usage
	%g7	%r7	global 7 (reserved for system)
	%g6	%r6	global 6 (reserved for system)
	%g5	%r5	global 5 (reserved for system)
	%g4	%r4	global 4 (reserved for application)
	%g3	%r3	global 3 (reserved for application)
	%g2	%r2	global 2 (reserved for application)
	%g1	%r1	global 1 ‡
global	%g0	%r0	0
		%f31	floating-point value ‡
floating-point		. . .	. . .
		%f0	floating-point value, ‡ floating-point return value
special		%y	Y register ‡

要注意的是与 MIPS 一样 r0 始终为 0

2. 窗口结构

2.1 概述

任一时刻 SPARC 的 GPR “视图” 为：



其中 r8 ~ r31 为一个窗口，实际实现时 V9 规定需有 3 ~ 32 个窗口。

引入多窗口的初衷主要是为了尽可能规避函数调用时保存用户上下文的存储器访问。

为了平滑父子函数间的参数传递和返回值传递，SPARC 将多个窗口组织为：

Author: Jack Tan <jiankemeng@gmail.com>

Window (CWP – 1)
R[31] ins R[24]
R[23] locals R[16]
R[15] outs R[8]

Window (CWP)
R[31] ins R[24]
R[23] locals R[16]
R[15] outs R[8]

Window (CWP + 1)
R[31] ins R[24]
R[23] locals R[16]
R[15] outs R[8]

R[7] globals R[1]
R[0] 0
63 0

相邻窗口间 8 个寄存器重叠，即：

当前窗口的 o0~ o7 和下一个窗口的 i0 ~ i7 对应于同一组物理寄存器；
当前窗口的 i0 ~ i7 和上一窗口的 o0 ~ o7 也是对应于同一组物理寄存器；
所有窗口的 local 寄存器绝对独立，这也是其取名为 local 的原因 :)

结合 ABI 粗略来看，就是当前的输入参数来自于上一个窗口的输出参数，当前的输出参数即是下一个窗口的输入参数！

另外需要特别留意的是，最后一个窗口的输出寄存器组 (o0 ~ o7) 与第一个窗口的输入寄存器组 (i0 ~ i7) 也是重叠的，这样整个寄存器窗口看起来就是一个圆。

则对于 SPARC V9，其 GPR 的数目范围为：

64 ~ 528
(3 x 16 + (8 + 8)) <= N_GPR <= (32 x 16 + (8 + 8))
窗口范围为 3 ~ 32，相邻的窗口 8 个重叠，则实际窗口内的“独立”寄存器为 16，加上 16 个全局寄存器即得。

而对于 UltraSPARC 2007 其 GPR 数目的范围为：

72 ~ 640
(3 x 16 + (8 x 3)) <= N_GPR <= (32 x 16 + (8 x 16))，只因其全局寄存器组数的范围为 3 ~ 16

2.2 窗口管理初步

为了支持窗口的管理，SPARC 引入了 5 个特权寄存器，一组指令和几个异常来管理窗口。

其中 CWP (Current Window Pointer) 寄存器指定当前使用的窗口（实际置窗口的编号，从 0 开始），则正常情形下，函数调用时都需转动 CWP，以获取一个新的窗口，则在设计 'CALL' 指令（用于支持函数调用）时，理应让其内含 CWP += 1 操作，一如用于函数返回的 'RETURN' 指令内含 CWP -= 1 操作一样，但 SPARC 设计时，为了支持一些特殊的叶函数能直接使用父函数的窗口，尽可能降低开销，其让所有用于函数调用的指令 'CALL' 和 'JMPL' 都不去影响 CWP，而要子函数自己负责，则非叶子函数的第一条指令就是用'SAVE' 转动 CWP 同时分配栈空间（更新 sp 的值）

可以想见，当调用链达到一定的深度后，窗口肯定会不够用，这个时候，处理器就会抛出一个异常，由 OS 来负责保存窗口到栈并释放窗口，关于这个层面的窗口管理，留待以后讨论吧。

3. 过程调用

先看一个简单的 C 函数调用，在 ISA 层面上怎么被支持：

```
int test_call(int a, int b, int c)
{
    a = b + c;
    return a;
}

int main()
{
    int d;
    d = test_call(1, 2, 3);
    return d;
}
```

对应的汇编码为：

注：SPARC 汇编的操作习惯是从左到右，即：左边是源寄存器，右边是目标寄存器，这个和 MIPS 相反，与 x86 AT&T 语法相似。

```
0000000000000000 <test_call>:
   0:   9d e3 bf 40     save %sp, -192, %sp ---> 将父窗口的 sp 减 192 后，置入当前窗口的 sp
   4:   82 10 00 18     mov %i0, %g1        ---> 保存 a (param1) 入 g1
   8:   84 10 00 19     mov %i1, %g2        ---> b (param2) 入 g2
   c:   86 10 00 1a     mov %i2, %g3        ---> c (param3) 入 g3
  10:   c2 27 a8 7f     st %g1, [ %fp + 0x87f ]  ---> 将 g1, g2, g3 保存于栈后又将其读出
  14:   c4 27 a8 87     st %g2, [ %fp + 0x887 ]       此代码生成时没有加 -O 优化
  18:   c6 27 a8 8f     st %g3, [ %fp + 0x88f ]
  1c:   c4 07 a8 87     ld [ %fp + 0x887 ], %g2
  20:   c2 07 a8 8f     ld [ %fp + 0x88f ], %g1
```

24: 82 00 80 01 add %g2, %g1, %g1 ---> g1 = g2 + g1

28: c2 27 a8 7f st %g1, [%fp + 0x87f] ---> stw %g1, [%fp + 0x87f]，g1 保存于栈

2c: c2 07 a8 7f ld [%fp + 0x87f], %g1 ---> lduw [%fp + 0x87f], %g1，又将刚保存的读入 g1，很傻吧，实际这是没有优化的代码，gcc 固定生成，逻辑清晰

30: 83 38 60 00 sra %g1, 0, %g1 ---> 算术右移 0 位

34: b0 10 00 01 mov %g1, **%i0** ---> 将返回值入 i0，return 后(CWP - 1) 就是前一个窗口的 o0

38: 81 cf e0 08 rett %i7 + 8 <==> return %i7 + 8

3c: 01 00 00 00 nop

save ---> 从前一个窗口取值，CWP += 1 后，向当前窗口写值，其是承担转动窗口的大任的 :)

return %i7 + 8 ---> 将 CWP -= 1 后跳转到 %i7 + 8 处，其是承担恢复窗口的大任的 :)

stw ---> 向内存写入一个 word (32-bit)

lduw ---> 从内存加载一个无符号 word

以上 st, ld, rett 皆为汇编助记符号，在上面的环境下分别对应于 stw, lduw, return

0000000000000040 <main>:

40: 9d e3 bf 30 save %sp, -208, %sp

44: 90 10 20 01 mov 1, %o0 ---> 依次置 3 个参数 a, b, c 入 o0, o1, o2

48: 92 10 20 02 mov 2, %o1 在 save 后 (CWP + 1) 其就变为 i0, i1, i2 （名字变了，但依然是同样的物理寄存器）

4c: 94 10 20 03 mov 3, %o2

50: 40 00 00 00 call 0 <test_call> ---> 调用 test_call()

54: 01 00 00 00 nop

58: 82 10 00 08 mov **%o0**, %g1 ---> 取返回值入 g1

5c: c2 27 a7 eb st %g1, [%fp + 0x7eb]

60: c2 07 a7 eb ld [%fp + 0x7eb], %g1

64: 83 38 60 00 sra %g1, 0, %g1

68: b0 10 00 01 mov %g1, %i0 ---> 将返回值入 i0，对应于调用者的 o0

6c: 81 cf e0 08 rett %i7 + 8

70: 01 00 00 00 nop

指令 call 在将其自身所在之地址写入 o7 后，就直接跳转目标函数地址处，要留意的是 call 并不会改变 CWP 的值，即不会转动窗口，似乎和 return 不对称

a. 参数传递

父函数写入 o0 ~ o5，依次对应前 5 个参数
子函数使用 i0 ~ i5 接受之

b. 返回值

子函数写入 i0
父函数用 o0 接受之

c. 指令支持

call
call label
两个操作：自身所在之地址写入 o7；跳转到 label 处

return
return %i7 + 8
return %i7 + %g1
两个操作：CWP -= 1；跳转

call 与 return 后皆有一个延迟槽 (delay slot)

save

save r1, r2, rd
save r1, simm13, rd
三个操作：从前一个窗口 sp (o6) 取值；CWP += 1；向当前窗口 sp (o6) 写值

以上指令已足以支持过程调用，为有更大的灵活性，SPARC 又引入了 Jump and Link (jmpl) 指令：

jmpl

jmpl r1+r2, rd
两个操作：将自身所在之地址写入 rd；跳转到 address 处。如：

```
jmpl    %l2+%l3, %o7
jmpl    %i7+8, %g0      <=> 汇编助记符为 ret    ----> 用于非叶函数的返回
jmpl    %o7+8, %g0      <=> 汇编助记符为 retl   ----> 用于叶函数的返回
```

d. 宏观上看返回地址的精巧安排

call 将自身所在之地址写入 o7，则其延迟槽后的指令为子函数返回后的第一条指令，则该指令所在地址为 o7 + 8（o7 + 4 为延迟槽）

子函数来看，i7 （对应于父函数 o7）加 8 即为返回地址

此即为 ABI 规定 i7 为 (return address - 8) 的原因

e. 宏观上来看 **sp, fp** 的转变

fp (i6), sp (o6)

父函数之 sp (o6) 在进入子函数 save 后，父函数之 sp 值不变，但却变成子函数之 fp，只要子函数不改变 fp 的值，则返回时 sp 的恢复都省了。
子函数之 sp 则直接来之父函数之 sp 减去一个常数（分配局部变量）

实在佩服 SPARC 的这个设计，太 COOL 了

4. 判断循环

先看一些基本的 C 语言判断循环结构在 ISA 层面上怎么被支持：

4.1 判断

```
int test_if(int s)
{
    int i;
    if(s > 0)
        i = s;
    else if(s < 0)
        i = -s;
    else
        i = s * 8;

    return i;
}
```

```
000000000000000c <test_if>:
  c:   80 a2 20 00     cmp %o0, 0        ---> %o0 与 0 比较
10:   14 48 00 06     bg %icc, 28 <test_if+0x1c>        ---> 大于 0 则跳转；annul 位为 0，则延迟槽中的指令总是被执行
14:   82 10 00 08     mov %o0, %g1    ---> 位于延迟槽，o0 值拷贝入 g1
18:   84 20 00 08     neg %o0, %g2    ---> o0 取反，对应于 -s
1c:   83 2a 20 03     sll %o0, 3, %g1    ---> o0 逻辑左移 3 位，其对应于 s*8
20:   80 a2 20 00     cmp %o0, 0    ---> 重新比较，据结果置 CCR[icc] 的相应位
24:   83 64 c0 02     movl %icc, %g2, %g1    ---> 小于 0 则将 g2 入 g1
28:   81 c3 e0 08     retl   ---> 叶函数返回，不改变 CWP，等价于 jmpl %o7+8, %g0
2c:   91 38 60 00     sra %g1, 0, %o0    ---> 延迟槽，总是被执行
```

可以看到：

大于 0 时 g1 置 o0 就直接返回了；

小于等于 0 时，g1 置 s*8，g2 置 -s；
后条件拷贝，小于 0 则将 g2 入 g1 作为返回值 (-s)，等于 0 就以 g1(s*8) 为返回值

注意到 test_if 是一个叶函数（其不调用其他函数），作为一个优化，其无需使用 save 指令去新获取一个独立的窗口，小心复用父函数的即可。

4.2 循环

```
int test_cyc1(int c)
{
   int sum = 0;
   do {
      sum += c;
      c--;
   } while(c > 0);

   return sum;
}
```

```
0000000000000030 <test_cyc1>:
30:   84 10 20 00     clr %g2
34:   84 00 80 08     add %g2, %o0, %g2    ---> sum += c
38:   82 02 3f ff     add %o0, -1, %g1    ---> g1 = o0 - 1
3c:   91 38 60 00     sra %g1, 0, %o0     ---> g1 拷贝入 o0
40:   80 a2 20 00     cmp %o0, 0
44:   34 4f ff fd     bg,a   %icc, 38 <test_cyc1+0x8>    ---> o0 > 0 则跳转，annul 位为 1，则延迟槽中的指令只在跳转发生时才会被执行
48:   84 00 80 08     add %g2, %o0, %g2    ---> 延迟槽，跳转时被执行
4c:   81 c3 e0 08     retl ---> 叶函数返回，不改变 CWP，等价于 jmpl %o7+8, %g0
50:   91 38 a0 00     sra %g2, 0, %o0    ---> 置返回值
```

跳转类指令，附加 ',a' 则是将 annul 位置 1

对条件跳转类指令，annul 位为 1，则 CPU 在执行时只在跳转发生时才会执行延迟槽中的指令

```
int test_cyc2(int c)
{
    int sum = 0;
    for(; c > 0; c--)
        sum += c;

    return sum;
}
```

```
000000000000004c <test_cyc2>:
54:   80 a2 20 00     cmp %o0, 0
58:   04 40 00 08     ble,pn   %icc, 78 <test_cyc2+0x24>   ---> annul 位为 0，则延迟槽中的指令总是被执行
5c:   84 10 20 00     clr %g2
60:   84 00 80 08     add %g2, %o0, %g2
64:   82 02 3f ff     add %o0, -1, %g1
68:   91 38 60 00     sra %g1, 0, %o0
6c:   80 a2 20 00     cmp %o0, 0
70:   34 4f ff fd     bg,a   %icc, 64 <test_cyc2+0x10>
74:   84 00 80 08     add %g2, %o0, %g2
78:   81 c3 e0 08     retl
7c:   91 38 a0 00     sra %g2, 0, %o0
```

跳转类指令，附加 ',pn' 则是指令编码为静态转移预测为不发生；',pt' 则为静态转移预测为发生

A. 寄存器

SPARC 实现有条件寄存器 CCR (Condition Codes Register)
其长为 8 位，结构如下：

```
7       4 3       0
----------------------
|   xcc   |   icc   |
----------------------
```

xcc 用于存放 64 bit 计算结果的条件位
icc 用于存放 32 bit 计算结果的条件位

他们的结构一样： | N | Z | V | C |
四位，分别表示 Negative, Zero, Overflow, Carry (or borrow)
其中 Carry 位，只要计算结果小于 0，也要置位

B. 指令

Bicc ---> branch on CCR[icc]
BPcc ---> branch on CCR (including icc and xcc) with prediction
BPr ---> branch on Interger Register with prediction

BPcc 类指令只是在 Bicc 类的基础上加了静态转移预测

Bicc 类有：

BA/BN --- Branch Always/Never
BNE/BE/BG/BLE/BGE/BL/BGU/BLEU --- Branch on !=/=/>/<=/>=/</>, Unsigned/<=, Unsigned

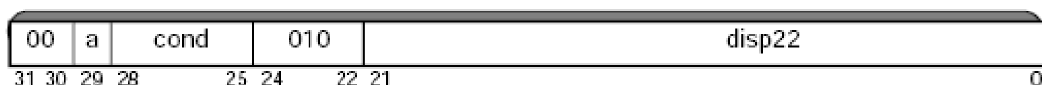
BCC/BCS --- Branch on Carry Clear (>=, unsigned)/Carry Set (<, unsigned)

BPOS/BNEG --- Branch on Positive/Negative

BVC/BVS --- Branch on Overflow Clear/Set

Opcode	cond	Operation	*icc* Test	Assembly Language Syntax	
BA	1000	Branch Always	1	ba{,a}	*label*
BN	0000	Branch Never	0	bn{,a}	*label*
BNE	1001	Branch on Not Equal	not Z	bne†{,a}	*label*
BE	0001	Branch on Equal	Z	be‡{,a}	*label*
BG	1010	Branch on Greater	not (Z or (N xor V))	bg{,a}	*label*
BLE	0010	Branch on Less or Equal	Z or (N xor V)	ble{,a}	*label*
BGE	1011	Branch on Greater or Equal	not (N xor V)	bge{,a}	*label*
BL	0011	Branch on Less	N xor V	bl{,a}	*label*
BGU	1100	Branch on Greater Unsigned	not (C or Z)	bgu{,a}	*label*
BLEU	0100	Branch on Less or Equal Unsigned	C or Z	bleu{,a}	*label*
BCC	1101	Branch on Carry Clear (Greater Than or Equal, Unsigned)	not C	bcc◊{,a}	*label*
BCS	0101	Branch on Carry Set (Less Than, Unsigned)	C	bcs∇{,a}	*label*
BPOS	1110	Branch on Positive	not N	bpos{,a}	*label*
BNEG	0110	Branch on Negative	N	bneg{,a}	*label*
BVC	1111	Branch on Overflow Clear	not V	bvc{,a}	*label*
BVS	0111	Branch on Overflow Set	V	bvs{,a}	*label*

† *synonym:* bnz ‡ *synonym:* bz ◊ *synonym:* bgeu ∇ *synonym:* blu



其中 BA/BN 为可分为非条件转移，annul 位的语义有些不同：

annul = 0, 延迟槽中的指令总是被执行

annul = 1, 延迟槽中的指令总是不被执行

其它属条件转移类指令，annul 位的语义为：

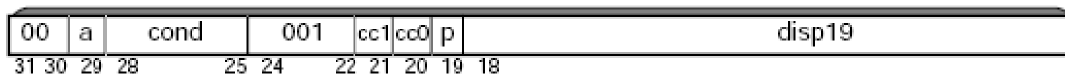
annul = 0, 延迟槽中的指令总是被执行

annul = 1, 延迟槽中的指令只在跳转发生时才被执行

BPcc 类指令有：

Instruction	cond	Operation	cc Test	Assembly Language Syntax	
BPA	1000	Branch Always	1	ba{,a}{,pt | ,pn}	*i_or_x_cc, label*
BPN	0000	Branch Never	0	bn{,a}{,pt | ,pn}	*i_or_x_cc, label*
BPNE	1001	Branch on Not Equal	**not** Z	bne†{,a}{,pt | ,pn}	*i_or_x_cc, label*
BPE	0001	Branch on Equal	Z	be‡{,a}{,pt | ,pn}	*i_or_x_cc, label*
BPG	1010	Branch on Greater	**not** (Z **or** (N **xor** V))	bg{,a}{,pt | ,pn}	*i_or_x_cc, label*
BPLE	0010	Branch on Less or Equal	Z **or** (N **xor** V)	ble{,a}{,pt | ,pn}	*i_or_x_cc, label*
BPGE	1011	Branch on Greater or Equal	**not** (N **xor** V)	bge{,a}{,pt | ,pn}	*i_or_x_cc, label*
BPL	0011	Branch on Less	N **xor** V	bl{,a}{,pt | ,pn}	*i_or_x_cc, label*
BPGU	1100	Branch on Greater Unsigned	**not** (C **or** Z)	bgu{,a}{,pt | ,pn}	*i_or_x_cc, label*
BPLEU	0100	Branch on Less or Equal Unsigned	C **or** Z	bleu{,a}{,pt | ,pn}	*i_or_x_cc, label*
BPCC	1101	Branch on Carry Clear (Greater than or Equal, Unsigned)	**not** C	bcc◊{,a}{,pt | ,pn}	*i_or_x_cc, label*
BPCS	0101	Branch on Carry Set (Less than, Unsigned)	C	bcs∇{,a}{,pt | ,pn}	*i_or_x_cc, label*
BPPOS	1110	Branch on Positive	**not** N	bpos{,a}{,pt | ,pn}	*i_or_x_cc, label*
BPNEG	0110	Branch on Negative	N	bneg{,a}{,pt | ,pn}	*i_or_x_cc, label*
BPVC	1111	Branch on Overflow Clear	**not** V	bvc{,a}{,pt | ,pn}	*i_or_x_cc, label*
BPVS	0111	Branch on Overflow Set	V	bvs{,a}{,pt | ,pn}	*i_or_x_cc, label*

† *synonym:* bnz ‡ *synonym:* bz ◊ *synonym:* bgeu ∇ *synonym:* blu



与 Bicc 类在，汇编语法上的差别在：

1. 可以带 ',pt'（静态预测为发生） 和 ',pn'（静态预测为不发生）后缀；不明确指定，汇编器默认使用 ',pt'
2. 可以指定使用的 CCR 域是 icc 还是 xcc

如：

```
bg,a,pt    %icc, label
```

BPr 类指令：

这类指令从通用寄存器中读取 Condition Code，而不是从 CCR 中取之

Instruction	rcond	Operation	Register Contents Test	Assembly Language Syntax
—	000	*Reserved*	—	
BRZ	001	Branch on Register Zero	R[rs1] = 0	brz {,a}{,pt | ,pn} reg_{rs1}, *label*
BRLEZ	010	Branch on Register Less Than or Equal to Zero	R[rs1] ≤ 0	brlez {,a}{,pt | ,pn} reg_{rs1}, *label*
BRLZ	011	Branch on Register Less Than Zero	R[rs1] < 0	brlz {,a}{,pt | ,pn} reg_{rs1}, *label*
—	100	*Reserved*	—	
BRNZ	101	Branch on Register Not Zero	R[rs1] ≠ 0	brnz {,a}{,pt | ,pn} reg_{rs1}, *label*
BRGZ	110	Branch on Register Greater Than Zero	R[rs1] > 0	brgz {,a}{,pt | ,pn} reg_{rs1}, *label*
BRGEZ	111	Branch on Register Greater Than or Equal to Zero	R[rs1] ≥ 0	brgez {,a}{,pt | ,pn} reg_{rs1}, *label*

其亦带 ',a' ',pt' ',pn' 后缀。不明确指定，汇编器默认使用 ',pt'

4. 中断异常 (Traps)

SPARC 的文档里将异常和中断统称为 Trap。一个 Trap 是指对异常或中断的响应行为。

4.1 Trap 概述

SPARC 有 3 个运行级别：用户态，特权级 (Privileged)，超级特权级 (Hyperprivileged)。

OS 运行于特权级。为支持系统虚拟化，SUN 在 2005 年修订了 SPARC 的 Specification 引入了超级特权级 (Hyperprivileged) 用于运行 Hypervisor

SPARC 下指定处理器状态的两个关键寄存器是 PSTATE 和 HPSTATE，分别对应于 Privileged 和 Hyperprivileged

为了良好地支持 Trap 嵌套，SPARC 引入了一个 Trap Level 寄存器，用于指定当前 Trap 的深度，正常运行的处理器 TL = 0， 处理器进入一个 Trap，TL 则加 1，SPARC 规范要求：兼容的实现得支持最少 4 个 Trap 嵌套 (用这个表示：MAXTL >= 4)

当一个 Trap 发生时，硬件做的反应是：

a. 自动保存 PC, NPC, (CCR, ASI, PSTATE, CWP) 寄存器到 TPC, TNPC, TSTATE (CCR, ASI, PSTATE, CWP 一起保存在 TSTATE)
b. 将 PSTATE 置为 privileged 模式
c. 跳转到 Trap vector 入口

因此与 Trap 相关的寄存器有：

TL, Trap Level 寄存器，指定当前处理器的 Trap Level，0 为正常模式

TSTATE, Trap State 寄存器，实现有 MAXTL 个
TT, Trap Type 寄存器，实现有 MAXTL 个
TPC, Trap Program Counter 寄存器，实现有 MAXTL 个
TNPC, Trap Next Program Counter 寄存器，实现有 MAXTL 个

因为 PC, NPC, (CCR, ASI, PSTATE, CWP) 都是硬件自动保存，因此支持多少层的 Trap 嵌套硬件就得实现相应数目的 TPC, TNPC，TT 和 TSTATE

Trap 发生时的处理器运行级别，位于 PSTATE[priv] 和 HPSTATE[hpriv]

SPARC 将 Reset, Error, Debug 的状态抽取出来，合称为 RED_state，用于对系统重启，硬件出错处理，处理器调试进行特殊的“照顾”。正常运行的状态称为 execute_state。当以下 Trap 出现时，处理器进入 RED_state:

POR (Power-On reset)
WDR (Watchdog Reset)
XIR (Externally Initiated Reset)
SIR (Software-Initiated Reset)

4.2 Trap 向量入口

4.2.1 正常运行时的向量入口

正常的运行状态 (execute_state)，Trap 的向量入口由一个寄存器指定

使处理器进入特权级的 Trap，其向量入口由 TBA (Trap Base Address) 寄存器指定

使处理器进入超级特权级的 Trap，其向量入口由 HTBA (Hyperprivileged Trap Base Address) 指定

TBA 寄存器的位 63 ~ 15 为向量入口的基地址；

位 13 ~ 5 对应于 TT 寄存器，即不同类型的 Trap，其向量入口偏移也不一样；

TBA[4:0] 始终为 0，即向量入口是 32 字节对齐，每个 Trap handler 的初始长度是 32 字节，8 条指令

SPARC 还根据 Trap Level 的值，使用不同的向量入口，即：TL = 0 时，TBA[14] = 0； TL > 0 时，TBA[14] = 1；

4.2.2 RED_state 向量入口

因为 Reset, Error 及 Debug 的特殊性，当处理器位于 RED_state 状态时，其使用一个固定的向量入口基地址 RSTVaddr = 0xFFFF FFFF F000 0000，这是一个物理地址，位于物理地址空间的高 256MB。

RED_state 下的 Trap 偏移为：

Reserved	0x00
POR (Power-On reset)	0x20
WDR (Watchdog Reset)	0x40
XIR (Externally Initiated Reset)	0x60
SIR (Software-Initiated Reset)	0x80
Other	0xA0

因此上电启动时，UltraSPARC 处理器是从 0xFFFF FFFF F000 0020 处取第一条指令的，则 Bootloader 也应该位于该地址处

4.3 Trap 向量类型

Trap 的类型由 TT 寄存器指定，这是一个 9 bits 的寄存器，因此其最多可描述 2^9 = 512 种 Trap

但实际中，一个处理器要不了这么多的 Trap，以下择其要者 (SPARC V9 64bit)：

TT	Description
0x001	power on reset
0x005	RED state exception
0x008	instruction access exception
0x009	instruction access MMU miss
0x00a	instruction access error
0x010	illegal instruction
0x011	privileged opcode
0x024 - 27	clean window
0x030	data access exception
0x031	data access MMU miss
0x032	data access error
0x033	data access protection
0x034	memory address not aligned
0x041 - 04F	interrupt level N (N = 1 ~ 15)
0x080 - 09F	spill N normal (N = 0 ~ 7)
0x0A0 - 0BF	spill N other (N = 0 ~ 7)
0x0C0 - 0DF	fill N normal (N = 0 ~ 7)
0x0E0 - 0FF	fill N other (N = 0 ~ 7)
0x100 - 17F	trap instruction

为加快 TLB 相关 Trap 的处理， UltraSPARC 还引入了:

0x064	fast instruction access MMU miss
0x068	fast data access MMU miss
0x06C	fast data access protection

SPARC V7 (32bit) 的：

Exception or Interrupt Request	Priority	TT
reset	1	(see text)
data_store_error	2	0x2B
instruction_access_MMU_miss	2	0x3C
instruction_access_error	3	0x21
r_register_access_error	4	0x20
instruction_access_exception	5	0x01
privileged_instruction	6	0x03
illegal_instruction	7	0x02
fp_disabled	8	0x04
cp_disabled	8	0x24
unimplemented_FLUSH	8	0x25
watchpoint_detected	8	0x0B
window_overflow	9	0x05
window_underflow	9	0x06
mem_address_not_aligned	10	0x07
fp_exception	11	0x08
cp_exception	11	0x28
data_access_error	12	0x29
data_access_MMU_miss	12	0x2C
data_access_exception	13	0x09
tag_overflow	14	0x0A
division_by_zero	15	0x2A
trap_instruction	16	0x80 - 0xFF
interrupt_level_15	17	0x1F
interrupt_level_14	18	0x1E
interrupt_level_13	19	0x1D

interrupt_level_12	20	0x1C
interrupt_level_11	21	0x1B
interrupt_level_10	22	0x1A
interrupt_level_9	23	0x19
interrupt_level_8	24	0x18
interrupt_level_7	25	0x17
interrupt_level_6	26	0x16
interrupt_level_5	27	0x15
interrupt_level_4	28	0x14
interrupt_level_3	29	0x13
interrupt_level_2	30	0x12
interrupt_level_1	31	0x11
impl.-dependent exception	impl.-dep.	0x60 - 0x7F

4.4 Linux SPARC Trap 入口概要

4.4.1 SPARC64

Linux SPARC64 的 Trap table 定义在 arch/sparc/kernel/ttable.S:

```
sparc64_ttable_tl0:
tl0_resv000:    BOOT_KERNEL BTRAP(0x1) BTRAP(0x2) BTRAP(0x3)
tl0_resv004:    BTRAP(0x4)  BTRAP(0x5) BTRAP(0x6) BTRAP(0x7)
tl0_iax:    membar #Sync
        TRAP_NOSAVE_7INSNS(__spitfire_insn_access_exception)        -------------> TT = 0x008
......
......
tl0_resv03e:    BTRAP(0x3e) BTRAP(0x3f) BTRAP(0x40)
/* 以下是 15 个 interrupt 的入口 */
#ifdef CONFIG_SMP
tl0_irq1:   TRAP_IRQ(smp_call_function_client, 1)           -------------> TT = 0x041
tl0_irq2:   TRAP_IRQ(smp_receive_signal_client, 2)
tl0_irq3:   TRAP_IRQ(smp_penguin_jailcell, 3)
tl0_irq4:   TRAP_IRQ(smp_new_mmu_context_version_client, 4)
#else
tl0_irq1:   BTRAP(0x41)
tl0_irq2:   BTRAP(0x42)
tl0_irq3:   BTRAP(0x43)
tl0_irq4:   BTRAP(0x44)
#endif
tl0_irq5:   TRAP_IRQ(handler_irq, 5)        ----> TT = 0x045 接到中断控制器，由 handler_irq() 负责处理
#ifdef CONFIG_SMP
tl0_irq6:   TRAP_IRQ(smp_call_function_single_client, 6)
#else
tl0_irq6:   BTRAP(0x46)
#endif
tl0_irq7:   TRAP_IRQ(deferred_pcr_work_irq, 7)
#ifdef CONFIG_KGDB
tl0_irq8:   TRAP_IRQ(smp_kgdb_capture_client, 8)
#else
tl0_irq8:   BTRAP(0x48)
#endif
tl0_irq9:   BTRAP(0x49)
```

```
tl0_irq10:  BTRAP(0x4a) BTRAP(0x4b) BTRAP(0x4c) BTRAP(0x4d)
tl0_irq14:  TRAP_IRQ(timer_interrupt, 14)      ---> 时钟中断处理 timer_interrupt(), TT = 0x04E
tl0_irq15:  TRAP_NMI_IRQ(perfctr_irq, 15)      ---> 性能计数器中断处理 perfctr_irq(), TT = 0x04F
tl0_resv050:    BTRAP(0x50) BTRAP(0x51) BTRAP(0x52) BTRAP(0x53) BTRAP(0x54) BTRAP(0x55)
tl0_resv056:    BTRAP(0x56) BTRAP(0x57) BTRAP(0x58) BTRAP(0x59) BTRAP(0x5a) BTRAP(0x5b)
tl0_resv05c:    BTRAP(0x5c) BTRAP(0x5d) BTRAP(0x5e) BTRAP(0x5f)
tl0_ivec:   TRAP_IVEC
tl0_paw:    TRAP(do_paw)
tl0_vaw:    TRAP(do_vaw)
tl0_cee:    membar #Sync
        TRAP_NOSAVE_7INSNS(__spitfire_cee_trap)
tl0_iamiss:
#include    "itlb_miss.S"   ---> TT=0x064, 65 ~ 67 为空，则其入口可用空间为 32 * 4 字节，可存储 32 条指令；
文件 itlb_miss.S 中的指令数正好为 32
tl0_damiss:
#include    "dtlb_miss.S"    ---> TT = 0x068，同上，占用 4 个入口。dtlb_miss.S 中的指令数也是 32
tl0_daprot:
#include    "dtlb_prot.S"    ---> TT = 0x06C，同上，占用 4 个入口。dtlb_prot.S 中的指令数也是 32
tl0_fecc:   BTRAP(0x70) /* Fast-ECC on Cheetah */
tl0_dcpe:   BTRAP(0x71) /* D-cache Parity Error on Cheetah+ */
tl0_icpe:   BTRAP(0x72) /* I-cache Parity Error on Cheetah+ */
......
......
```

handler_irq() 定义于 arch/sparc/kernel/irq_64.c

sparc64_ttable_tl0 则由 arch/sparc/kernel/head_64.S 中的 setup_tba() --> setup_trap_table() 链入 TBA

4.4.2 SPARC32

Linux SPARC32 的 trap entry 定义于：arch/sparc/kernel/head_32.S

```
trapbase:
#ifdef CONFIG_SMP
trapbase_cpu0:
#endif
/* We get control passed to us here at t_zero. */
t_zero: b gokernel; nop; nop; nop;
t_tflt: SPARC_TFAULT                    /* Inst. Access Exception        */
t_bins: TRAP_ENTRY(0x2, bad_instruction)    /* Illegal Instruction          */
t_pins: TRAP_ENTRY(0x3, priv_instruction)   /* Privileged Instruction        */
t_fpd:  TRAP_ENTRY(0x4, fpd_trap_handler)   /* Floating Point Disabled       */
t_wovf: WINDOW_SPILL                     /* Window Overflow              */
t_wunf: WINDOW_FILL                      /* Window Underflow             */
t_mna:  TRAP_ENTRY(0x7, mna_handler)        /* Memory Address Not Aligned    */  ---> 非对齐访问异常
t_fpe:  TRAP_ENTRY(0x8, fpe_trap_handler)   /* Floating Point Exception      */
t_dflt: SPARC_DFAULT                     /* Data Miss Exception          */
t_tio:  TRAP_ENTRY(0xa, do_tag_overflow)    /* Tagged Instruction Ovrflw     */
t_wpt:  TRAP_ENTRY(0xb, do_watchpoint)      /* Watchpoint Detected          */
t_badc: BAD_TRAP(0xc) BAD_TRAP(0xd) BAD_TRAP(0xe) BAD_TRAP(0xf) BAD_TRAP(0x10)
t_irq1: TRAP_ENTRY_INTERRUPT(1)           /* IRQ Software/SBUS Level 1     */      --> TT = 0x011
t_irq2: TRAP_ENTRY_INTERRUPT(2)           /* IRQ SBUS Level 2              */
t_irq3: TRAP_ENTRY_INTERRUPT(3)           /* IRQ SCSI/DMA/SBUS Level 3     */
t_irq4: TRAP_ENTRY_INTERRUPT(4)           /* IRQ Software Level 4          */
t_irq5: TRAP_ENTRY_INTERRUPT(5)           /* IRQ SBUS/Ethernet Level 5     */
t_irq6: TRAP_ENTRY_INTERRUPT(6)           /* IRQ Software Level 6          */
t_irq7: TRAP_ENTRY_INTERRUPT(7)           /* IRQ Video/SBUS Level 5        */
t_irq8: TRAP_ENTRY_INTERRUPT(8)           /* IRQ SBUS Level 6              */
t_irq9: TRAP_ENTRY_INTERRUPT(9)           /* IRQ SBUS Level 7              */
t_irq10:TRAP_ENTRY_INTERRUPT(10)          /* IRQ Timer #1 (one we use)     */
t_irq11:TRAP_ENTRY_INTERRUPT(11)          /* IRQ Floppy Intr.              */
t_irq12:TRAP_ENTRY_INTERRUPT(12)          /* IRQ Zilog serial chip         */
t_irq13:TRAP_ENTRY_INTERRUPT(13)          /* IRQ Audio Intr.               */
t_irq14:TRAP_ENTRY_INTERRUPT(14)          /* IRQ Timer #2                  */
    .globl  t_nmi
#ifndef CONFIG_SMP
t_nmi:  NMI_TRAP                        /* Level 15 (NMI)               */          ---> TT = 0x015
#else
```

```
t_nmi:  TRAP_ENTRY(0x1f, linux_trap_ipi15_sun4m)
#endif
t_racc: TRAP_ENTRY(0x20, do_reg_access)     /* General Register Access Error */
t_iacce:BAD_TRAP(0x21)                      /* Instr Access Error            */
t_bad22:BAD_TRAP(0x22) BAD_TRAP(0x23)
t_cpdis:TRAP_ENTRY(0x24, do_cp_disabled)    /* Co-Processor Disabled         */
t_uflsh:SKIP_TRAP(0x25, unimp_flush)        /* Unimplemented FLUSH inst.     */
t_bad26:BAD_TRAP(0x26) BAD_TRAP(0x27)
t_cpexc:TRAP_ENTRY(0x28, do_cp_exception)   /* Co-Processor Exception        */
t_dacce:SPARC_DFAULT                        /* Data Access Error             */
t_hwdz: TRAP_ENTRY(0x2a, do_hw_divzero)     /* Division by zero, you lose... */
t_dserr:BAD_TRAP(0x2b)                      /* Data Store Error              */
t_daccm:BAD_TRAP(0x2c)                      /* Data Access MMU-Miss          */
t_bad2d:BAD_TRAP(0x2d) BAD_TRAP(0x2e) BAD_TRAP(0x2f) BAD_TRAP(0x30) BAD_TRAP(0x31)
......
......
```

TRAP_ENTRY 和 TRAP_ENTRY_INTERRUPT 都定义在 arch/sparc/include/asm/head_32.h:

```
/* Generic trap entry. */
#define TRAP_ENTRY(type, label) \
    rd %psr, %l0; b label; rd %wim, %l3; nop;

/*
 * This is for hard interrupts from level 1-14, 15 is non-maskable (nmi) and
 * gets handled with another macro.
 */
#define TRAP_ENTRY_INTERRUPT(int_level) \
        mov int_level, %l7; rd %psr, %l0; b real_irq_entry; rd %wim, %l3;
```

因此这些入口只是一个下级处理函数的简单调用，比如非对齐访问异常的真正处理函数是 mna_handler()，其定义于 arch/sparc/kernel/entry.S （其它处理函数大多定义在此文件中）：

```
    /* This routine handles unaligned data accesses. */
    .align  4
    .globl  mna_handler
mna_handler:
    andcc   %l0, PSR_PS, %g0
```

```
    be  mna_fromuser
     nop

    SAVE_ALL

    wr  %l0, PSR_ET, %psr
    WRITE_PAUSE

    ld  [%l1], %o1
    call    kernel_unaligned_trap
     add    %sp, STACKFRAME_SZ, %o0

    RESTORE_ALL

mna_fromuser:
    SAVE_ALL

    wr  %l0, PSR_ET, %psr       ! re-enable traps
    WRITE_PAUSE

    ld  [%l1], %o1
    call    user_unaligned_trap
     add    %sp, STACKFRAME_SZ, %o0

    RESTORE_ALL
```

中断处理函数 real_irq_entry() 也是定义在 entry.S，在做一些简单的处理后，其会调用真正的处理函数 handler_irq()，SPARC32 定义在 arch/sparc/kernel/irq_32.c

32 bit 的 Trap 向量表基地址 trapbase 会在 arch/sparc/kernel/setup_32.c 中被写入 TBA 寄存器：

```
extern unsigned long trapbase;

/* Pretty sick eh? */
static void prom_sync_me(void)
{
    unsigned long prom_tbr, flags;

    /* XXX Badly broken. FIX! - Anton */
```

```
    local_irq_save(flags);
    __asm__ __volatile__("rd %%tbr, %0\n\t" : "=r" (prom_tbr));
    __asm__ __volatile__("wr %0, 0x0, %%tbr\n\t"
                "nop\n\t"
                "nop\n\t"
                "nop\n\t" : : "r" (&trapbase));

    prom_printf("PROM SYNC COMMAND...\n");
    show_free_areas();
    if(current->pid != 0) {
        local_irq_enable();
        sys_sync();
        local_irq_disable();
    }
    prom_printf("Returning to prom\n");

    __asm__ __volatile__("wr %0, 0x0, %%tbr\n\t"
                "nop\n\t"
                "nop\n\t"
                "nop\n\t" : : "r" (prom_tbr));
    local_irq_restore(flags);

    return;
}
```

prom_sync_me() 会在 setup_arch() 里被链入