



Technical notes on using Analog Devices DSPs, processors and development tools
 Visit our Web resources <http://www.analog.com/ee-notes> and <http://www.analog.com/processors> or
 e-mail processor.support@analog.com or processor.tools.support@analog.com for technical support.

Blackfin® Processor Troubleshooting Tips Using VisualDSP++® Tools

Contributed by Jorge Manguane

Rev 1 – December 11, 2006

Introduction

This document describes debug features of Blackfin® processors and the VisualDSP++® development tools. By applying the tips below, programmers can narrow down the cause of an encountered problem before reporting it to the Analog Devices embedded processor support team. This will help to solve the issue quickly.

Debugging Tips and Tricks

The following topics are covered:

- Running an application via an emulator versus booting the application. Specifically, SDRAM initialization considerations are addressed.
- Unlocking core B for dual-core processors. The emulator does this automatically; however, when code is booted in, core B must be unlocked manually.
- Hardware errors and software exceptions
- Blackfin processor debug features and tools, including:
 - Trace buffer
 - Breakpoints (software, embedded, and hardware)
 - VDK debugging (VDK Status window and VDK State History window)
- Debugging issues when cache is enabled
- Interrupts



This EE-Note does not cover issues associated with troubleshooting peripherals.

Emulation vs. Standalone Booting

The emulator software uses .xml files to configure resources such as SDRAM timing on an EZ-KIT Lite® evaluation board. There are default .xml files for all Blackfin processors for which an evaluation platform exists (i.e., the EZ-KIT Lite boards), where definitions are created to initialize certain registers upon emulator connection.

For example, the following is an excerpt from the ADSP-BF537-proc.xml file for the ADSP-BF537 Blackfin processor:

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

Thus, when developing an application for the ADSP-BF537 EZ-KIT Lite board, the SDRAM is initialized automatically when the emulator software is invoked.

However, when moving to a standalone application (that is, booting the application rather than downloading it using an emulator), it is the

user's responsibility to enable the SDRAM controller if SDRAM memory is used in the system. This is accomplished by including an initialization file via the Loader page of the Project Options dialog box page when creating a loader file. See *ADSP-BF533 Blackfin Booting Process (EE-240)*^[1].

Another difference that can cause issues when moving an application from an emulator session to booting standalone is related to ADSP-BF561 Blackfin dual-core systems. By default, the emulator software “unlocks” core B and allows it to run from the beginning of L1 instruction memory. When both cores are to be used, core B must be unlocked by core A by clearing bit 5 of the system reset configuration register (SICA_SYSCR).

Another issue that sometimes occurs as a result of the emulator unlocking core B is with regard to changing operating modes or clock frequencies. Specifically, core B must be in the idle state (not simply at a breakpoint) when making changes to the PLL or voltage regulator. This can cause trouble if, for example, a breakpoint is set in core B and code is run on core A that changes the PLL frequency. Be sure to run code that places core B into the idle state prior to changing the PLL frequency. This can be done using supplemental interrupts or GPIO pins.

Hardware Errors and Software Exceptions

Hardware errors and software exceptions are two specific types of events that can occur on Blackfin processors. Each of these events has a separate entry in the event vector table (EVT). Handlers should be installed for each of these events so that they can be trapped by using any of the breakpoint methods described later in this document. At that point, the state of the processor can be examined to understand what caused the particular event. The sequencer status register (SEQSTAT) has two fields that can be used to gain more insight into the offending

condition. The HWERRCAUSE field is used to identify the condition that generated a hardware error, and the EXCAUSE field is used to identify the condition that generated an exception.

A hardware error can be generated for a variety of reasons, such as when an MMR is accessed with the wrong word size (e.g., a 16-bit MMR is accessed as 32-bit, or vice versa) or when the core or DMA controller tries to access reserved or uninitialized memory space. The RETI address will contain an address within 10 locations of the offending location. If hardware errors are enabled and the event is serviced, the condition will clear, but the hardware error cause will remain with the last error condition.

For ADSP-BF561 Blackfin dual-core processors, hardware errors generated by a specific core will only cause an error on that core. If the DMA controller generates a hardware error, the error is sent to both cores.

In each of the handlers (hardware error or exception) the HWERRCAUSE and EXCAUSE fields can be read to identify the cause of the event. Alternatively, when using an emulator to debug, trap instructions can be placed in the handlers, such as `emuxcept`, so that the processor will stop running whenever a hardware error and/or an exception occurs. The appropriate field in the SEQSTAT register can then be inspected to determine the cause of the event.

Now that the cause of the event is known, the address of the offending instruction should be noted to determine when the problem occurred in the program. For exceptions, the return from exception register (RETX) contains the address of the “offending” instruction or the address of the next instruction to be executed. The address in RETX depends on the type of exception: service (S) or error (E). The *ADSP-BF53x/BF56x Blackfin Processor Programming Reference*^[2] lists the events that cause exceptions, as well as their type (service or error). For convenience, this table is included in [Appendix A](#).

For error type exceptions, `RETX` holds the address of the offending instruction; for service type exceptions, `RETX` holds the address of the next instruction after that of the offending instruction.

At this point, the offending instruction can be examined for further insight into the problem. Is the instruction accessing memory which has no valid `CPLB` definition? Is the instruction performing a memory load/store from/to a misaligned location? Does the pointer or index register point to an invalid memory region?

A breakpoint can be set in the vicinity of the instruction that caused the hardware error or exception, and the code can be single-stepped while observing the address registers (`IX` or `PX`). Setting breakpoints and/or stepping through the code prior to the instruction of interest sometimes changes the behavior of the problem (i.e., the problem can no longer be observed under these conditions). In cases like this, the breakpoint can be placed past the instruction of interest and the state of the processor can be examined when the breakpoint is encountered. Note that the processor will have vectored off to the event handler, so the breakpoint is placed at the first instruction in the event handler (exception handler or hardware error handler).

Using the Trace Buffer

A 16-slot trace buffer, available on Blackfin processors, allows the last 16 non-contiguous changes of flow (excluding zero-overhead hardware loops) to be captured. The information in the trace buffer can be instrumental in determining the cause of a problem or, more importantly, in narrowing down the problem such that a small test case can be derived that consistently shows the unexpected behavior. In the previous section, methods to identify the instruction which causes a particular event were described; however, in many cases, that same instruction in isolation would not exhibit the problem. It is what happens before the instruction is fetched and executed (in some

cases it does not make it to the execution stage) that is critical in zeroing in on the root cause. For example, consider an instruction that performs a memory load using the `P2` register. Immediately before executing this instruction, an interrupt is taken, which, through bad programming practice, does not save and restore the registers it uses. The ISR code modifies the pointer register `P2` and, upon returning from the interrupt service routine, the original memory load instruction is executed. However, `P2` no longer points to the desired memory location because it was overwritten during an asynchronous event, which could result in any one of the events previously discussed. More subtly, the data is written/read to/from the wrong memory. The latter is typically harder to detect.

The trace buffer allows changes of flow that took place before the problem occurred to be seen easily in a window. It records the last 16 pairs of discontinuities. The first entry in the pair is the source of the discontinuity (i.e., a `call` instruction), and the second entry is the destination, or target, of the discontinuity (i.e., the first instruction of the called function). In the above `P2` example, the first instruction of a given trace pair would be a return from interrupt instruction (`RTI`), and the second entry in the pair would be the load instruction or an instruction before it. Since the trace buffer also shows the addresses of the discontinuities, the address of the `RTI` instruction can be inspected in the ISR to find that `P2` was modified and never restored before exiting the ISR. This ISR could be part of a scheduler of an RTOS that the application uses. Of course, the example discussed here is very simplistic. It could turn out that the ISR did not implement a workaround to a known issue.

There will be times when nothing is apparent (i.e., with all this analysis, why the problem occurs still cannot be explained). Knowing the transitions that took place before the problem was encountered can help create a small test case, which can be very helpful to the support

team to quickly investigate and resolve the problem.

Figure 1 shows how entries are structured in a trace buffer. The leftmost column lists the cycles from 0 to 31. Cycles 0 and 1 are the last pair of discontinuities recorded in the trace buffer, cycles 2 and 3 are the penultimate pair, etc. The second column from the left shows the grouping of the pairs. For example, cycles 0 and 1 are the 15th pair (0xf), cycles 2 and 3 are the 14th pair (0xe), and cycles 0x1e and 0x1f are the zero pair (0x0). The first instruction of the pair is the source of the discontinuity, and the second instruction is the destination of the discontinuity. For the 0xf pair, cycle 0 is the source address (the RTS instruction), and cycle 1 is the destination instruction (CALL Initialize__3VDKFv). That is, this instruction is executed first after returning from the subroutine ending at address 0xffa086be.

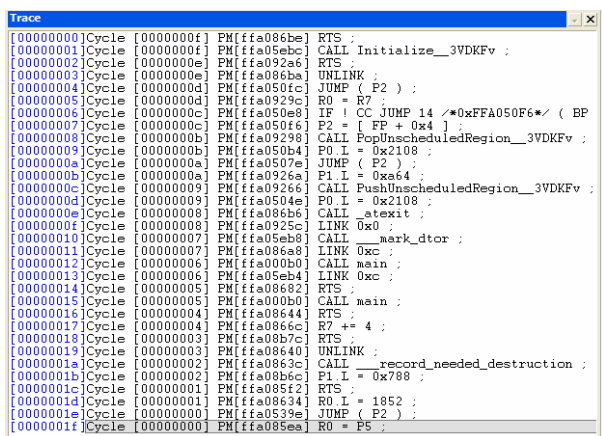


Figure 1. Trace Buffer Example

Using Breakpoints

This section describes differences between software, embedded, and hardware breakpoints, and explains when and how to use them.

Software Breakpoints

Software breakpoints are convenient and easy to use. Simply double-click on an instruction in the editor (source) window or the Disassembly window of the IDDE to set the breakpoint, and

execution halts when that line of code is hit. However, behind the scenes, the value of the location where the breakpoint is placed is ‘cached’ within the emulator. The emulator reads the memory at the breakpoint location and saves it to the emulator’s internal breakpoint list. When the application is run, it places a trap instruction at that location. When any breakpoint is hit, or any halt event occurs, the trap instruction at the breakpoint location is replaced by the instruction that was previously ‘cached’. Clearly, this suggests that software breakpoints are intrusive in nature. Thus, many encountered problems may seem to go away when software breakpoints are used to diagnose the problem because the timing of the application has changed due to the nature of software breakpoints. Figure 2 shows what a software breakpoint looks like in a VisualDSP++ IDDE session.

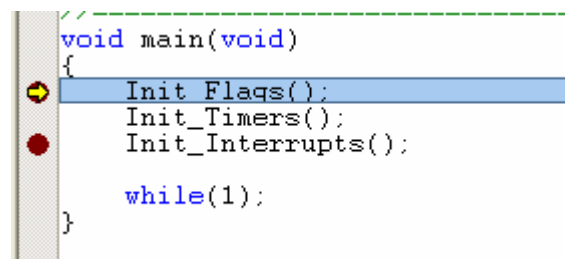


Figure 2. Software Breakpoint Example

Embedded Breakpoints

An embedded breakpoint is part of the application code itself. It is similar to software breakpoints, except that the debugger does not need to look up a breakpoint table list or insert ‘halting’ op-codes into the application. As such, this class of breakpoints is quasi-non-intrusive. The `emuexcept` instruction causes the processor to halt when executed. This instruction only has meaning when an emulator is connected; otherwise, it acts as a NOP. It is good practice to use embedded breakpoints inside event handlers because, other than using code space, they do not affect the timing of the application, thus allowing for the state of the processor to be observed confidently soon after the event occurred. Used in conjunction with the trace buffer information,

now you can observe the state of the processor and can also observe the transitions that took place right before an event occurred. Figure 3 shows an example of an embedded breakpoint.

```

EX_INTERRUPT_HANDLER(Timer0_ISR)
{
    asm("EMUEXCPT;"); // embedded breakpoint
    // confirm interrupt handling
    *pTIMER_STATUS = 0x0001;
    brkptcounter++; // hv 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;
}
    
```

Figure 3. Embedded Breakpoint Example

Hardware Breakpoints

On the other hand, hardware breakpoints are completely non-intrusive, as they do not alter the application code in any way. Instead, hardware breakpoints rely on physical hardware logic on the chip, which monitors both the instruction and data buses. On Blackfin processors, hardware breakpoints are implemented by the watchpoint register unit. There are six instruction watchpoint registers and two data watchpoint registers. Instruction hardware breakpoints can be set on six specific instruction addresses or on three instruction address ranges. Data hardware breakpoints can be set on two specific data addresses or on one data address range. Hardware breakpoints can be used in RAM or ROM types of memory.

To enable hardware breakpoints from within the VisualDSP++ IDDE, go to Settings and select Hardware Breakpoints. Figure 4 shows one of the Instruction pages of the Hardware Breakpoints window.

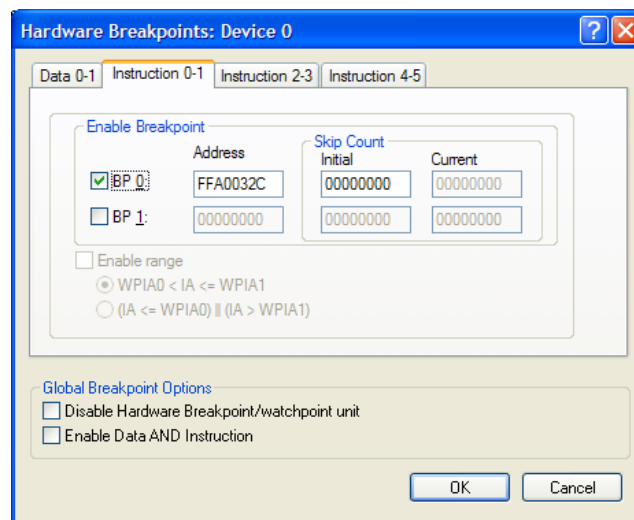


Figure 4. Hardware Breakpoints (Instruction)

The instruction address or address range can then be specified to force halts to the processor when these instructions are about to be executed.

For data accesses, the type of access must be specified (read, write, or both), to trigger an emulation halt. Figure 5 shows the Data page of the Hardware Breakpoints window.

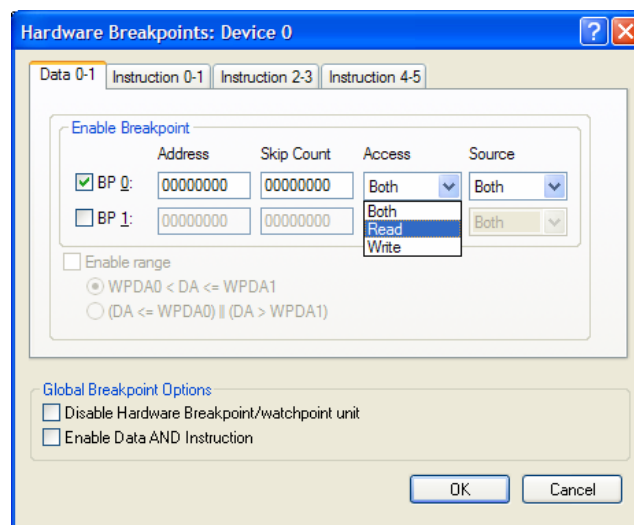


Figure 5. Hardware Breakpoints (Data)

The code can then be run, and the processor will halt if a match is seen between the internal instruction/data address bus and the address specified in the hardware breakpoint registers.

Hardware breakpoints provide a skip count feature, which can be used to indicate how many times to ignore accesses to the specified region before the processor is halted. For example, if skip count is set to 0xA, the processor will halt on the 10th occurrence of the address match.

VisualDSP++ Kernel (VDK)

VDK is a real-time kernel that simplifies the management of projects with multiple tasks. However, it adds a level of abstraction to the application. Due to this, just like any RTOS, it makes it that much harder to pinpoint bugs in the system.

VisualDSP++ has a kernel-aware debugger that can display system performance details, which can help in application tuning and in debugging an RTOS-based system. It allows you to visualize the various threads at any given time (i.e., running, blocked, ready, etc.). Among other debug needs, this may be instrumental in identifying why a particular thread never gets to run. Figure 6 shows the VDK State History window.

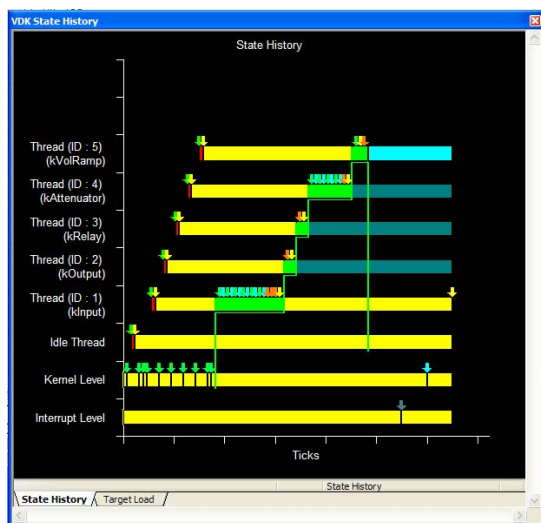


Figure 6. VDK State History Window



Ensure that thread priorities are set appropriately. You should know each task's needs in terms of run time. The VDK State History window can be instrumental in identifying overall thread time balancing.

Another useful debug window, the VDK Status window, displays the cause of a kernel panic error. Figure 7 shows the VDK Status window.

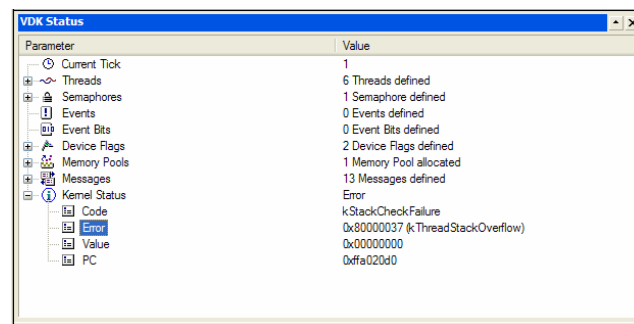


Figure 7. VDK Status Window

The example in Figure 7 shows a stack overflow, which caused the kernel panic. The indicated value identifies the thread (IDLE thread) whose stack size is insufficient.

Cache-Related Issues

When a cache problem is suspected, first consult the appropriate processor anomaly list to verify whether the particular observed behavior has been characterized there.

If the unexpected behavior does not appear to be related to a known issue, try to rule out the cache controller as the cause by moving the region of interest into L1 memory. The previous sections showed how to identify this region. Run the application with cache turned on, and then run it a second time with cache turned off. Observe any differences in behavior. If the problem is still present with the cache turned off, it could indicate a race condition in the software application. Turning cache off may change the timing for the rest of the application, causing the failure to stop occurring. Because of this, try

bringing the region of interest into L1 and leave the cache turned on. If the problem persists and an exception and/or hardware error is generated in the region of the code that is in L1, it is not a cache integrity issue.

If exceptions are present, refer to [Hardware Errors and Software Exceptions](#) above.

Cache Coherency

Blackfin processors do not maintain coherency between cache memory and main memory. Typically, coherency will be an issue in systems where a peripheral DMA channel accesses a region of external memory defined as cacheable. The cache controller has no knowledge of these accesses and, as a result, may use its stale (old) data for computations, thus producing unexpected results. Software must ensure that coherency is maintained by invalidating lines that might have been accessed by a DMA controller.

Interrupt-Related Issues

In the ISR, ensure that the pushing and popping of resources is done in the correct order. Also, note the significance of pushing and popping `RETI`. When `RETI` is pushed onto the stack, interrupt nesting is enabled; conversely, popping `RETI` disables interrupt nesting. So, if higher priority interrupts should not interrupt the interrupt service routine, do not push `RETI` onto the stack. If programming in C/C++, use the non-nested interrupt handler:

```
EX_INTERRUPT_HANDLER(Timer_handler)
```

If interrupt nesting should be enabled for a particular ISR, use the following interrupt handler:

```
EX_REENTRANT_HANDLER(Timer_handler)
```

This reentrant handler pushes `RETI` at the beginning of the ISR and pops it at the end, right before the `RTI` is executed.

To prevent repeated vectoring to the same ISR, clear the cause of the interrupt in the ISR before

exiting. For example, for a core timer, clearing the `TINT` (timer interrupt) bit in the core timer control register will clear the interrupt.

When using nested interrupts, be sure to avoid problems that may arise due to using shared resources. Minimizing an ISR's execution time allows lower-priority ISRs to also be serviced in a timely manner. Keeping ISRs short also reduces the number of used resources within the ISR, thus alleviating stack usage. Another type of problem that arises with nested interrupts is stack overflow. One way to detect stack overflow in nested interrupts (or even in deeply nested subroutines) is to read the stack pointer (SP) at the beginning of every ISR to check if the pointer is nearing the end of the stack.

Summary

This EE-Note describes VDK tools and Blackfin processor features that are available to help narrow down problems.

First and foremost, always check the anomaly list for the silicon revision of the processor used to verify whether the problem is already known. If it is, implement the given workaround. To get automated software support for known silicon errata, ensure that the latest tools are being used and that silicon workarounds are enabled.

Applications should install event handlers (exception handlers, interrupt handlers) before running the main application so you can trap events when necessary.

Verify the behavior. What exactly is not working properly? Are exceptions/hardware errors being generated? If so, what exception and/or hardware error? The tables in Appendix A will help determine this. Are the peripherals overflowing/underflowing? Are DMA errors being generated?

Find ways to increase repeatability. Although not always possible, increasing the frequency with which a problem occurs enhances the chance of

fixing the problem. Increasing repeatability could mean increasing or shortening loop iterations, changing the core voltage, adjusting the core and/or system frequencies, etc. It should be noted that only one variable should be changed at a time. If the modified variable has no effect on the bug, leave the variable as it was before a new modification is made.

Use software breakpoints to observe processor state prior to the triggering of the fault. If the failure stops occurring when software breakpoints are inserted, then either use embedded or, ultimately, hardware breakpoints.

If hardware errors/exceptions are generated, find the respective causes from the sequencer status register and check the tables in Appendix A to see what might have generated those events.

Trap the events in the respective exception handlers by using embedded breakpoints or hardware breakpoints.

Use the Trace window to observe processor transitions prior to the occurrence of the problem.

Save all registers for post analysis by choosing Register->Save Registers in VisualDSP++, as shown in Figure 8.

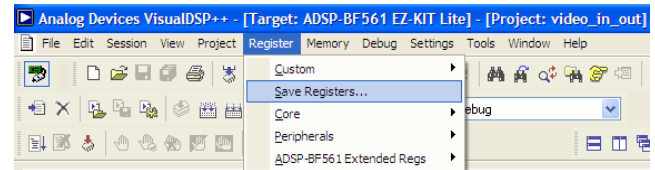


Figure 8. Save Registers Feature

If, after the steps described above, the fault cannot be corrected, having the knowledge of the sequence of events that triggers the unexpected behavior should allow for a small test case to be generated. Once a test case is available, summarize your findings for the embedded processor support team and include the test case. This will allow for rapid reproduction of the issue, which will ultimately help get the issue resolved.

Appendix A

Table 1. Events that Cause Exceptions

Exception	EXCAUSE [5:0]	Type: Error (E) Service (S)	Notes/Examples
Force Exception instruction EXCPT with 4-bit m field	m field	S	Instruction provides 4 bits of EXCAUSE.
Single step	0x10	S	When the processor is in single step mode, every instruction generates an exception. Primarily used for debugging.
Exception caused by a trace buffer full condition	0x11	S	The processor takes this exception when the trace buffer overflows (only when enabled by the Trace Unit Control register).
Undefined instruction	0x21	E	May be used to emulate instructions that are not defined for a particular processor implementation.
Illegal instruction combination	0x22	E	See section for multi-issue rules in the <i>ADSP-BF53x/BF56x Blackfin Processor Programming Reference</i> .
Data access CPLB protection violation	0x23	E	Attempted read or write to Supervisor resource, or illegal data memory access. Supervisor resources are registers and instructions that are reserved for Supervisor use: Supervisor only registers, all MMRs, and Supervisor only instructions. (A simultaneous, dual access to two MMRs using the data address generators generates this type of exception.) In addition, this entry is used to signal a protection violation caused by disallowed memory access, and it is defined by the Memory Management Unit (MMU) cacheability protection lookaside buffer (CPLB).
Data access mis-aligned address violation	0x24	E	Attempted misaligned data memory or data cache access.
Unrecoverable event	0x25	E	For example, an exception generated while processing a previous exception.
Data access CPLB miss	0x26	E	Used by the MMU to signal a CPLB miss on a data access.
Data access multiple CPLB hits	0x27	E	More than one CPLB entry matches data fetch address.
Exception caused by an emulation watch-point match	0x28	E	There is a watchpoint match, and one of the EMUSW bits in the Watchpoint Instruction Address Control (WPIACTL) register is set.

Instruction fetch misaligned address violation	0x2A	E	Attempted misaligned instruction cache fetch. On a misaligned instruction fetch exception, the return address provided in RETX is the destination address which is misaligned, rather than the address of the offending instruction. For example, if an indirect branch to a misaligned address held in P0 is attempted, the return address in RETX is equal to P0, rather than to the address of the branch instruction. (Note this exception can never be generated from PC-relative branches, only from indirect branches.)
Instruction fetch CPLB protection violation	0x2B	E	Illegal instruction fetch access (memory protection violation).
Instruction fetch CPLB miss	0x2C	E	CPLB miss on an instruction fetch.
Instruction fetch multiple CPLB hits	0x2D	E	More than one CPLB entry matches instruction fetch address.
Illegal use of supervisor resource	0x2E	E	Attempted to use a Supervisor register or instruction from User mode. Supervisor resources are registers and instructions that are reserved for Supervisor use: Supervisor only registers, all MMRs, and Supervisor only instructions.

Table 2. Hardware Conditions Causing Hardware Error Interrupts

Hardware Condition	HWERRCAUSE (Hexadecimal)	Notes / Examples
System MMR Error	0x02	An error can occur if an invalid System MMR location is accessed, if a 32-bit register is accessed with a 16-bit instruction, or if a 16-bit register is accessed with a 32-bit instruction.
External Memory Addressing Error	0x03	
Performance Monitor Overflow	0x12	
RAISE 5 instruction	0x18	Software issued a RAISE 5 instruction to invoke the Hardware Error Interrupt (IVHW).
Reserved	All other values.	

References

- [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.

Document History

Revision	Description
<i>Rev 1 – December 11, 2006 by J. Manguane</i>	Initial Release.