更多关于 ADI 公司的 DSP、处理器以及开发工具的技术资料，
请访问网站：http://www.analog.com/ee-note 和 http://www.analog.com/processor
如需技术支持，请发邮件至 processor.support@analog.com 或 processor.tools.support@analog.com

Blackfin®处理器利用 VisualDSP++®工具的调试方法

作者：Jorge Manguane

Rev 1 – December 11, 2006

引言

该文档描述了Blackfin®处理器和VisualDSP++®开发工具的调试特性。通过使用以下方法，程序员在遇到问题并向ADI嵌入式处理器支持团队报告前，可以缩小发生引发问题的范围，因此有助于快速地解决问题。

调试提示和方法：

涵盖以下方面：

- 与启动应用程序相对应，通过仿真器来执行该程序。特别地，讨论了 SDRAM 初始化需要考虑的问题。
- 对于双核处理器，核 B 必须开启。仿真器能自动地完成这一过程；但是当代码在启动时，核 B 必须手动的开锁。
- 硬件错误和软件异常
- Blackfin 处理器调试特性和工具，包括：
  - 跟踪缓冲器
  - 断点(软件、嵌入的和硬件)
  - VDK调试(VDK状态窗口以及VDK状态历史窗口)
- 当 cache 使能时出现的调试问题
- 中断

注意：该电子工程文档不涵盖调试外设时所出现的问题。

仿真器与独立启动

仿真器软件使用.xml文件来配置EZ-KIT Lite®评估板上的资源，如SDRAM时序等。所有存在评估平台(如EZ-KIT Lite板)的Blackfin处理器都有默认的.xml文件，用于在仿真器连接时使用该文件里的定义来初始化某些寄存器。

例如，下面是引用ADSP-BF537-proc.xml文件中的一段内容，用于ADSP-BF537 Blackfin 处理器：

```
<register-reset-definitions>
<register name="EBIU_SDRRC" resetvalue=" 0x03A0"
core="Common" />
<register name="EBIU_SDBCTL" resetvalue=" 0x25"
core="Common" />
<register name="EBIU_SDGCTL" resetvalue="
0x0091998d" core="Common" />
<register name="EBIU_AMGCTL" resetvalue=" 0xff"
core="Common" />
</register-reset-definitions>
```

因此，当开发一个用于ADSP-BF537 EZ-KIT Lite板的应用程序时，SDRAM将在唤醒仿真软件时自动初始化。

然而，当开发一个独立的应用程序时(即不通过仿真器下载，而是直接启动程序)，如果系统中

使用SDRAM存储器，则需要用户编程去使能SDRAM控制器。这需要当创建一个装载文件时，在Project Options对话框的装载页中通过包含一个初始化文件来完成。参阅*ADSP-BF533 Blackfin Booting Process (EE-240)*[1]。

对于ADSP-BF561 Blackfin双核系统，另外的不同点是当把一个应用程序从仿真会话环境转为独立启动时，可能会引发问题。默认时，仿真软件“解锁”核B，并允许它从L1指令内存的开始运行。两个核都在使用时，核B必须由核A开启，通过消除系统复位配置寄存器的位5来完成(SICA_SYSCR)。

在进行改变操作模式和时钟频率时,仿真器打开核B有时会导致另一个问题发生。特定地，当改变PLL或者电压调节器时，核B必须处在空闲状态(不仅仅在一个断点上)。这可能会引发问题，例如如果核B设置了一个断点，而核A在运行一段改变PLL频率的代码，这时就需要在改变PLL频率前，先运行代码将核B处于空闲状态，这可通过增加中断或者GPIO引脚来完成。

硬件错误和软件异常

硬件错误和软件异常是发生在Blackfin处理器上的两个特殊的事件类型，每个类型在事件向量表(EVT)上都有一个单独的入口。每类事件的操作处理程序都要安装，以便在使用文档后文所描述的任一个断点方法时，它们能被捕获到。在这一点上，就能检查到处理器的状态，从而确认是什么导致了这种特殊的事件发生。序列状态寄存器(SEQSTAT)中有两个字段用于获得关于错误状况的更多信息。HWERRCAUSE域用于识别产生一个硬件错误的条件，而EXCAUSE域用于识别产生一个异常的状况。

有许多原因可能导致硬件错误产生，如当一个MMR通过错误字大小访问(如，一个16位的MMR通过32位访问，反之亦然)，或者当内核或DMA控制器试图访问预留的或者未初始化的内存空间时，都可能引发硬件错误。RETI地址将包含引发错误位置的10个位置中的一个地址，如果硬件错误激活了，并且相应的事件进入了服务，则清除该状况，但是硬件错误起因将保留最后的错误状态。

对于ADSP-BF561Blackfin双核处理器，由特定核产生的硬件错误将仅在那个内核上引发一个错误。如果DMA控制器产生一个硬件错误，则该错误将被发送到两个内核中。

在每个处理程序中(硬件错误或异常)，能读取HWERRCAUSE和EXCAUSE字段来确认产生事件的原因。可选的，当使用仿真器调试时，能在处理程序中放置捕获指令，如emuxcept，以便当一个硬件错误和/或一个异常发生时，处理器将停止运行，然后检测SEQSTAT寄存器中的相应位来确认事件的起因。

既然知道了产生事件的原因，则标注出出错指令的地址，用以确定程序运行到什么时候出现了问题。对于异常情况，从异常寄存器(RETX)的返回包含了引发异常指令的地址或者下一条将要执行指令的地址。在RETX的地址依赖于异常类型：服务(S)或错误(E)。*ADSP-BF53x/BF56x Blackfin处理器编程参考*[2] 列出了产生异常的原因，和它们的类型(服务或异常)。为方便查看，该表包含在附录A中。

对于错误类型异常，RETX包含出错指令的地址；而对于服务类型异常，RETX则包含该指令的下一条指令的地址。

在这一点上，能检查引发异常的指令来进一步查找问题。如，该指令是否访问了没有有效的CPLB定义的存储空间？是否执行了从/到一个未对齐位置的的一个装载/存储操作？指针或者索引寄存器是否指向了一个无效的内存区域？

在引发硬件错误或异常的指令附近设置断点，然后单步执行代码，并观察寄存器的地址(Ix 或Px)。在关键指令前设置断点和/或单步执行整个程序，有时会改变问题的状态(如，该问题不会在这些条件下被观察到)。在这些类似的情况下，断点可以放在关键指令的后面，然后当碰到断点时就可以检测到处理器的状态。注意,处理器将向量化事件处理程序，所以要在事件处理程序的第一条指令设置断点(异常处理程序或硬件错误处理程序)。

使用跟踪缓冲器

Blackfin处理器中存在一个16空隙的跟踪缓冲器，允许捕获到最多16个非连接的流改变(零耗硬件循环除外)。跟踪缓冲器中的信息能用来确定产生一个问题的原因，或者更重要的是，能缩小产生问题的范围，以至于能得到一个一直出现异常行为的小的测试案例。在前面段中，描述了确定引发特别事件的指令的方法。然而，在许多情况下，相同的指令在孤立环境中不会出现问题。该情况发生在取和执行指令(一些情况不会到执行阶段)之前，是找到根源起因的关键。例如，考虑一个使用P2寄存器执行内存装载的指令，在执行这条指令前，立即引发一个中断，而该中断采用的编程，由于不好的编程习惯,没有保存和恢复所用寄存器。ISR代码修改指针寄存器P2，在从中断服务程序返回后，执行原始的内存装载指令。然而，P2不再指向期望的内存位置，因为它已经在一个异步事件中被重写了，这些异步事件可能导致任何一个前面讨论的事件发生。更严重的是，向/从错误的存储位置写/读数据。后者的错误更难发现。

跟踪缓冲器允许问题出现前发生流的改变，并且能容易地通过一个窗口观察到。它记录最多16对不连续。每对中的第一个入口是不连续的源(如，一个调用指令)，第二个入口是不连续的目的地，或目标(如，被调用函数的第一条指令)。在上面的P2例子中，一个给定跟踪对的第一条指令是中断指令的返回(RTI)，而该对的第二个入口是装载指令或者它之前的一个指令。因为跟踪缓冲器也显示不连续的地址，ISR中可以检查RTI指令的地址来发现P2是经过修改的，并且在退出ISR之前绝不会恢复。ISR是应用程序所使用的一个RTOS的一个调度器的一部分。当然，这里所讨论的例子是很简单的，它能证明ISR没有执行一个已知问题的工作区。

有时也会出现没有明显现象的情况(例如，采用所有这些分析，问题发生的原因还是不能得到解释)。在碰到问题前了解发生的转变能帮助创建

一个小的测试案例，该案例能帮助支持团队快速地研究和解决问题。

图1显示了在一个跟踪缓冲器中的条目是如何组织的。最左边一列列出了从0到31的周期数，周期0和1是记录在跟踪缓冲器中的最后一对不连续处，周期2和3是倒数第二对，等等。从左边数的第二列显示了这些对的分组。例如，周期0和1是第15对(0xf)，周期2和3是第14对(0xe)，周期0x1e和0x1f是第0对(0x0)。每对中的第一条指令是不连续的源，第二条指令是不连续的目的地。如对于0xf，周期0是源地址(RTS指令)，周期1是目的指令(CALL Initialize_3VDKFv)，即从在地址0xffa086be结束的子程序返回后，首先执行该指令。

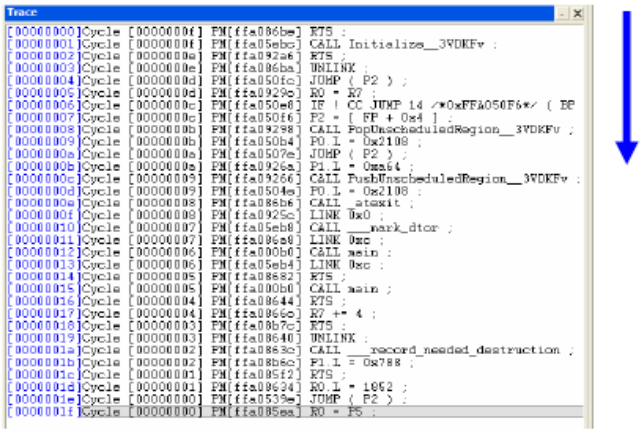

图1 跟踪缓冲器实例

使用断点

这一段描述了软件断点、嵌入断点和硬件断点的差异，并解释了什么时候和怎样使用它们。

软件断点

软件断点方便，容易使用。在IDDE的编辑(源)窗口或反汇编窗口中的一条指令上简单的双击就可以设置断点，当执行到那行代码时暂停下来。然而，在这些事件的后面，放置断点的位置值是“隐藏”在仿真器中的。仿真器读取断点位置的内存值，并将它保存到仿真器的内部断点列表。当应用程序运行时，在那个位置放置一个捕获指令。当碰到任一个断点，或任一个中断事件发生，则断点位置的捕获指令将由先前“隐藏的”指令代替。无疑地，这要求软件断点性质上是可插入的。因此，当软件断点用于诊断问题时，许多遇到的问题似乎不存在了，因为由于软件断点的性质，应用程序的时序已经改变了。图2显示了VisualDSP++ IDDE会话中的一个软件断点。

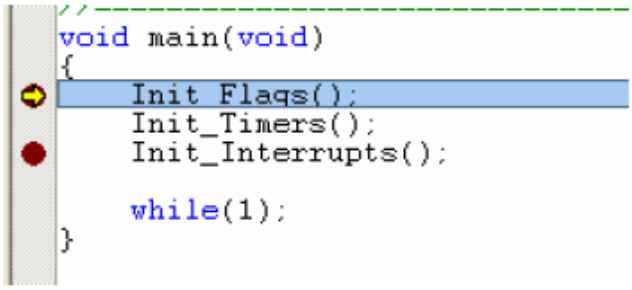

图2 软件断点实例

嵌入断点

一个嵌入断点是应用程序代码的一部分。它类似于软件断点，不同的是调试器不需要查询一个断点列表或插入“暂停”操作码到应用程序中。因此，这类断点是半非插入的。emuexcpt指令促使处理器执行时暂停，该指令仅当仿真器连接时才有意义，否则，它的功能类似于NOP。在事件处理程序中使用嵌入断点是好的习惯，因为除了使用代码空间，它们不影响应用程序的时序，因此允许处理器状态在事件发生后不久就会被观察到。与跟踪缓冲器信息结合使用，现在您就能观

察到处理器的状态，也能观察到在一个事件发生前所发生的转变。图3显示了一个嵌入断点的实例。

```
EX_INTERRUPT_HANDLER(Timer0_ISR)
{
    asm('EMUEXCPT;');  // embedded breakpoint

    // confirm interrupt handling
    *pTIMER_STATUS = 0x0001;
    brkptcounter++; // hw breakpoint counter
    // shift old LED pattern by one
    if(sLight_Move_Direction)
    {
        if((ucActive_LED = ucActive_LED >> 1) <= 0x0020) ucActive_LED = 0x1000;
    }
    else
    {
        if((ucActive_LED = ucActive_LED << 1) == 0x1000) ucActive_LED = 0x0020;
    }

    // write new LED pattern to PORTF LEDs
    *pPORTFIO_TOGGLE = ucActive_LED;
}
```

图3 嵌入断点的实例

硬件断点

另一方面，硬件断点是完全非插入的，因为它们不通过任何方式改变应用程序代码。相反地，硬件断点依赖于芯片的物理硬件逻辑，来监视指令和数据总线。在Blackfin处理器，硬件断点由观察点寄存器单元执行，共有6个指令观察点寄存器和两个数据观察点寄存器。指令硬件断点能在6个特殊的指令地址或3个指令地址范围设置，而数据硬件断点能在2个特殊的数据地址或一个数据地址范围设置。硬件断点能用于RAM或ROM类型的存储空间。

为了在VisualDSP++IDDE中使能硬件断点，在Setting中选择Hardware Breakpoints。图4显示了Hardware Breakpoints窗口的一个指令页。

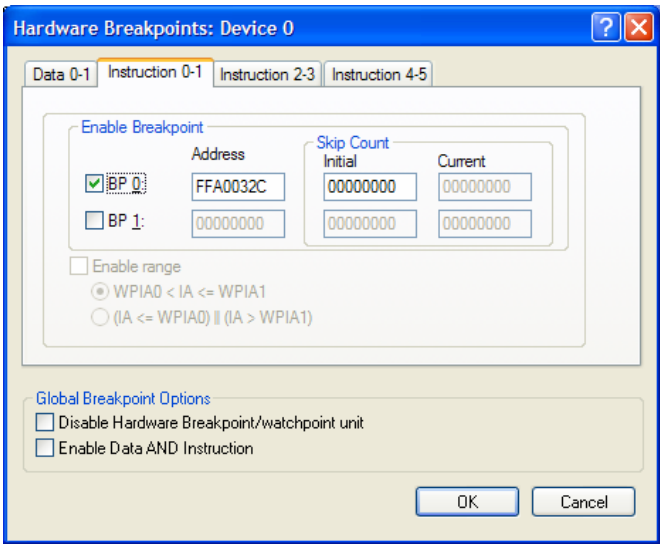

图4 硬件断点(指令)

定义指令地址或地址范围，使得当这些指令即将执行时，暂停处理器的运行。

对于数据访问，必须定义访问类型(读，写，或读/写)来触发一个仿真暂停。图5显示了Hardware Breakpoints的数据页。

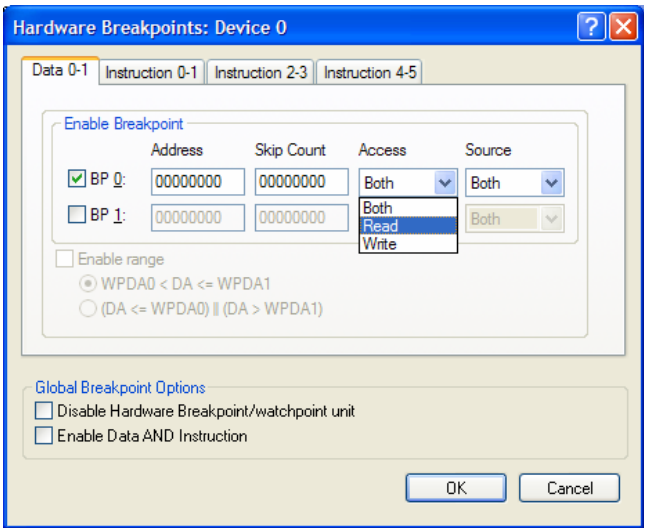

图5 硬件断点(数据)

然后代码开始运行，如果片内指令/数据地址总线与定义在硬件断点寄存器中的地址匹配时，处理器将暂停代码的执行。

硬件断点提供一个跳跃的计数特性，能用于指出在处理器暂停前，可以忽略多少次对特殊区域的访问。例如，如果跳跃计数值设置为0xA，则处理器在第10次地址匹配时才暂停。

VisualDSP++内核(VDK)

VDK是一个实时内核，它简化了带有多个任务的工程的管理。然而，它增加了应用程序的抽象水平。因此，跟任何RTOS一样，它使得准确地找到系统中的错误更加困难。

VisualDSP++有一个内核相关的调试器，能显示系统性能的详细信息，因此能帮助调整应用程序和调试基于RTOS的系统。它允许在任意给定的时间可视化不同的线程(如运行，阻塞，就绪等)。在其他的调试需求中，它能被用于确认一个特别的线程为什么没有运行。图6显示VDK状态历史窗口。

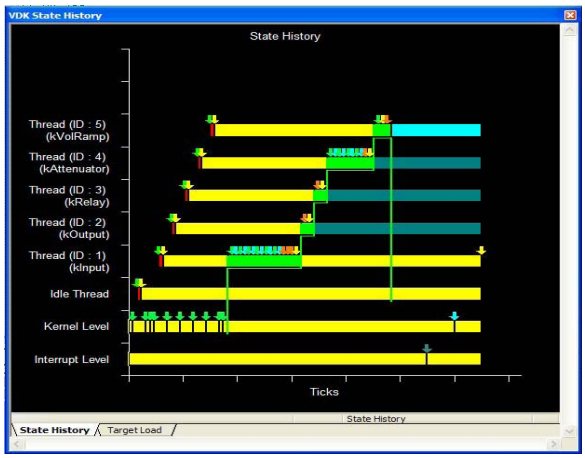

图6　VDK状态历史窗口

注意，确保合适地设置线程的优先级，您应该知道每一个任务的运行时间需求。VDK状态历史窗口能用于确认整体的线程时间消耗情况。

另一个有用的调试窗口是VDK状态窗口，它显示了一个内核恐慌错误的原因。图7显示VDK状态窗口。

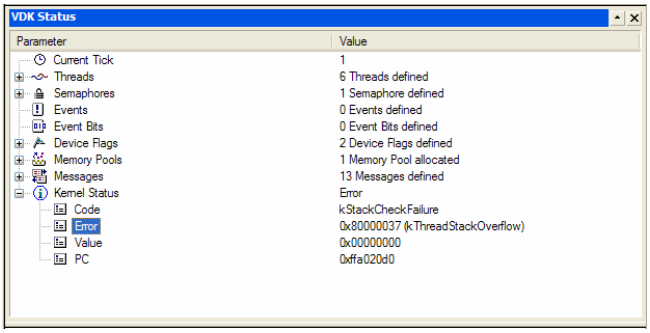

图7　VDK状态窗口

图7中的实例显示了一个栈溢出，从而触发了内核恐慌(kernel panic)。显示的Value值表明了哪一个线程(IDLE线程)的栈大小不够。

Cache 相关的问题

当怀疑出现了一个cache问题时，首先查阅适当的处理器异常列表，确认观察到的特别行为是否在列表中。

如果异常行为的表现跟已知一些问题不相关，则尝试将重要区域移到L1内存的方法排除cache控制器的原因，而本文档前面已经描述了怎样鉴别这个区域。首先打开cache，运行程序，然后关掉cache，再次运行相同的程序，观察前后两次运行的每一个行为差异。如果cache关掉后问题

还是存在，则表明在软件程序上存在问题。关掉cache可能改变其余程序的时序，从而初始程序没有出现问题。因此，尝试将重要区域放到L1中，并打开cache，如果问题依然存在，并且在L1中的代码区域产生了一个异常和/或硬件错误，则可判断该问题不是完全由cache引起的。

如果存在异常，参阅上文中的硬件错误和软件异常。

Cache 一致性

Blackfin处理器没有提供维护cache缓存和主存之间一致性的机制。典型地,当一个外设DMA通道访问一个定义为可缓存的片外区域时，一致性是这个系统中的问题。Cache控制器不知道这些DMA访问，结果，可能使用该区域的旧数据参与运算，因此出现了不期望的结果。这时可采用软件编程来维护一致性，这可通过使那些被DMA控制器访问的行无效来实现。

中断相关的问题

中断服务程序中，确保资源按照正确的顺序进出栈，也要注意RETI指令的进出栈的重要性。当RETI被压入栈时，使能中断嵌套；相反，RETI出栈则消除中断嵌套。因此，如果更高优先级的中断不应该中断中断服务程序时，就不要放RETI到栈中。如果采用C/C++编程，使用非嵌套的中断处理程序：

```
EX_INTERRUPT_HANDLER(Timer_handler)
```

如果特殊的ISR的中断嵌套应该使能时，用下面的中断处理程序：

```
EX_REENTRANT_HANDLER(Timer_handler)
```

这个可重入的处理程序在ISR的开头放RETI，并在最后，正好执行RTI指令前退出栈。

为了避免反复的重定向到相同的ISR，退出前要清除ISR中的中断起因。例如，对于内核定时器，清除它的控制寄存器中的TINT(定时器中断)位将消除该中断。

当使用嵌套中断时，要避免使用共享资源所带来的问题。可以缩短一个ISR的执行时间，使得更低优先级的ISRs能及时的得到执行；保持短的ISRs执行时间也能减少ISR中使用的资源数量，因此减轻了栈的使用负荷。另一种嵌套中断引出的问题是栈溢出，发现嵌套中断(或者深度嵌套子程序)中栈溢出的一种方法是，在开始时读取每个ISR的栈指针(SP)，来检测它是否靠近栈的末端。

总结

这篇电子工程文档描述了VDK工具以及Blackfin处理器特性，它们能用来帮助缩小问题产生的原因范围。

首先，检测处理器芯片版本的异常列表，用于确认是否这个问题已经列出，如果是，则执行已有的工作区环境。为了得到已知芯片的自动化的软件支持，确认使用最新的工具，并且使能芯片工作环境。

应用程在运行主程序之前，应该安装事件处理程序(异常处理程序，中断处理程序)，使得需要的时候可以捕捉到事件。

查证异常行为的原因。什么地方没有正确地工作？产生了异常/硬件错误？是什么类型的异常和/或硬件错误？附录A中的表可以帮助查询。外设发生溢出/下溢了吗？产生了DMA错误吗？

寻找增加可重复性的方法。虽然不是必须的，增加一个问题发生的频率可以增强定位一个问题的可能性。增加可重复性可能意味着增加或者缩

短了循环的迭代次数，改变了内核电压，调整了内核和/或系统频率等等。注意一次仅改变一个变量。如果修改的变量对出现的错误没有效果，则在其他新的修改之前，恢复该变量的值。

在错误触发前，使用软件断点观察处理器的状态。如果插入软件断点时程序运行失败没有发生，则或者使用嵌入断点，或者最后使用硬件断点来跟踪处理器。

如果硬件错误/异常出现了，从序列状态寄存器中找出相应的原因，查阅附录A中的表，找出引发这些事件的起因。使用嵌入断点或硬件断点，在各自的异常处理程序中捕捉事件。

在问题出现前使用跟踪窗口观察处理器的转变情况。

通过选择VisualDSP++中的Register->Save Registers，如图8所示，保存所有的寄存器，用于上电时的启动分析。

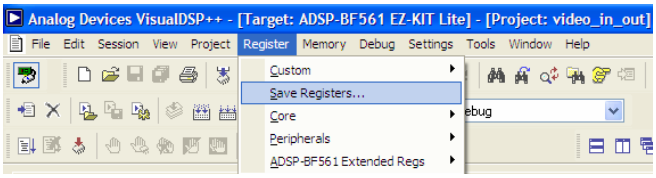

图8　保存寄存器

如果在进行了上面讨论的所有步骤后，错误仍然没有得到解决，则利用触发异常行为的事件序列，来产生一个小的测试案例。一旦有了这个测试案例，则将它与您的问题总结出来，交给嵌入式处理器支持团队。他们将根据你的资料再现你的问题，最后使问题得到解决。

附录 A

表1 产生异常的事件

异常	EXCAUSE [5..0]	类型：错误(E) 服务(S)	备注/举例
带4位段m的强制异常指令EXCPT	m-field	S	指令提供EXCAUSE的4个位
单步	0x10	S	当处理器处于单步模式时每执行一条指令都产生异常，主要用于测试
异常导致仿真跟踪缓冲溢出	0x11	S	当仿真跟踪缓冲溢出时处理器捕获这个异常(只有被跟踪单元控制寄存器使能时)
未定义的指令	0x21	E	可能用于仿真对于特定处理器没有定义的指令
非法的指令组合	0x22	E	*参考ADSP-BF53x/BF56x Blackfin 处理器编程*
数据访问CPLB保护违例	0x23	E	试图读或写监控资源，或非法的数据内存访问，监控资源指为监控用户保留的寄存器和指令：仅监控用户寄存器、所有MMR、监控用户指令等(使用DAG同时对两个MMR进行双访问会产生这个异常)，另外，这个异常入口还标志出访问被禁止的内存产生的保护违例，并由内存管理单元的CPLB定义
数据访问时地址未对齐违例	0x24	E	试图访问地址未对齐的数据存储器或数据缓存
不可恢复的事件	0x25	E	例如在处理前一个异常时又产生一个异常
数据访问时CPLB丢失	0x26	E	MMU用来标识数据访问时出现CPLB丢失
数据访问选中多个CPLB	0x27	E	多个CPLB匹配取数地址
由仿真观测点产生的异常	0x28	E	到达仿真观测点指令地址控制寄存器(WPIACTL)的EMUSW位中的1位被置位

取指时地址未对齐	0x2A	E	试图进行地址未对齐的指令缓冲访问，在这个异常中，有RETX返回的地址是没有对齐的目的地址而不是非法指令的地址。例如，如果间接跳转到P0中保存的一个未对齐地址，RETX返回的值等于P0而不是返回跳转指令的地址(注意这个异常不会发生在PC相当跳转，只会发生在间接跳转中)
取指时CPLB保护违例	0x2B	E	非法取值访问(内存保护违例)
取值时CPLB丢失	0x2C	E	取值时CPLB丢失
取值时多个CPLB被选中	0x2D	E	多个CPLB入口匹配取值地址
非法使用监控资源	0x2E	E	用户模式时试图使用监控寄存器或指令。监控资源是为监控保留的寄存器和指令：仅监控用户寄存器、所有MMR、监控用户指令

表2 导致硬件错误中断的硬件条件

硬件条件	HWERRCAUSE (Hexadecimal)	备注/举例
系统MMR错误	0x02	如果访问了一个无效的系统MMR地址，则可能产生错误，如一个16位指令访问了一个32位寄存器，或者一个32位指令访问了一个16位地址
外存寻址错误	0x03	
性能监视器溢出	0x12	
RAISE 5指令	0x18	软件指令RAISE 5 引起硬件错误中断(IVHW)
预留	所有其它位的组合	

参考文献

[1] *ADSP-BF533 Blackfin Booting Process (EE-240)*. Rev 3. January 2005. Analog Devices, Inc.

[2] *ADSP-BF53x/ADSP-BF56x Programming Reference.* Rev 1. May 2005. Analog Devices, Inc.

文档记录

Revision	**Description**
Rev 1 – December 11, 2006 *by J. Manguane*	Initial Release.