程序 11-1 linux/kernel/math/math_emulate.c

```
1  /*
2   * linux/kernel/math/math_emulate.c
3   *
4   * (C) 1991 Linus Torvalds
5   */
6
7  /*
8   * Limited emulation 27.12.91 - mostly loads/stores, which gcc wants
9   * even for soft-float, unless you use bruce evans' patches. The patches
10  * are great, but they have to be re-applied for every version, and the
11  * library is different for soft-float and 80387. So emulation is more
12  * practical, even though it's slower.
13  *
14  * 28.12.91 - loads/stores work, even BCD. I'll have to start thinking
15  * about add/sub/mul/div. Urgel. I should find some good source, but I'll
16  * just fake up something.
17  *
18  * 30.12.91 - add/sub/mul/div/com seem to work mostly. I should really
19  * test every possible combination.
20  */
    /*
     * 仿真范围有限的程序 91.12.27 - 绝大多数是一些加载/存储指令。除非你使用
     * 了 Bruce Evans 的补丁程序，否则即使使用软件执行浮点运算，gcc 也需要这些
     * 指令。Bruce 的补丁程序非常好，但每次更换 gcc 版本你都得用这个补丁程序。
     * 而且对于软件浮点实现和 80387，所使用的库是不同的。因此使用仿真是更为实
     * 际的方法，尽管仿真方法更慢。
     *
     * 91.12.28 - 加载/存储协处理器指令可以用了，即使是 BCD 码的也能使用。我将
     * 开始考虑实现 add/sub/mul/div 指令。唉，我应该找一些好的资料，不过现在
     * 我会先仿造一些操作。
     *
     * 91.12.30 - add/sub/mul/div/com 这些指令好像大多数都可以使用了。我真应
     * 该测试每种指令可能的组合操作。
     */
21
22 /*
23  * This file is full of ugly macros etc: one problem was that gcc simply
24  * didn't want to make the structures as they should be: it has to try to
25  * align them. Sickening code, but at least I've hidden the ugly things
26  * in this one file: the other files don't need to know about these things.
27  *
28  * The other files also don't care about ST(x) etc - they just get addresses
29  * to 80-bit temporary reals, and do with them as they please. I wanted to
30  * hide most of the 387-specific things here.
31  */
    /*
     * 这个程序中到处都是些别扭的宏：问题之一是 gcc 就是不想把结构建立成其应该
     * 成为的样子：gcc 企图对结构进行对齐处理。真是讨厌，不过我起码已经把所有
     * 蹩脚的代码都隐藏在这么一个文件中了：其他程序文件不需要了解这些信息。
     *
     * 其他的程序也不需要知道 ST(x)等 80387 内部结构 - 它们只需要得到 80 位临时
     * 实数的地址就可以随意操作。我想尽可能在这里隐藏所有 387 专有信息。
```

```
 */
32
33 #include <signal.h>        // 信号头文件。定义信号符号，信号结构及信号操作函数原型。
34
35 #define __ALIGNED_TEMP_REAL 1
36 #include <linux/math_emu.h> // 协处理器头文件。定义临时实数结构和 387 寄存器操作宏等。
37 #include <linux/kernel.h>   // 内核头文件。含有一些内核常用函数的原形定义。
38 #include <asm/segment.h>    // 段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数。
39
40 #define bswapw(x) __asm__("xchgb %%al,%%ah":"=a" (x):"" ((short)x)) // 交换 2 字节位置。
41 #define ST(x) (*__st((x)))                          // 取仿真的 ST(x)累加器值。
42 #define PST(x) ((const temp_real *) __st((x)))      // 取仿真的 ST(x)累加器的指针。
43
44 /*
45  * We don't want these inlined - it gets too messy in the machine-code.
46  */
   /*
    * 我们不想让这些成为嵌入的语句 - 因为这会使得到的机器码太混乱。
    */
   // 以下这些是相同名称浮点指令的仿真函数。
47 static void fpop(void);
48 static void fpush(void);
49 static void fxchg(temp_real_unaligned * a, temp_real_unaligned * b);
50 static temp_real_unaligned * __st(int i);
51
   // 执行浮点指令仿真。
   // 该函数首先检测仿真的 I387 结构状态字寄存器中是否有未屏蔽的异常标志置位。若有则对状
   // 态字中忙标志 B 进行设置。然后把指令指针保存起来，并取出代码指针 EIP 处的 2 字节浮点
   // 指令代码 code。接着分析代码 code，并根据其含义进行处理。针对不同代码类型值，Linus
   // 使用了几个不同的 switch 程序块进行仿真处理。
   // 参数是 info 结构的指针。
52 static void do_emu(struct info * info)
53 {
54         unsigned short code;
55         temp_real tmp;
56         char * address;
57
   // 该函数首先检测仿真的 I387 结构状态字寄存器中是否有未屏蔽的异常标志置位。若有就设置
   // 状态字中的忙标志 B（位 15），否则复位 B 标志。然后我们把指令指针保存起来。再看看执
   // 行本函数的代码是否是用户代码。如果不是，即调用者的代码段选择符不等于 0x0f，则说明
   // 内核中有代码使用了浮点指令。于是在显示出浮点指令出的 CS、EIP 值和信息“内核中需要
   // 数学仿真”后停机。
58         if (I387.cwd & I387.swd & 0x3f)
59                 I387.swd |= 0x8000;                 // 设置忙标志 B。
60         else
61                 I387.swd &= 0x7fff;                 // 清忙标志 B。
62         ORIG_EIP = EIP;                             // 保存浮点指令指针。
63 /* 0x0007 means user code space */
64         if (CS != 0x000F) {                         // 不是用户代码则停机。
65                 printk("math_emulate: %04x:%08x\n\r",CS,EIP);
66                 panic("Math emulation needed in kernel");
67         }
   // 然后我们取出代码指针 EIP 处的 2 字节浮点指令代码 code。 由于 Intel CPU 存储数据时是
```

```
    // “小头”（Little endien）在前的，此时取出的代码正好与指令的第 1、第 2 字节顺序颠倒。
    // 因此我们需要交换一下 code 中两个字节的顺序。然后再屏蔽掉第 1 个代码字节中的 ESC 位
    // （二进制 11011）。接着把浮点指令指针 EIP 保存到 TSS 段 i387 结构中的 fip 字段中，而 CS
    // 保存到 fcs 字段中，同时把略微处理过的浮点指令代码 code 放到 fcs 字段的高 16 位中。
    // 保存这些值是为了在出现仿真的处理器异常时程序可以像使用真实的协处理器一样进行处理。
    // 最后让 EIP 指向随后的浮点指令或操作数。
68          code = get_fs_word((unsigned short *) EIP);  // 取 2 字节的浮点指令代码。
69          bswapw(code);                                // 交换高低字节。
70          code &= 0x7ff;                               // 屏蔽代码中的 ESC 码。
71          I387.fip = EIP;                              // 保存指令指针。
72          *(unsigned short *) &I387.fcs = CS;          // 保存代码段选择符。
73          *(1+(unsigned short *) &I387.fcs) = code;    // 保存代码。
74          EIP += 2;                                    // 指令指针指向下一个字节。
    // 然后分析代码值 code，并根据其含义进行处理。针对不同代码类型值，Linus 使用了几个不同
    // 的 switch 程序块进行处理。首先，若指令操作码是具有固定代码值（与寄存器等无关），则
    // 在下面处理。
75          switch (code) {
76                  case 0x1d0: /* fnop */    /* 空操作指令 FNOP */
77                          return;
78                  case 0x1d1: case 0x1d2: case 0x1d3:  // 无效指令代码。发信号，退出。
79                  case 0x1d4: case 0x1d5: case 0x1d6: case 0x1d7:
80                          math_abort(info,1<<(SIGILL-1));
81                  case 0x1e0:                  // FCHS - 改变 ST 符号位。即 ST = -ST。
82                          ST(0).exponent ^= 0x8000;
83                          return;
84                  case 0x1e1:                  // FABS - 取绝对值。即 ST = |ST|。
85                          ST(0).exponent &= 0x7fff;
86                          return;
87                  case 0x1e2: case 0x1e3:   // 无效指令代码。发信号，退出。
88                          math_abort(info,1<<(SIGILL-1));
89                  case 0x1e4:                  // FTST - 测试 TS，同时设置状态字中 Cn。
90                          ftst(PST(0));
91                          return;
92                  case 0x1e5:                  // FXAM - 检查 TS 值，同时修改状态字中 Cn。
93                          printk("fxam not implemented\n\r");  // 未实现。发信号退出。
94                          math_abort(info,1<<(SIGILL-1));
95                  case 0x1e6: case 0x1e7:   // 无效指令代码。发信号，退出。
96                          math_abort(info,1<<(SIGILL-1));
97                  case 0x1e8:                  // FLD1 - 加载常数 1.0 到累加器 ST。
98                          fpush();
99                          ST(0) = CONST1;
100                         return;
101                 case 0x1e9:                  // FLDL2T - 加载常数 Log2(10) 到累加器 ST。
102                         fpush();
103                         ST(0) = CONSTL2T;
104                         return;
105                 case 0x1ea:                  // FLDL2E - 加载常数 Log2(e) 到累加器 ST。
106                         fpush();
107                         ST(0) = CONSTL2E;
108                         return;
109                 case 0x1eb:                  // FLDPI - 加载常数 Pi 到累加器 ST。
110                         fpush();
111                         ST(0) = CONSTPI;
```

```
112                     return;
113             case 0x1ec:                     // FLDLG2 - 加载常数 Log10(2) 到累加器 ST。
114                     fpush();
115                     ST(0) = CONSTLG2;
116                     return;
117             case 0x1ed:                     // FLDLN2 - 加载常数 Loge(2) 到累加器 ST。
118                     fpush();
119                     ST(0) = CONSTLN2;
120                     return;
121             case 0x1ee:                     // FLDZ - 加载常数 0.0 到累加器 ST。
122                     fpush();
123                     ST(0) = CONSTZ;
124                     return;
125             case 0x1ef:                     // 无效和未实现仿真指令代码。发信号，退出。
126                     math_abort(info,1<<(SIGILL-1));
127             case 0x1f0: case 0x1f1: case 0x1f2: case 0x1f3:
128             case 0x1f4: case 0x1f5: case 0x1f6: case 0x1f7:
129             case 0x1f8: case 0x1f9: case 0x1fa: case 0x1fb:
130             case 0x1fc: case 0x1fd: case 0x1fe: case 0x1ff:
131                     printk("%04x fxxx not implemented\n\r",code + 0xc800);
132                     math_abort(info,1<<(SIGILL-1));
133             case 0x2e9:                     // FUCOMPP - 无次序比较。
134                     fucom(PST(1),PST(0));
135                     fpop(); fpop();
136                     return;
137             case 0x3d0: case 0x3d1:         // FNOP - 对 387。!!应该是 0x3e0, 0x3e1。
138                     return;
139             case 0x3e2:                     // FCLEX - 清状态字中异常标志。
140                     I387.swd &= 0x7f00;
141                     return;
142             case 0x3e3:                     // FINIT - 初始化协处理器。
143                     I387.cwd = 0x037f;
144                     I387.swd = 0x0000;
145                     I387.twd = 0x0000;
146                     return;
147             case 0x3e4:                     // FNOP - 对 80387。
148                     return;
149             case 0x6d9:                     // FCOMPP - ST(1)与 ST 比较，出栈操作两次。
150                     fcom(PST(1),PST(0));
151                     fpop(); fpop();
152                     return;
153             case 0x7e0:                     // FSTSW AX - 保存当前状态字到 AX 寄存器中。
154                     *(short *) &EAX = I387.swd;
155                     return;
156     }

// 下面开始处理第 2 字节最后 3 比特是 REG 的指令。即 11011,XXXXXXXX,REG 形式的代码。
157     switch (code >> 3) {
158             case 0x18:                      // FADD ST, ST(i)。
159                     fadd(PST(0),PST(code & 7),&tmp);
160                     real_to_real(&tmp,&ST(0));
161                     return;
162             case 0x19:                      // FMUL ST, ST(i)。
```

```
163                     fmul(PST(0),PST(code & 7),&tmp);
164                     real_to_real(&tmp,&ST(0));
165                     return;
166             case 0x1a:              // FCOM ST(i)。
167                     fcom(PST(code & 7),&tmp);
168                     real_to_real(&tmp,&ST(0));
169                     return;
170             case 0x1b:              // FCOMP ST(i)。
171                     fcom(PST(code & 7),&tmp);
172                     real_to_real(&tmp,&ST(0));
173                     fpop();
174                     return;
175             case 0x1c:              // FSUB ST, ST(i)。
176                     real_to_real(&ST(code & 7),&tmp);
177                     tmp.exponent ^= 0x8000;
178                     fadd(PST(0),&tmp,&tmp);
179                     real_to_real(&tmp,&ST(0));
180                     return;
181             case 0x1d:              // FSUBR ST, ST(i)。
182                     ST(0).exponent ^= 0x8000;
183                     fadd(PST(0),PST(code & 7),&tmp);
184                     real_to_real(&tmp,&ST(0));
185                     return;
186             case 0x1e:              // FDIV ST, ST(i)。
187                     fdiv(PST(0),PST(code & 7),&tmp);
188                     real_to_real(&tmp,&ST(0));
189                     return;
190             case 0x1f:              // FDIVR ST, ST(i)。
191                     fdiv(PST(code & 7),PST(0),&tmp);
192                     real_to_real(&tmp,&ST(0));
193                     return;
194             case 0x38:              // FLD ST(i)。
195                     fpush();
196                     ST(0) = ST((code & 7)+1);
197                     return;
198             case 0x39:              // FXCH ST(i)。
199                     fxchg(&ST(0),&ST(code & 7));
200                     return;
201             case 0x3b:              // FSTP ST(i)。
202                     ST(code & 7) = ST(0);
203                     fpop();
204                     return;
205             case 0x98:              // FADD ST(i), ST。
206                     fadd(PST(0),PST(code & 7),&tmp);
207                     real_to_real(&tmp,&ST(code & 7));
208                     return;
209             case 0x99:              // FMUL ST(i), ST。
210                     fmul(PST(0),PST(code & 7),&tmp);
211                     real_to_real(&tmp,&ST(code & 7));
212                     return;
213             case 0x9a:              // FCOM ST(i)。
214                     fcom(PST(code & 7),PST(0));
215                     return;
```

```
216                 case 0x9b:                  // FCOMP ST(i)。
217                         fcom(PST(code & 7),PST(0));
218                         fpop();
219                         return;
220                 case 0x9c:                  // FSUBR ST(i), ST。
221                         ST(code & 7).exponent ^= 0x8000;
222                         fadd(PST(0),PST(code & 7),&tmp);
223                         real_to_real(&tmp,&ST(code & 7));
224                         return;
225                 case 0x9d:                  // FSUB ST(i), ST。
226                         real_to_real(&ST(0),&tmp);
227                         tmp.exponent ^= 0x8000;
228                         fadd(PST(code & 7),&tmp,&tmp);
229                         real_to_real(&tmp,&ST(code & 7));
230                         return;
231                 case 0x9e:                  // FDIVR ST(i), ST。
232                         fdiv(PST(0),PST(code & 7),&tmp);
233                         real_to_real(&tmp,&ST(code & 7));
234                         return;
235                 case 0x9f:                  // FDIV ST(i), ST。
236                         fdiv(PST(code & 7),PST(0),&tmp);
237                         real_to_real(&tmp,&ST(code & 7));
238                         return;
239                 case 0xb8:                  // FFREE ST(i)。未实现。
240                         printk("ffree not implemented\n\r");
241                         math_abort(info,1<<(SIGILL-1));
242                 case 0xb9:                  // FXCH ST(i)。
243                         fxchg(&ST(0),&ST(code & 7));
244                         return;
245                 case 0xba:                  // FST ST(i)。
246                         ST(code & 7) = ST(0);
247                         return;
248                 case 0xbb:                  // FSTP ST(i)。
249                         ST(code & 7) = ST(0);
250                         fpop();
251                         return;
252                 case 0xbc:                  // FUCOM ST(i)。
253                         fucom(PST(code & 7),PST(0));
254                         return;
255                 case 0xbd:                  // FUCOMP ST(i)。
256                         fucom(PST(code & 7),PST(0));
257                         fpop();
258                         return;
259                 case 0xd8:                  // FADDP ST(i), ST。
260                         fadd(PST(code & 7),PST(0),&tmp);
261                         real_to_real(&tmp,&ST(code & 7));
262                         fpop();
263                         return;
264                 case 0xd9:                  // FMULP ST(i), ST。
265                         fmul(PST(code & 7),PST(0),&tmp);
266                         real_to_real(&tmp,&ST(code & 7));
267                         fpop();
268                         return;
```

```
269                 case 0xda:                  // FCOMP ST(i)。
270                         fcom(PST(code & 7),PST(0));
271                         fpop();
272                         return;
273                 case 0xdc:                  // FSUBRP ST(i), ST。
274                         ST(code & 7).exponent ^= 0x8000;
275                         fadd(PST(0),PST(code & 7),&tmp);
276                         real_to_real(&tmp,&ST(code & 7));
277                         fpop();
278                         return;
279                 case 0xdd:                  // FSUBP ST(i), ST。
280                         real_to_real(&ST(0),&tmp);
281                         tmp.exponent ^= 0x8000;
282                         fadd(PST(code & 7),&tmp,&tmp);
283                         real_to_real(&tmp,&ST(code & 7));
284                         fpop();
285                         return;
286                 case 0xde:                  // FDIVRP ST(i), ST。
287                         fdiv(PST(0),PST(code & 7),&tmp);
288                         real_to_real(&tmp,&ST(code & 7));
289                         fpop();
290                         return;
291                 case 0xdf:                  // FDIVP ST(i), ST。
292                         fdiv(PST(code & 7),PST(0),&tmp);
293                         real_to_real(&tmp,&ST(code & 7));
294                         fpop();
295                         return;
296                 case 0xf8:                  // FFREE ST(i)。未实现。
297                         printk("ffree not implemented\n\r");
298                         math_abort(info,1<<(SIGILL-1));
299                         fpop();
300                         return;
301                 case 0xf9:                  // FXCH ST(i)。
302                         fxchg(&ST(0),&ST(code & 7));
303                         return;
304                 case 0xfa:                  // FSTP ST(i)。
305                 case 0xfb:                  // FSTP ST(i)。
306                         ST(code & 7) = ST(0);
307                         fpop();
308                         return;
309         }

    // 处理第 2 个字节位 7--6 是 MOD、位 2--0 是 R/M 的指令，即 11011,XXX,MOD,XXX,R/M 形式的
    // 代码。MOD 在各子程序中处理，因此这里首先让代码与上 0xe7（0b11100111）屏蔽掉 MOD。
310         switch ((code>>3) & 0xe7) {
311                 case 0x22:                  // FST  - 保存单精度实数（短实数）。
312                         put_short_real(PST(0),info,code);
313                         return;
314                 case 0x23:                  // FSTP  - 保存单精度实数（短实数）。
315                         put_short_real(PST(0),info,code);
316                         fpop();
317                         return;
318                 case 0x24:                  // FLDENV - 加载协处理器状态和控制寄存器等。
```

```
319                     address = ea(info,code);
320                     for (code = 0 ; code < 7 ; code++) {
321                             ((long *) & I387)[code] =
322                                get_fs_long((unsigned long *) address);
323                             address += 4;
324                     }
325                     return;
326             case 0x25:              // FLDCW - 加载控制字。
327                     address = ea(info,code);
328                     *(unsigned short *) &I387.cwd =
329                             get_fs_word((unsigned short *) address);
330                     return;
331             case 0x26:              // FSTENV - 储存协处理器状态和控制寄存器等。
332                     address = ea(info,code);
333                     verify_area(address,28);
334                     for (code = 0 ; code < 7 ; code++) {
335                             put_fs_long( ((long *) & I387)[code],
336                                     (unsigned long *) address);
337                             address += 4;
338                     }
339                     return;
340             case 0x27:              // FSTCW - 储存控制字。
341                     address = ea(info,code);
342                     verify_area(address,2);
343                     put_fs_word(I387.cwd,(short *) address);
344                     return;
345             case 0x62:              // FIST - 储存短整形数。
346                     put_long_int(PST(0),info,code);
347                     return;
348             case 0x63:              // FISTP - 储存短整形数。
349                     put_long_int(PST(0),info,code);
350                     fpop();
351                     return;
352             case 0x65:              // FLD - 加载扩展（临时）实数。
353                     fpush();
354                     get_temp_real(&tmp,info,code);
355                     real_to_real(&tmp,&ST(0));
356                     return;
357             case 0x67:              // FSTP - 储存扩展实数。
358                     put_temp_real(PST(0),info,code);
359                     fpop();
360                     return;
361             case 0xa2:              // FST - 储存双精度实数。
362                     put_long_real(PST(0),info,code);
363                     return;
364             case 0xa3:              // FSTP - 储存双精度实数。
365                     put_long_real(PST(0),info,code);
366                     fpop();
367                     return;
368             case 0xa4:              // FRSTOR - 恢复所有 108 字节的寄存器内容。
369                     address = ea(info,code);
370                     for (code = 0 ; code < 27 ; code++) {
371                             ((long *) & I387)[code] =
```

```
372                                 get_fs_long((unsigned long *) address);
373                         address += 4;
374                 }
375                 return;
376         case 0xa6:                  // FSAVE - 保存所有 108 字节寄存器内容。
377                 address = ea(info,code);
378                 verify_area(address,108);
379                 for (code = 0 ; code < 27 ; code++) {
380                         put_fs_long( ((long *) & I387)[code],
381                                 (unsigned long *) address);
382                         address += 4;
383                 }
384                 I387.cwd = 0x037f;
385                 I387.swd = 0x0000;
386                 I387.twd = 0x0000;
387                 return;
388         case 0xa7:                  // FSTSW - 保存状态字。
389                 address = ea(info,code);
390                 verify_area(address,2);
391                 put_fs_word(I387.swd,(short *) address);
392                 return;
393         case 0xe2:                  // FIST - 保存短整型数。
394                 put_short_int(PST(0),info,code);
395                 return;
396         case 0xe3:                  // FISTP - 保存短整型数。
397                 put_short_int(PST(0),info,code);
398                 fpop();
399                 return;
400         case 0xe4:                  // FBLD - 加载 BCD 类型数。
401                 fpush();
402                 get_BCD(&tmp,info,code);
403                 real_to_real(&tmp,&ST(0));
404                 return;
405         case 0xe5:                  // FILD - 加载长整型数。
406                 fpush();
407                 get_longlong_int(&tmp,info,code);
408                 real_to_real(&tmp,&ST(0));
409                 return;
410         case 0xe6:                  // FBSTP - 保存 BCD 类型数。
411                 put_BCD(PST(0),info,code);
412                 fpop();
413                 return;
414         case 0xe7:                  // BISTP - 保存长整型数。
415                 put_longlong_int(PST(0),info,code);
416                 fpop();
417                 return;
418         }
    // 下面处理第 2 类浮点指令。首先根据指令代码的位 10--9 的 MF 值取指定类型的数，然后根据
    // OPA 和 OPB 的组合值进行分别处理。即处理 11011,MF,000,XXX,R/M 形式的指令代码。
419         switch (code >> 9) {
420         case 0:                     // MF = 00，短实数（32 位实数）。
421                 get_short_real(&tmp,info,code);
422                 break;
```

```
423                 case 1:                     // MF = 01，短整数（32 位整数）。
424                         get_long_int(&tmp,info,code);
425                         break;
426                 case 2:                     // MF = 10，长实数（64 位实数）。
427                         get_long_real(&tmp,info,code);
428                         break;
429                 case 4:                     // MF = 11，长整数（64 位整数）。！应是 case 3。
430                         get_short_int(&tmp,info,code);
431         }
    // 处理浮点指令第 2 字节中的 OPB 代码。
432         switch ((code>>3) & 0x27) {
433                 case 0:                     // FADD。
434                         fadd(&tmp,PST(0),&tmp);
435                         real_to_real(&tmp,&ST(0));
436                         return;
437                 case 1:                     // FMUL。
438                         fmul(&tmp,PST(0),&tmp);
439                         real_to_real(&tmp,&ST(0));
440                         return;
441                 case 2:                     // FCOM。
442                         fcom(&tmp,PST(0));
443                         return;
444                 case 3:                     // FCOMP。
445                         fcom(&tmp,PST(0));
446                         fpop();
447                         return;
448                 case 4:                     // FSUB。
449                         tmp.exponent ^= 0x8000;
450                         fadd(&tmp,PST(0),&tmp);
451                         real_to_real(&tmp,&ST(0));
452                         return;
453                 case 5:                     // FSUBR。
454                         ST(0).exponent ^= 0x8000;
455                         fadd(&tmp,PST(0),&tmp);
456                         real_to_real(&tmp,&ST(0));
457                         return;
458                 case 6:                     // FDIV。
459                         fdiv(PST(0),&tmp,&tmp);
460                         real_to_real(&tmp,&ST(0));
461                         return;
462                 case 7:                     // FDIVR。
463                         fdiv(&tmp,PST(0),&tmp);
464                         real_to_real(&tmp,&ST(0));
465                         return;
466         }
    // 处理形如 11011,XX,1,XX,000,R/M 的指令代码。
467         if ((code & 0x138) == 0x100) {          // FLD、FILD。
468                         fpush();
469                         real_to_real(&tmp,&ST(0));
470                         return;
471         }
    // 其余均为无效指令。
472         printk("Unknown math-insns: %04x:%08x %04x\n\r",CS,EIP,code);
```

```
473         math_abort(info,1<<(SIGFPE-1));
474 }
475
    // CPU 异常中断 int7 调用的 80387 仿真接口函数。
    // 若当前进程没有使用过协处理器，就设置使用协处理器标志 used_math，然后初始化 80387
    // 的控制字、状态字和特征字。最后使用中断 int7 调用本函数的返回地址指针作为参数调用
    // 浮点指令仿真主函数 do_emu()。
    // 参数 ___false 是 _orig_eip。
476 void math_emulate(long ___false)
477 {
478         if (!current->used_math) {
479                 current->used_math = 1;
480                 I387.cwd = 0x037f;
481                 I387.swd = 0x0000;
482                 I387.twd = 0x0000;
483         }
484 /* &___false points to info->___orig_eip, so subtract 1 to get info */
485         do_emu((struct info *) ((&___false) - 1));
486 }
487
    // 终止仿真操作。
    // 当处理到无效指令代码或者未实现的指令代码时，该函数首先恢复程序的原 EIP，并发送指定
    // 信号给当前进程。最后将栈指针指向中断 int7 处理过程调用本函数的返回地址，直接返回到
    // 中断处理过程中。
488 void __math_abort(struct info * info, unsigned int signal)
489 {
490         EIP = ORIG_EIP;
491         current->signal |= signal;
492         __asm__("movl %0,%%esp ; ret"::"g" ((long) info));
493 }
494
    // 累加器栈弹出操作。
    // 将状态字 TOP 字段值加 1，并以 7 取模。
495 static void fpop(void)
496 {
497         unsigned long tmp;
498
499         tmp = I387.swd & 0xffffc7ff;
500         I387.swd += 0x00000800;
501         I387.swd &= 0x00003800;
502         I387.swd |= tmp;
503 }
504
    // 累加器栈入栈操作。
    // 将状态字 TOP 字段减 1（即加 7），并以 7 取模。
505 static void fpush(void)
506 {
507         unsigned long tmp;
508
509         tmp = I387.swd & 0xffffc7ff;
510         I387.swd += 0x00003800;
511         I387.swd &= 0x00003800;
512         I387.swd |= tmp;
```

```
513 }
514
    // 交换两个累加器寄存器的值。
515 static void fxchg(temp_real_unaligned * a, temp_real_unaligned * b)
516 {
517         temp_real_unaligned c;
518
519         c = *a;
520         *a = *b;
521         *b = c;
522 }
523
    // 取 ST(i)的内存指针。
    // 取状态字中 TOP 字段值。加上指定的物理数据寄存器号并取模，最后返回 ST(i)对应的指针。
524 static temp_real_unaligned * __st(int i)
525 {
526         i += I387.swd >> 11;            // 取状态字中 TOP 字段值。
527         i &= 7;
528         return (temp_real_unaligned *) (i*10 + (char *)(I387.st_space));
529 }
530
```