Blackfin Embedded Processor

Silicon Anomaly List — ADSP-BF534/BF536/BF537

ABOUT ADSP-BF534/BF536/BF537 SILICON ANOMALIES

These anomalies represent the currently known differences between revisions of the Blackfin® ADSP-BF534/BF536/BF537 product(s) and the functionality specified in the ADSP-BF534/BF536/BF537 data sheet(s) and the Hardware Reference book(s).

SILICON REVISIONS

A silicon revision number with the form "-x.x" is branded on all parts. The implementation field bits <15:0> of the DSPID core MMR register can be used to differentiate the revisions as shown below.

Silicon REVISION	DSPID<15:0>
0.3	0x0003
0.2	0x0002

APPLICABILITY

Each anomaly applies to specific silicon revisions. See Summary or Detailed List for affected revisions. Additionally, not all processors described by this anomaly list have the same feature set. Therefore, peripheral-specific anomalies may not apply to all processors. See the below table for details. An "x" indicates that anomalies related to this peripheral apply only to the model indicated, and the list of specific anomalies for that peripheral appear in the rightmost column.

Peripheral	ADSP-BF534	ADSP-BF536	ADSP-BF537	Anomalies
Ethernet MAC		x	x	05000252, 05000256, 05000285, 05000316, 05000321, 05000322, 05000341

ANOMALY LIST REVISION HISTORY

The following revision history lists the anomaly list revisions and major changes for each anomaly list revision.

Date	Anomaly List Revision	Data Sheet Revision	Additions and Changes
12/21/2015	H	J	Style/consistency updates
03/14/2014	G	I	Added Anomalies - 05000503, 05000506
05/23/2011	F	I	Added Anomalies - 05000489, 05000491, 05000494, 05000501
05/25/2010	E	G	Added Anomalies - 05000443, 05000461, 05000462, 05000473, 05000475, 05000477, 05000480, 05000481
09/18/2008	D	E	Added Anomalies - 05000416, 05000425, 05000426 Revised Anomaly - 05000283, 05000315
02/08/2008	C	D	Added Anomalies - 05000402, 05000403 Removed Anomaly - 05000359
12/10/2007	B	D	Removed Silicon Revision 0.1 Added Anomalies - 05000350, 05000355, 05000366, 05000371 Removed Anomaly - 05000167
09/04/2007	A	D	Initial Consolidated Revision - Replaces anomaly lists for ADSP-BF534 (Rev M), ADSP-BF536 (Rev L) and ADSP-BF537 (Rev M) Added Anomalies - 05000167, 05000341, 05000357, 05000359

NR003531H Document Feedback

One Technology Way, P.O.Box 9106, Norwood, MA 02062-9106 U.S.A.
Tel: 781.329.4700 **Technical Support** **www.analog.com**

SUMMARY OF SILICON ANOMALIES

The following table provides a summary of ADSP-BF534/BF536/BF537 anomalies and the applicable silicon revision(s) for each anomaly.

No.	ID	Description	Rev 0.2	Rev 0.3
1	05000074	Multi-Issue Instruction with dsp32shiftimm in slot1 and P-reg Store in slot2 Not Supported	x	x
2	05000119	DMA_RUN Bit Is Not Valid after a Peripheral Receive Channel DMA Stops	x	x
3	05000122	Rx.H Cannot Be Used to Access 16-bit System MMR Registers	x	x
4	05000180	PPI_DELAY Not Functional in PPI Modes with 0 Frame Syncs	x	x
5	05000244	If I-Cache Is On, CSYNC/SSYNC/IDLE Around Change of Control Causes Failures	x	.
6	05000245	False Hardware Error from an Access in the Shadow of a Conditional Branch	x	x
7	05000250	Incorrect Bit Shift of Data Word in Multichannel (TDM) Mode in Certain Conditions	x	.
8	05000252	EMAC TX DMA Error After an Early Frame Abort	x	.
9	05000253	Maximum External Clock Speed for Timers	x	.
10	05000254	Incorrect Timer Pulse Width in Single-Shot PWM_OUT Mode with External Clock	.	x
11	05000255	Entering Hibernate State with RTC Seconds Interrupt Not Functional	x	.
12	05000256	EMAC MDIO Input Latched on Wrong MDC Edge	x	.
13	05000257	Interrupt/Exception During Short Hardware Loop May Cause Bad Instruction Fetches	x	.
14	05000258	Instruction Cache Is Corrupted When Bits 9 and 12 of the ICPLB Data Registers Differ	x	.
15	05000260	ICPLB_STATUS MMR Register May Be Corrupted	x	.
16	05000261	DCPLB_FAULT_ADDR MMR Register May Be Corrupted	x	.
17	05000262	Stores To Data Cache May Be Lost	x	.
18	05000263	Hardware Loop Corrupted When Taking an ICPLB Exception	x	.
19	05000264	CSYNC/SSYNC/IDLE Causes Infinite Stall in Penultimate Instruction in Hardware Loop	x	.
20	05000265	Sensitivity To Noise with Slow Input Edge Rates on External SPORT TX and RX Clocks	x	x
21	05000268	Memory DMA Error when Peripheral DMA Is Running with Non-Zero DEB_TRAFFIC_PERIOD	x	.
22	05000270	High I/O Activity Causes Output Voltage of Internal Voltage Regulator (Vddint) to Decrease	x	.
23	05000272	Certain Data Cache Writethrough Modes Fail for Vddint <= 0.9V	x	x
24	05000273	Writes to Synchronous SDRAM Memory May Be Lost	x	.
25	05000277	Writes to an I/O Data Register One SCLK Cycle after an Edge Is Detected May Clear Interrupt	x	.
26	05000278	Disabling Peripherals with DMA Running May Cause DMA System Instability	x	.
27	05000280	SPI Master Boot Mode Does Not Work Well with Atmel Data Flash Devices	x	x
28	05000281	False Hardware Error when ISR Context Is Not Restored	x	.
29	05000282	Memory DMA Corruption with 32-Bit Data and Traffic Control	x	.
30	05000283	System MMR Write Is Stalled Indefinitely when Killed in a Particular Stage	x	.
31	05000285	TXDWA Bit in EMAC_SYSCTL Register Is Not Functional	x	.
32	05000288	SPORTs May Receive Bad Data If FIFOs Fill Up	x	.
33	05000301	Memory-To-Memory DMA Source/Destination Descriptors Must Be in Same Memory Space	x	x
34	05000304	SSYNCs After Writes To CAN/DMA MMR Registers Are Not Always Handled Correctly	x	.
35	05000305	SPORT_HYS Bit in PLL_CTL Register Is Not Functional	x	.
36	05000307	SCKELOW Bit Does Not Maintain State Through Hibernate	x	.
37	05000309	Writing UART_THR While UART Clock Is Disabled Sends Erroneous Start Bit	x	.
38	05000310	False Hardware Errors Caused by Fetches at the Boundary of Reserved Memory	x	x
39	05000312	Errors when SSYNC, CSYNC, or Loads to LT, LB and LC Registers Are Interrupted	x	x
40	05000313	PPI Is Level-Sensitive on First Transfer In Single Frame Sync Modes	x	x
41	05000315	Killed System MMR Write Completes Erroneously on Next System MMR Access	x	.

No.	ID	Description	Rev 0.2	Rev 0.3
42	05000316	EMAC RMII Mode: Collisions Occur in Full Duplex Mode	x	.
43	05000321	EMAC RMII Mode: TX Frames in Half Duplex Fail with Status "No Carrier"	x	.
44	05000322	EMAC RMII Mode at 10-Base-T Speed: RX Frames Not Received Properly	x	x
45	05000341	Ethernet MAC MDIO Reads Do Not Meet IEEE Specification	.	x
46	05000350	UART Gets Disabled after UART Boot	x	.
47	05000355	Regulator Programming Blocked when Hibernate Wakeup Source Remains Active	x	x
48	05000357	Serial Port (SPORT) Multichannel Transmit Failure when Channel 0 Is Disabled	x	x
49	05000366	PPI Underflow Error Goes Undetected in ITU-R 656 Mode	x	x
50	05000371	Possible RETS Register Corruption when Subroutine Is under 5 Cycles in Duration	x	x
51	05000402	SSYNC Stalls Processor when Executed from Non-Cacheable Memory	x	.
52	05000403	Level-Sensitive External GPIO Wakeups May Cause Indefinite Stall	x	x
53	05000416	Speculative Fetches Can Cause Undesired External FIFO Operations	x	x
54	05000425	Multichannel SPORT Channel Misalignment Under Specific Configuration	x	x
55	05000426	Speculative Fetches of Indirect-Pointer Instructions Can Cause False Hardware Errors	x	x
56	05000443	IFLUSH Instruction at End of Hardware Loop Causes Infinite Stall	x	x
57	05000461	False Hardware Error when RETI Points to Invalid Memory	x	x
58	05000462	Synchronization Problem at Startup May Cause SPORT Transmit Channels to Misalign	x	x
59	05000473	Interrupted SPORT Receive Data Register Read Results In Underflow when SLEN > 15	x	x
60	05000475	Possible Lockup Condition when Modifying PLL from External Memory	x	x
61	05000477	TESTSET Instruction Cannot Be Interrupted	x	x
62	05000480	Multiple Simultaneous Urgent DMA Requests May Cause DMA System Instability	x	.
63	05000481	Reads of ITEST_COMMAND and ITEST_DATA Registers Cause Cache Corruption	x	x
64	05000489	PLL May Latch Incorrect Values Coming Out of Reset	x	x
65	05000491	Instruction Memory Stalls Can Cause IFLUSH to Fail	x	x
66	05000494	EXCPT Instruction May Be Lost If NMI Happens Simultaneously	x	x
67	05000501	RXS Bit in SPI_STAT May Become Stuck In RX DMA Modes	x	x
68	05000503	SPORT Sign-Extension May Not Work	x	x
69	05000506	Hardware Loop Can Underflow Under Specific Conditions	x	x

Key: x = anomaly exists in revision
. = Not applicable

DETAILED LIST OF SILICON ANOMALIES

The following list details all known silicon anomalies for the ADSP-BF534/BF536/BF537 including a description, workaround, and identification of applicable silicon revisions.

1. 05000074 - Multi-Issue Instruction with dsp32shiftimm in slot1 and P-reg Store in slot2 Not Supported:

DESCRIPTION:
A multi-issue instruction with dsp32shiftimm in slot 1 and a P register store in slot 2 is not supported. It will cause an exception.

The following type of instruction is not supported because the P3 register is being stored in slot 2 with a dsp32shiftimm in slot 1:

```
R0 = R0 << 0x1 || [ P0 ] = P3 || NOP;      // Not Supported - Exception
```

This also applies to rotate instructions:

```
R0 = ROT R0 by 0x1 || [ P0 ] = P3 || NOP;  // Not Supported - Exception
```

Examples of supported instructions:

```
R0 = R0 << 0x1 || [ P0 ] = R1 || NOP;
R0 = R0 << 0x1 || R1 = [ P0 ] || NOP;
R0 = R0 << 0x1 || P3 = [ P0 ] || NOP;
R0 = ROT R0 by R0.L || [ P0 ] = P3 || NOP;
```

WORKAROUND:
In assembly programs, separate the multi-issue instruction into 2 separate instructions. This workaround may be built into the development tool chain and/or into the operating system source code. For tool chains and operating systems supported by Analog Devices (VisualDSP++, VDK, CrossCore Embedded Studio, the GNU Tool Chain, and the Linux kernel), please consult the "Silicon Anomaly Tools Support" help page in the applicable documentation and release notes for details.

For all other tool chains and operating systems, see the appropriate supporting documentation for details.

APPLIES TO REVISION(S):
0.2, 0.3

2. 05000119 - DMA_RUN Bit Is Not Valid after a Peripheral Receive Channel DMA Stops:

DESCRIPTION:
After completion of a Peripheral Receive DMA, the DMAx_IRQ_STATUS:DMA_RUN bit will be in an undefined state.

WORKAROUND:
The DMA interrupt and/or the DMAx_IRQ_STATUS:DMA_DONE bits should be used to determine when the channel has completed running.

APPLIES TO REVISION(S):
0.2, 0.3

3. 05000122 - Rx.H Cannot Be Used to Access 16-bit System MMR Registers:

DESCRIPTION:
When accessing 16-bit system MMR registers, the high half of the data registers may not be used. If a high half register is used, incorrect data will be written to the system MMR register, but no exception will be generated. For example, this access would fail:

```
W[P0] = R5.H;   // P0 points to a 16-bit System MMR
```

WORKAROUND:
Use other forms of 16-bit transfers when accessing 16-bit system MMR registers. For example:

```
W[P0] = R5.L;   // P0 points to a 16-bit System MMR
R4.L = W[P0];
R3 = W[P0](Z);
W[P0] = R3;
```

This workaround may be built into the development tool chain and/or into the operating system source code. For tool chains and operating systems supported by Analog Devices (VisualDSP++, VDK, CrossCore Embedded Studio, the GNU Tool Chain, and the Linux kernel), please consult the "Silicon Anomaly Tools Support" help page in the applicable documentation and release notes for details.

For all other tool chains and operating systems, see the appropriate supporting documentation for details.

APPLIES TO REVISION(S):
0.2, 0.3

4. 05000180 - PPI_DELAY Not Functional in PPI Modes with 0 Frame Syncs:

DESCRIPTION:
In self-triggered, continuous sampling operation of the PPI, the delay count specified in the PPI_DELAY register is ignored. As soon as this mode is enabled, data is transferred.

WORKAROUND:
If a delay is needed, either ignore received data in software or use a mode with at least one frame sync.

APPLIES TO REVISION(S):
0.2, 0.3

5. 05000244 - If I-Cache Is On, CSYNC/SSYNC/IDLE Around Change of Control Causes Failures:

DESCRIPTION:

When instruction cache is enabled, a CSYNC/SSYNC/IDLE around a Change of Control (including asynchronous exceptions/interrupts) can cause unpredictable results.

An example of the most common sequence that can cause this issue consists of a BRCC (NP) followed by CSYNC/SSYNC/IDLE anywhere in the next three instructions:

```
BRCC X [predicted not taken]
NOP
NOP
CSYNC/SSYNC/IDLE  // this instruction is bad in any of the 3 instructions following BRCC X
```

Another sequence that would encounter this problem would be if a BRCC (BP) which points to a CSYNC/SSYNC/IDLE is followed by a stalling instruction that allows the speculatively fetched CSYNC/SYNC/IDLE to "catch up" to the BRCC to within two cycles:

```
BRCC X (BP)
Y: ...
   ...
X: CSYNC/SSYNC/IDLE
```

This sequence is extremely difficult to reproduce with a failure. It requires an exact combination of stalls before the BRCC along with some very specific cache behavior.

WORKAROUND:

Turning the instruction cache off is one way to avoid the anomaly.

Assembly code must avoid the above scenarios. For all cases not related to asynchronous events, NOPs should be inserted to avoid the anomaly condition described:

```
IF CC JUMP ...;                          IF CC JUMP X (BP);
NOP;           // 3 NOP Pads             ...
NOP;                                     ...
NOP;                                     X: NOP; // NOP Pad at Jump Target
CSYNC/SSYNC/IDLE;                        CSYNC/SSYNC/IDLE;
```

For asynchronous interrupt events, the SSYNC/CSYNC/IDLE instruction can be protected by disabling interrupts and padding the SSYNC/CSYNC/IDLE with 2 leading NOPs:

```
CLI R0;
NOP; NOP;             // 2 Padding Instructions
CSYNC/SSYNC/IDLE
STI R0;
```

For exceptions, 3 padding NOPs should be implemented following any access to a cacheable region of memory.

Finally, as the workaround involves Supervisor Mode instructions to disable and enable interrupts, this does not apply to User Mode. In user space, do not use CSYNC or SSYNC instructions.

This workaround may be built into the development tool chain and/or into the operating system source code. For tool chains and operating systems supported by Analog Devices (VisualDSP++, VDK, the GNU Tool Chain, and the Linux kernel), please consult the "Silicon Anomaly Tools Support" help page in the applicable documentation and release notes for details.

For all other tool chains and operating systems, see the appropriate supporting documentation for details.

APPLIES TO REVISION(S):

0.2

6. 05000245 - False Hardware Error from an Access in the Shadow of a Conditional Branch:

DESCRIPTION:

If a load accesses reserved or illegal memory on the opposite control flow of a conditional jump to the taken path, a false hardware error will occur.

The following sequences demonstrate how this can happen:

Sequence #1:
For the "predicted not taken" branch, the pipeline will load the instructions that sequentially follow the branch instruction that was predicted not taken. By the pipeline design, these instructions can be speculatively executed before they are aborted due to the branch misprediction. The anomaly occurs if any of the three instruction slots following the branch contain loads which might cause a hardware error:

```
BRCC X [predicted not taken]
R0 = [P0];     // If any of these three loads accesses non-existent
R1 = [P1];     // memory, such as external SDRAM when the SDRAM
R2 = [P2];     // controller is off, then a hardware error will result.
```

Sequence #2:
For the "predicted taken" branch, the one instruction slot at the destination of the branch cannot contain an access which might cause a hardware error:

```
BRCC X (BP)
Y: ...
   ...
X: R0 = [P0];  // If this instruction accesses non-existent memory,
               // such as external SDRAM when the SDRAM controller
               // is off, then a hardware error will result.
```

WORKAROUND:

If you are programming in assembly, it is necessary to avoid the conditions described above.

This workaround may be built into the development tool chain and/or into the operating system source code. For tool chains and operating systems supported by Analog Devices (VisualDSP++, VDK, CrossCore Embedded Studio, the GNU Tool Chain, and the Linux kernel), please consult the "Silicon Anomaly Tools Support" help page in the applicable documentation and release notes for details.

For all other tool chains and operating systems, see the appropriate supporting documentation for details.

APPLIES TO REVISION(S):

0.2, 0.3

7. 05000250 - Incorrect Bit Shift of Data Word in Multichannel (TDM) Mode in Certain Conditions:

DESCRIPTION:
In multichannel mode, when the period of the frame sync is bigger than the actual data frame width by ONE bit (i.e. there is one "inactive bit"), the FIRST word of the transmitted frame is shifted to the left by one bit and the LSB is the MSB of the second word. All other words are transmitted correctly.

All data is transmitted correctly if the Frame Sync period is equal to the actual frame width or bigger by more than 1 bit.

For example, if there are 8 words of 16-bit data each, that would be 128 bits in the data frame.

If RFSDIV = 127 --> all data words are CORRECT
If RFSDIV = 128 --> first word is INCORRECT
If RFSDIV = 129 --> all data words are CORRECT
If RFSDIV = 130 --> all data words are CORRECT

WORKAROUND:
Set the RFSDIV register value to the number of data bits +/- 1 to avoid the case described above.

APPLIES TO REVISION(S):
0.2

8. 05000252 - EMAC TX DMA Error After an Early Frame Abort:

DESCRIPTION:
A DMA error can occur when the EMAC performs a TX early abort operation when the data and descriptors are in different memory spaces (i.e., data in external memory and descriptors in L1).

A TX Early Abort is rare. It occurs when both:

1) Early in a frame, the EMAC TX FIFO is still doing its initial FIFO fill.
   (i.e., it has not yet reached the 90th byte or the end of frame, whichever is less)
and
2) Any of the following four frame abort cases occurs:
   a) DMA Underrun
   b) Excessive Collisions
   c) Excessive Deferral, only if Deferral Check is enabled (OPMODE:DC = 1)
   d) Late Collision,only if Late Collision Retry is disabled (OPMODE:LCTRE = 0)

In addition to the four causes of early abort, it may be possible to see this anomaly later in the frame (after 90 bytes have been delivered by DMA) in the event of either a:

a) DMA Underrun
b) Late Collision, only if Late Collision Retry is disabled (OPMODE:LCTRE = 0)

WORKAROUND:
The workaround is to configure the memory space like this:

	Descriptor	Data Buffer
Data 1	in A	in B
Data 2	in C	in D
Status	in D	in E

Each letter (A-E) represents a memory space. Different letters may represent the same or different memory space, but each group of items where the same letter appears must be in the same memory space.

APPLIES TO REVISION(S):
0.2

9. 05000253 - Maximum External Clock Speed for Timers:

DESCRIPTION:
The General-Purpose Timers can generate PWM output waveforms on the TMRx pin whose timing is quantified in either system clock (SCLK) periods or in periods of an externally supplied clock (TMRCLK or TACLK). For proper operation, SCLK must be faster than the source that is utilized, TMRCLK or TACLK.

The specification in the data sheet and hardware reference manual allows for a SCLK::TMRCLK and SCLK::TACLK ratio of up to 2::1. However, the maximum rate is less than this limit.

WORKAROUND:
A minimum SCLK::TMRCLK or SCLK::TACLK ratio of 3::1 is safe to use.

APPLIES TO REVISION(S):
0.2

10. 05000254 - Incorrect Timer Pulse Width in Single-Shot PWM_OUT Mode with External Clock:

DESCRIPTION:
If a Timer is in PWM_OUT mode AND is clocked by an external clock as opposed to the system clock (i.e., clocked by a signal applied to either PPI_CLK or a flag pin) AND is in single-pulse mode (PERIOD_CNT = 0), then the generated pulse width may be off by +1 or -1 count. All other modes are not affected by this anomaly.

WORKAROUND:
The suggested workaround is to use continuous mode instead of the single-pulse mode. You may enable the timer and immediately disable it again. The timer will generate a single pulse and count to the end of the period before effectively disabling itself. The generated waveform will be of the desired length.

If PULSEWIDTH is the desired width, the following sequence will produce a single pulse:

```
TIMERx_CONFIG = PWM_OUT|CLK_SEL|PERIOD_CNT|IRQ_ENA;  // Optional: PULSE_HI|TIN_SEL|EMU_RUN
TIMERx_PERIOD = PULSEWIDTH + 2;                      // Slightly bigger than the width
TIMERx_WIDTH  = PULSEWIDTH;
TIMER_ENABLE  = TIMENx;
TIMER_DISABLE = TIMDISx;
<wait for interrupt (at end of period)>
```

APPLIES TO REVISION(S):
0.3

11. 05000255 - Entering Hibernate State with RTC Seconds Interrupt Not Functional:

DESCRIPTION:
Entering the low-power Hibernate state is achieved by disabling the internal Voltage regulator (Vddint = 0 Volts). The Real-Time Clock (RTC) is programmed to wake up the voltage regulator at a specific event.

If the RTC wake-up event is the Seconds event (RTC_ICTL = 0x0004), the wake-up signal is erroneously active for the entire second. In this case, the Voltage Regulator cannot be disabled because it will always be woken up immediately.

This applies only to Hibernate state. Deep-Sleep and other low power modes work correctly with the Seconds event.

Note that, for RTC events of greater period, the minimum time before the processor can re-enter Hibernate state after a wake-up event from the RTC is one second. For instance, in the case of the Minute event, the processor cannot re-enter Hibernate mode during the first second.

WORKAROUND:
A possible workaround is to do the following steps:

1) Disable the prescaler (RTC_PREN = 0); thus, the RTC will generate 32768 ticks every second.
2) Use the Stopwatch event instead of the Seconds event. The stopwatch register must be set to 32768 (or, more generally, to a value corresponding to the desired frequency) at every wake-up event.
3) In this case, the wake-up signal will be only approximately 15usec long. This is the minimum time the application has to wait (or do useful things) before re-entering Hibernate mode after a wake-up.

Note that this workaround implies that the RTC is not used for keeping track of the actual time, since the counters are incremented at 32768Hz instead of 1Hz.

APPLIES TO REVISION(S):
0.2

12. 05000256 - EMAC MDIO Input Latched on Wrong MDC Edge:

DESCRIPTION:
The MDIO input is latched on the falling edge of MDC plus one SCLK cycle. This may result in communication errors with certain PHY devices when the MDC clock is programmed for very short cycle times. MDC/MDIO are used to communicate with the internal control registers of a PHY device.

Note: the MDC clock is driven by the processor and is derived from SCLK by a programmable clock divider in the EMAC peripheral. If the divisor register MDCDIV is set to N, the period of the MDC will be 2(N+1) times SCLK.

PHYs normally drive MDIO on the rising edge of MDC, while the processor samples MDIO on MDC falling plus one SCLK, as stated above. This means that the total delay of the PHY's clock-to-output delay plus propagation delay (call it tdMDIO) must not exceed one-half of the MDC period plus one SCLK cycle. If the processor's MDC divisor register is set to N, then the propagation delay must not exceed [2(N+1)]/2 + 1 = N+2 SCLK cycles.

WORKAROUND:
If a PHY encounters this issue, then determine tdMDIO for the PHY and set MDCDIV > (tdMDIO/tSCLK) + 2.

APPLIES TO REVISION(S):
0.2

13. 05000257 - Interrupt/Exception During Short Hardware Loop May Cause Bad Instruction Fetches:

DESCRIPTION:
Unpredictable behavior can result when hardware loops shorter than 4 instructions in length are interrupted at the end of the loop due to an interrupt or exception. In this situation, the processor's loop buffers, which are used to reduce the instruction fetch latency, operate incorrectly, resulting in the wrong instructions being fetched as the loop exits.

WORKAROUND:
There are a few possible workarounds for this anomaly. The first is to clear the loop buffers by writing to the Loop Counter registers (LC0 and LC1) inside all interrupt/exception handlers:

```
R0 = LC0;
LC0 = R0;
R0 = LC1;
LC1 = R0;
```

A second idea would be to include the loop counters in the context switch code:

```
[--SP] = LC0;
[--SP] = LC1;

<interrupt code>

LC1 = [SP++];
LC0 = [SP++];
```

Finally, another workaround would be to pad the loop with NOPs to increase the loop length to greater than or equal to 4 instructions.

Alternatively, if the event handlers use hardware loops, the above steps are not required, since every time an LCx register is written to, its corresponding loop buffer is cleared.

This workaround may be built into the development tool chain and/or into the operating system source code. For tool chains and operating systems supported by Analog Devices (VisualDSP++, VDK, the GNU Tool Chain, and the Linux kernel), please consult the "Silicon Anomaly Tools Support" help page in the applicable documentation and release notes for details.

For all other tool chains and operating systems, see the appropriate supporting documentation for details.

APPLIES TO REVISION(S):
0.2

14. 05000258 - Instruction Cache Is Corrupted When Bits 9 and 12 of the ICPLB Data Registers Differ:

DESCRIPTION:
When bit 9 and bit 12 of the ICPLB Data MMR differ, the cache may not update properly. For example, for a particular cache line, the cache tag may be valid while the contents of that cache line are not present in the cache.

WORKAROUND:
Set bit 9 to the state of bit 12 in each ICPLB entry.

This workaround may be built into the development tool chain and/or into the operating system source code. For tool chains and operating systems supported by Analog Devices (VisualDSP++, VDK, the GNU Tool Chain, and the Linux kernel), please consult the "Silicon Anomaly Tools Support" help page in the applicable documentation and release notes for details.

For all other tool chains and operating systems, see the appropriate supporting documentation for details.

APPLIES TO REVISION(S):
0.2

15. 05000260 - ICPLB_STATUS MMR Register May Be Corrupted:

DESCRIPTION:
The ICPLB Status register cannot be relied upon to determine which CPLB caused an exception. This register is corrupted if:

1) There is a jump to anywhere within the last 64 bits of a page (as defined by an ICPLB)
and
2) An instruction located within these last 64 bits generates an instruction exception,
and
3) Speculative instruction fetches increment into the next page and encounter another instruction exception cause.

When all of these criteria are met, ICPLB_STATUS will reflect the speculative instruction fetch rather than the initial exception cause.

WORKAROUND:
Handle instruction protection violations and ICPLB multiple hits without using this register.

Use the ICPLB_FAULT_ADDR register to see the address that caused the exception:
1) For CPLB misses, exceptions simply swap in a CPLB entry that covers the address in question.
2) For the case of multiple CPLB hits, use the ICPLB_FAULT_ADDR register to find out which address caused the exception and then iterate through all the CPLB entries to see which of the CPLBs cover the fault address.
3) For a protection violation exception, the handling is user-specific.

APPLIES TO REVISION(S):
0.2

16. 05000261 - DCPLB_FAULT_ADDR MMR Register May Be Corrupted:

DESCRIPTION:
The DCPLB_FAULT_ADDR MMR register may be corrupted. For this to happen, an aborted data memory access must generate BOTH a protection exception and a stall (due to either a dual-DAG collision, addressing of cacheable memory and missing, or simply fetching from L2).

WORKAROUND:
1) Immediately return from the data exception handler upon an initial entry into the handler (without any servicing yet), and then trust the DCPLB_FAULT_ADDR upon a second pass through the same data CPLB exception handler. Unless the cause is an exception that is serviced, the exception will be regenerated and cause a second pass. In the second pass, however, the DCPLB_FAULT_ADDR register will be correct because it is never generated incorrectly immediately after returning from an exception handler. To ensure that the same exception is being responded to in the second pass (rather than a higher priority exception), a copy of the RETX register should be acquired in the first pass and compared against in the second pass.

or

2) Be tolerant of the artifacts generated by misprocessing the exception. For the three types of data memory exceptions - protection violation, CPLB miss, and CPLB multiple hit - the recommended software workaround is as follows:

   a) For data protection exceptions, use the DCPLB_STATUS register rather than the DCPLB_FAULT_ADDR register in the handler. This will provide the page of the protection violation rather than the full address of the exception. Although not ideal, this likely provides sufficient information.

   b) For data CPLB miss exceptions, use the DCPLB_FAULT_ADDR register, but be warned that the address reported in this register might be that of a previously canceled speculative exception rather than the true current exception. It might therefore:

      i) point to a page which already has a loaded descriptor.
      or
      ii) point to an address which will never actually be fetched.

      For case i), although a CPLB miss handler might create a redundant CPLB entry (unless further page checking is done), this may be tolerated if a multiple CPLB hit handler exists to remove this rarely generated redundant page descriptor.

      For case ii), since the DCPLB_FAULT_ADDR register will never be incorrect immediately after returning from the exception handler, nonsensical addressees can be ignored by the CPLB miss handler without generating an infinite exception handler loop due to repetitively faulty DCPLB_FAULT_ADDR register contents.

   c) For data CPLB multiple hit exceptions, have such a handler thanks to the issue described above. Don't count on multiple CPLB exceptions never occurring.

This workaround may be built into the development tool chain and/or into the operating system source code. For tool chains and operating systems supported by Analog Devices (VisualDSP++, VDK, the GNU Tool Chain, and the Linux kernel), please consult the "Silicon Anomaly Tools Support" help page in the applicable documentation and release notes for details.

For all other tool chains and operating systems, see the appropriate supporting documentation for details.

APPLIES TO REVISION(S):
0.2

17. 05000262 - Stores To Data Cache May Be Lost:

DESCRIPTION:
A committed pending write into the sub-bank targeted by the first of two consecutive dual-DAG operations will be lost when:

1) Data cache is enabled,
and
2) For the first dual-DAG access, DAG0 is a cache miss, DAG1 is a read, and both accesses alias to the same non-L1 sub-bank,
and
3) The second dual-DAG is the next instruction, and DAG1 is an access (read or write) of L1 SRAM,
and
4) There's an unpredicted change of flow within three clock cycles after the first dual-DAG access. The user has no control over the change of flow.

WORKAROUND:
1) Don't use data cache,
or
2) Avoid consecutive dual-DAG memory accesses where the first dual-DAG access:
   a) has both DAGs targeting L2,
   and
   b) has both DAGs aliasing to the same sub-bank,
   and
   c) includes a read by DAG1, which is then immediately followed by the second dual-DAG access where DAG1 is an L1 access.

This workaround may be built into the development tool chain and/or into the operating system source code. For tool chains and operating systems supported by Analog Devices (VisualDSP++, VDK, the GNU Tool Chain, and the Linux kernel), please consult the "Silicon Anomaly Tools Support" help page in the applicable documentation and release notes for details.

For all other tool chains and operating systems, see the appropriate supporting documentation for details.

APPLIES TO REVISION(S):
0.2

18. 05000263 - Hardware Loop Corrupted When Taking an ICPLB Exception:

DESCRIPTION:
There is an error in the hardware loop logic which can cause incorrect instructions to get executed when the processor is running loops and instruction ICPLB exceptions occur.

WORKAROUND:
Either:
1) Avoid using hardware loops,
or
2) Make sure hardware loops are located only in L1 memory,
or
3) Make sure ICPLB exceptions do not occur while executing a hardware loop located outside L1 memory.

If a hardware loop is contained within L1 memory, the loop must not generate an ICPLB exception, for example, by crossing a CPLB page boundary into a page with no valid CPLB definitions. In addition, do not allow branching out to non-L1 memory from within the loop when an ICPLB exception might be generated at the target address. Also, if the loop might be interrupted and the interrupt service routines (ISR) reside in non-L1 memory, the ISRs should not generate ICPLB exceptions.

APPLIES TO REVISION(S):
0.2

19. 05000264 - CSYNC/SSYNC/IDLE Causes Infinite Stall in Penultimate Instruction in Hardware Loop:

DESCRIPTION:
If a SSYNC, CSYNC, or IDLE is placed in the second to last instruction of a hardware loop, there is a possibility that the processor will enter an infinite stall when trying to execute the sync.

WORKAROUND:
Do not put a SSYNC, CSYNC, or IDLE instruction in the second to last instruction of a hardware loop.

Because an interrupt or an exception will bring the processor out of the stall, this problem may not be obvious if you're running DMA or interrupts.

This workaround may be built into the development tool chain and/or into the operating system source code. For tool chains and operating systems supported by Analog Devices (VisualDSP++, VDK, the GNU Tool Chain, and the Linux kernel), please consult the "Silicon Anomaly Tools Support" help page in the applicable documentation and release notes for details.

For all other tool chains and operating systems, see the appropriate supporting documentation for details.

APPLIES TO REVISION(S):
0.2

20. 05000265 - Sensitivity To Noise with Slow Input Edge Rates on External SPORT TX and RX Clocks:

DESCRIPTION:

A noisy board environment combined with slow input edge rates on external SPORT receive (RSCLK) and transmit clocks (TSCLK) may cause a variety of observable problems. Unexpected high frequency transitions on the RSCLK/TSCLK can cause the SPORT to recognize an extra noise-induced glitch clock pulse.

The high frequency transitions on the RSCLK/TSCLK are most likely to be caused by noise on the rising or falling edge of external serial clocks. This noise, coupled with a slowly transitioning serial clock signal, can cause an additional bit-clock with a short period due to high sensitivity of the clock input. A slow slew rate input allows any noise on the clock input around the switching point to cause the clock input to cross and re-cross the switching point. This oscillation can cause a glitch clock pulse in the internal logic of the serial port.

Problems which may be observed due to this glitch clock pulse are:

- In stereo serial modes, this will show up as missed frame syncs, causing left/right data swaps.
- In multichannel mode, this will show up as MFD counts appearing inaccurate or skipped frames.
- In Normal (Early) Frame sync mode, data words received will be shifted right one bit. The MSB may be incorrectly captured in sign extension mode.
- In any mode, received or transmitted data words may appear to be partially right shifted if noise occurs on any input clocks between the start of frame sync and the last bit to be received or transmitted.

In Stereo Serial mode (bit 9 set in SPORTx_RCR2), unexpected high frequency transitions on RSCLK/TSCLK can cause the SPORT to miss rising or falling edges of the word clock. This causes left or right words of Stereo Serial data to be lost. This may be observed as a Left/Right channel swap when listening to stereo audio signals. The additional noise-induced bit-clock pulse on the SPORT's internal logic results in the FS edge-detection logic generating a pulse with a smaller width and, at the same time, prevents the SPORT from detecting the external FS signal during the next 'normal' bit-clock period. The FS pulse with smaller width, which is the output of the edge-detection logic, is ignored by the SPORT's sequential logic. Due to the fact that the edge detection part of the FS-logic was already 'triggered', the next 'normal' RSCLK will not detect the change in RFS anymore. In I^2S/EIAJ mode, this results in one stereo sample being detected/transferred as two left/right channels, and all subsequent channels will be word-swapped in memory.

In multichannel mode, the mutlichannel frame delay (MFD) logic receives the extra sync pulse and begins counting early or double counting (if the count has already begun). A MFD of zero can roll over to 15, as the count begins one cycle early.

In early frame sync mode, if the noise occurs on the driving edge of the clock the same cycle that FS becomes active, the FS logic receives the extra runt pulse and begins counting the word length one cycle early. The first bit will be sampled twice and the last bit will be skipped.

In all modes, if the noise occurs in any cycle after the FS becomes active, the bit counting logic receives the extra runt pulse and advances too rapidly. If this occurs once during a work unit, it will finish counting the word length one cycle early. The bit where the noise occurs will be sampled twice, and the last bit will be skipped.

WORKAROUND:

1) Decrease the sensitivity to noise by increasing the slew rate of the bit clock or make the rise and fall times of serial bit clocks short, such that any noise around the transition produces a short duration noise-induced bit-clock pulse. This small high-frequency pulse will not have any impact on the SPORT or on the detection of the frame-sync. Sharpen edges as much as possible, if this is suitable and within EMI requirements.
2) If possible, use internally generated bit-clocks and frame-syncs.
3) Follow good PCB design practices. Shield RSCLK with respect to TSCLK lines to minimize coupling between the serial clocks.
4) Separate RSCLK, TSCLK, and Frame Sync traces on the board to minimize coupling which occurs at the driving edge when FS switches.

A specific workaround for problems observed in Stereo Serial mode is to delay the frame-sync signal such that noise-induced bit-clock pulses do not start processing the frame-sync. This can be achieved if there is a larger serial resistor in the frame-sync trace than the one in the bit-clock trace. Frame-sync transitions should not cross the 50% point until the bit-clock crosses the 10% of VDD threshold (for a falling edge bit-clock) or the 90% threshold (for a rising edge bit-clock).

This workaround only applies to rev 0.3 and later silicon: To improve immunity to noise, optional hysteresis can be enabled for input pins by setting bit 15 of the PLL_CTL register.

APPLIES TO REVISION(S):

0.2, 0.3

21. 05000268 - Memory DMA Error when Peripheral DMA Is Running with Non-Zero DEB_TRAFFIC_PERIOD:

DESCRIPTION:
When a Memory DMA channel is active at the same time as any peripheral DMA channel and the DEB_TRAFFIC_PERIOD bits in the DMA_TC_PER register are non-zero, data in the Memory DMA transfer can be corrupted.

WORKAROUND:
1) Do not use a Memory DMA channel at the same time as a Peripheral DMA channel in your application
or
2) If you do use Memory DMA at the same time as peripheral DMA, set the DEB_TRAFFIC_PERIOD bits in the DMA_TC_PER register to b#0000.

APPLIES TO REVISION(S):
0.2

22. 05000270 - High I/O Activity Causes Output Voltage of Internal Voltage Regulator (Vddint) to Decrease:

DESCRIPTION:
Heavy I/O activity can cause VDDint to decrease. The reference voltage, which is used to create the set point for the loop, is decreased by the supply noise. The voltage may drop to a level that is lower than the minimum required to meet your application's frequency of operations. The VDDint value returns to the programmed value once high I/O activity is halted.

WORKAROUND:
This issue does not occur when an external regulator is used. To determine if the problem exists in your application, you should monitor the VDDint waveform under the following conditions/setup:

- Apply the maximum VDDext based on the tolerance of VDDext supply.
- Run the application in a steady state (non-startup) condition.
- Connect an oscilloscope with minimum ground and signal loops to VDDint.
- Set the oscilloscope to trigger on a VDDint value that is 5% lower than the programmed value.

The following items can mitigate this issue:

- Lower the I/O activity by reducing SCLK frequency, if possible.
- Increase the programmed value of the voltage regulator by an amount (in multiples of 50mV) closest to the observed decrease.
- Ensure adequate bypassing on VDDext.

APPLIES TO REVISION(S):
0.2

23. 05000272 - Certain Data Cache Writethrough Modes Fail for Vddint <= 0.9V:

DESCRIPTION:
Data can become corrupted if data cache is enabled in write through mode and the AOW bit of the DCPLB is not set and Vddint is 0.9V or less.

WORKAROUND:
When Vddint <= 0.9V, either operate data cache in write back mode or set the AOW bit of the DCPLB when operating in write through mode. When Vddint is greater than 0.9V, the anomaly does not exist.

APPLIES TO REVISION(S):
0.2, 0.3

24. 05000273 - Writes to Synchronous SDRAM Memory May Be Lost:

DESCRIPTION:
When the Core Clock is not at least twice as fast as the the System Clock, 32-bit or wider writes to SDRAM memory may be lost. Note that since cache victims are effectively 256 bit wide writes, cache victimization will also trigger this anomaly.

WORKAROUND:
Either:
1) Make sure that the Core Clock (CCLK) is at least twice as fast as the System Clock (SCLK)
or
2) Make sure all external memory writes are 16 bits wide or less:

```
W[P2] = R0;   // 16-bit write
B[P2] = R0;   //  8-bit write
```

If using data cache, the Write Through policy should be used since there is no cache victimization in this mode.

APPLIES TO REVISION(S):
0.2

25. 05000277 - Writes to an I/O Data Register One SCLK Cycle after an Edge Is Detected May Clear Interrupt:

DESCRIPTION:
If a write to any I/O data register (data, clear, set and toggle registers) occurs one system clock cycle after an edge is detected on an edge-triggered interrupt, then the bit may be cleared one system clock cycle after it has been set.

If the bit has been programmed to generate an interrupt, then the interrupt will occur, but there will be no indication of which bit signalled the interrupt. The interrupt will be lost if the core clock is not running or if the SIC_IMASK bit is not set to enable the interrupt.

WORKAROUND:
If only one edge-sensitive source is assigned to one interrupt, it can be assumed to be the source of the interrupt and a read instruction of SIC_ISR and the I/O registers is not required. Note that all interrupts are properly executed, when enabled.

Use level-sensitive interrupts instead of edge-sensitive interrupts. Toggle the polarity between received edges to prevent re-entry of the interrupt service routine and to sensitize for the next edge. This is applicable when the latency between two edges is sufficient to serve the interrupt service routine or can be used for request lines. Toggling polarity can be used when looking for both edges. For only one edge, however, the other interrupt must be ignored.

APPLIES TO REVISION(S):
0.2

26. 05000278 - Disabling Peripherals with DMA Running May Cause DMA System Instability:

DESCRIPTION:
If a peripheral (PPI, SPORT, SPI, etc.) is disabled while DMA is running and before the associated DMA channel is disabled, the DMA system may be corrupted. In applications with multiple DMA channels running concurrently, this anomaly manifests itself with missing data or shuffled data being transferred. Although the anomaly also affects applications with a single DMA channel, its effects may not be visible if the peripheral is being shut down by the user code.

WORKAROUND:
If the DMA channel is running, disable the peripheral's associated DMA channel before disabling the peripheral itself.

If the DMA channel is stopped, the peripheral must be disabled before the associated DMA channel is disabled. When a channel is disabled, the DMA unit ignores the peripheral interrupt and passes it directly to the interrupt controller, thus generating unwanted interrupts.

APPLIES TO REVISION(S):
0.2

27. 05000280 - SPI Master Boot Mode Does Not Work Well with Atmel Data Flash Devices:

DESCRIPTION:
The boot ROM code causes booting from Atmel serial dataflash devices to fail if the boot stream is larger than the SPI device's page size. This only happens if the dataflashes are operating in Standard Addressing Mode.

WORKAROUND:
Use the series D devices (i.e., AT45DB642D) in the Power-Of-2 Addressing Mode.

APPLIES TO REVISION(S):
0.2, 0.3

28. 05000281 - False Hardware Error when ISR Context Is Not Restored:

DESCRIPTION:
In some instances, exiting an interrupt service routine (ISR) without restoring context may be desired. Consider the following sequence:

```
ISR_Exit:
   RAISE 14;   // instruction A
   RTI;        // instruction B
```

This sequence will return from the current interrupt level and then immediately execute the level 14 interrupt service routine. Ideally, the latter would then restore the context before returning to user level, thus saving time in the first ISR.

In order to describe the problem, assume that the first interrupt occurs at an instruction like:

```
Rx = [Py];   // instruction C
```

or any similar instruction.

The processor will jump to the ISR (RETI will contain the address of instruction C). If the ISR changes Py, when the processor reaches instruction B above, it will speculatively fetch instruction C, which could now point to an invalid address. Because of instruction A, instruction B will not be executed, however, the hardware error condition will be latched. The hardware exception will then be triggered at the next system MMR read.

WORKAROUND:
Load the RETI register (before the above "`raise; rti;`" sequence) with a location where speculative fetches will not cause hardware errors.

APPLIES TO REVISION(S):
0.2

29. 05000282 - Memory DMA Corruption with 32-Bit Data and Traffic Control:

DESCRIPTION:
This anomaly applies to cases where:
1) Memory DMA (MDMA) channels are used in 32-bit mode (WDSIZE in MDMA_yy_CONFIG = 0b10).
and
2) Traffic Control is enabled to group accesses of the same direction together (DMA_TC_PER register contains non-zero fields).

In this particular case, high and low words may be inverted and/or interrupts may be lost.

WORKAROUND:
This anomaly is avoided if MDMA channels are used in 16-bit mode or if traffic control is disabled (DMA_TC_PER = 0x0000).

Note: on this device, the 16-bit MDMA is more efficient than the 32-bit mode for transfers from L1 to external memory and vice versa.

APPLIES TO REVISION(S):
0.2

30. 05000283 - System MMR Write Is Stalled Indefinitely when Killed in a Particular Stage:

DESCRIPTION:
Consider the following sequence:
1) System MMR write is stalled.
2) Interrupt/Exception occurs while the System MMR write is stalled (thus killing the write).
3) Interrupt/Exception Service Routine performs an SSYNC instruction.

In order for this anomaly to happen, the change in program flow must kill the write in one particular stage of the execution pipeline. In this case, the anomaly will cause the MMR logic to think that the killed System MMR access is still valid. The SSYNC will therefore stall the processor indefinitely or until it is interrupted itself by a higher priority interrupt or event.

Similarly, if the System MMR write is killed by an instruction itself, such as a conditional branch, the infinite stall can happen if the store buffer is full and emptying out to slow external memory.

```
cc = r0 == r0;   // always true
if cc jump skip;
W[p0] = r1.l;    // System MMR access is fetched and killed
skip: ...
```

NOTE: if a user tries to halt the processor in the handler via the debugging tools, the infinite stall will also lock out the Emulation event.

WORKAROUND:
The workaround is to reset the MMR logic with another killed System MMR access that has no other side-effects on the application. For instance, read from the CHIPID register. The following code snippet, executed at the beginning of each interrupt/exception handler, will work around this anomaly:

```
cc = r0 == r0;   // always true
p0.h = 0xffc0;   // System MMR space CHIPID
p0.l = 0x0014;
if cc jump skip; // always skip MMR access, but MMR access is fetched and killed
r0 = [p0];       // bogus System MMR read to work around the anomaly
skip: ...        // continue with handler code
```

In the case of MMR writes being killed by the conditional branches, it is sufficient to insert 2 NOPs or any other non-MMR instructions in the location immediately after the conditional branch.

NOTE: in order to prevent lock-ups during debugging sessions, always set a breakpoint after the above code snippet if you need to halt the processor in the handler code.

This workaround may be built into the development tool chain and/or into the operating system source code. For tool chains and operating systems supported by Analog Devices (VisualDSP++, VDK, CrossCore Embedded Studio, the GNU Tool Chain, and the Linux kernel), please consult the "Silicon Anomaly Tools Support" help page in the applicable documentation and release notes for details.

For all other tool chains and operating systems, see the appropriate supporting documentation for details.

APPLIES TO REVISION(S):
0.2

31. 05000285 - TXDWA Bit in EMAC_SYSCTL Register Is Not Functional:

DESCRIPTION:
The EMAC TX DMA Word Alignment bit (TXDWA, bit 4) of the EMAC_SYSCTL register, which allows software to program whether TX frame data will begin on an odd- or even-aligned word address, is not functional. Reads of this bit will always return 0, and writes to this bit are ignored.

WORKAROUND:
None.

APPLIES TO REVISION(S):
0.2

32. 05000288 - SPORTs May Receive Bad Data If FIFOs Fill Up:

DESCRIPTION:
The SPORT receives incorrect data if it is configured as follows:

1) The secondary receive data is enabled (RXSE=1) or the word length > 16 bits.
and
2) The RX FIFO is filled with 8 words of data.
and
3) An additional word is clocked into the SPORT.

In this case, the overflow does not assert because there is room to hold the data. The overflow will assert if the next piece of data is received without removing data from the FIFO.

This anomaly will cause one piece of primary data to be received in place of secondary data (RxSEC=1) or word swap (SLEN>0xF). Subsequent words will be received correctly.

WORKAROUND:
Avoid the conditions described in the problem description.

Operating so closely to a FIFO overflow should be avoided.

APPLIES TO REVISION(S):
0.2

33. 05000301 - Memory-To-Memory DMA Source/Destination Descriptors Must Be in Same Memory Space:

DESCRIPTION:
When MemDMA source and destination descriptors are in different memory spaces (one in internal memory and one in external memory), and if the traffic control is turned on, then the source descriptor count of descriptor words currently fetched can get corrupted by the value in the current destination descriptor count (which can be greater or less than the original source descriptor count). This will make the source fetch more/less descriptor elements than intended.

One possible result is that some elements of the descriptor may not be loaded. Another possible result is that extra descriptor element fetches may be performed. The descriptor element pointer may also overflow and wrap back to the start of the register set if too many extra fetches occur, thus overwriting good data with bad data in the first few registers (e.g., Next Descriptor Pointer). In this last case, the DMA may not appear to fail until the next descriptor fetch, when it fetches an invalid pointer.

WORKAROUND:
Place source and destination descriptors in the same memory space. Both should be located either in external or internal memory.

APPLIES TO REVISION(S):
0.2, 0.3

34. 05000304 - SSYNCs After Writes To CAN/DMA MMR Registers Are Not Always Handled Correctly:

DESCRIPTION:
The DMA Controller and the CAN peripheral can each hold off Peripheral Access Bus accesses to its MMR space when it is currently accessing the same space itself. This delay may exceed the duration of a subsequent SSYNC instruction in the application code following the write, which could lead to undesired results.

For DMA controllers, when a DMA channel has been granted permission to fetch descriptors from memory, other accesses to System MMRs associated with the same DMA controller will be held off until the descriptor fetch completes, which could take several SCLKs depending on the size of the descriptor being fetched. A side-effect from this behavior would be in the case of DMA interrupts, where the ISR code performs the correct sequence to clear the interrupt request:

```
p0.h = hi(DMA3_IRQ_STATUS);
p0.l = lo(DMA3_IRQ_STATUS);
r0.l = 0x0001;
w[p0] = r0.l;               // Write-1-to-Clear Interrupt Request
ssync;                      // Allow write to complete
rti;
```

If another DMA channel from the same DMA controller is currently fetching descriptors at the time of the write, this write will be delayed and, if the delay exceeds the duration of the subsequent SSYNC instruction, the ISR code will execute the RTI instruction and another vector to the ISR will be executed because the DMAx_IRQ_STATUS bit hasn't yet been cleared. This behavior is true for all System MMRs associated with the DMA Controller busy doing the descriptor fetch.

For the CAN controller, accesses to the RAM area could have similar results when the CAN controller is preparing to transmit or is receiving a message. The CAN registers that comprise the RAM area are the Acceptance Mask registers (CAN_AMxxL and CAN_AMxxH) and the Mailbox RAM registers (CAN_MBxx_DATA0, CAN_MBxx_DATA1, CAN_MBxx_DATA2, CAN_MBxx_DATA3, CAN_MBxx_LENGTH, CAN_MBxx_TIMESTAMP, CAN_MBxx_ID0, and CAN_MBxx_ID1).

WORKAROUND:
If a dummy read from the MMR register is inserted before the SSYNC, this will guarantee that the previous write completes before the read is able to execute. For example, using the above DMA Controller example, read back the IRQ Status register after it is written:

```
p0.h = hi(DMA3_IRQ_STATUS);
p0.l = lo(DMA3_IRQ_STATUS);
r0.l = 0x0001;
w[p0] = r0.l;               // Write-1-to-Clear Interrupt Request
r0.l = w[p0];               // Insert dummy read before ssync
ssync;                      // Allow write to complete
rti;
```

APPLIES TO REVISION(S):
0.2

35. 05000305 - SPORT_HYS Bit in PLL_CTL Register Is Not Functional:

DESCRIPTION:
The SPORT Hysteresis bit (SPORT_HYS, bit 15) in the PLL_CTL register is not functional. This bit always reads as 0, and writing a 1 has no effect.

WORKAROUND:
None.

APPLIES TO REVISION(S):
0.2

36. 05000307 - SCKELOW Bit Does Not Maintain State Through Hibernate:

DESCRIPTION:
The SCKELOW bit (bit 15) of VR_CTL does not maintain status through the hibernate reset sequence, therefore it cannot be read during the boot sequence to determine whether the processor is cold-starting or coming from the hibernate state.

WORKAROUND:
If the Hibernate feature is being used, application code can copy VR_CTL to a specific location in external memory before going to Hibernate, which can then be interrogated in an initialization block to determine whether a context save was performed previously or not. For example, create a 32-bit data element in your source code and use the LDF to resolve it to a specific location. In the LDF:

```
RESOLVE(32BitDataLabel, 32BIT_ADDR_IN_EXTERNAL_SDRAM_DATA_SEGMENT);
```

Then, when you prepare to enter Hibernate in your source code, this pseudo-code can be used:

```
#define VR_CTL_HIBERNATE_VALUE   0x000080DC      // Your value for VR_CTL goes here

PTR_TO_32BitDataLabel = VR_CTL_HIBERNATE_VALUE;  // 32-Bit Access
VR_CTL = VR_CTL_HIBERNATE_VALUE;                 // 16-Bit Access

CLI/IDLE PLL Programming Sequence;               // Latch write to VR_CTL
```

Then, in an initialization block, this pseudo-code can be used to check for this exact 32-bit value to determine whether that location was previously written or not:

```
Read PTR_TO_32BitDataLabel;
Compare to VR_CTL_HIBERNATE_VALUE;

if TRUE
   Execute post-hibernate boot sequence;
else
   Perform full boot;
```

APPLIES TO REVISION(S):
0.2

37. 05000309 - Writing UART_THR While UART Clock Is Disabled Sends Erroneous Start Bit:

DESCRIPTION:
If the UART Transmit Hold Register (UART_THR) is written while the internal UART clocks are off (UCEN=0 in UART_GCTL), the UART TX pin is driven low until the UART clocks are subsequently enabled (UCEN=1 in UART_GCTL). At that point, the UART logic will then drive the actual UART packet to the TX pin, including start bit, data, parity, and stop bits.

The net effect is that the original low-going edge on the TX pin (at the time the write to UART_THR takes place) will be interpreted as a start bit by a connected receiver. Since the UART logic will properly send the message when the UART clocks do get enabled, this is likely to cause frame errors because the UART will be driving data during the time that the receiver is expecting parity/stop information.

WORKAROUND:
Do not write to the UART_THR register when UCEN=0.

APPLIES TO REVISION(S):
0.2

38. 05000310 - False Hardware Errors Caused by Fetches at the Boundary of Reserved Memory:

DESCRIPTION:
Due to fetches near boundaries of reserved memory, a false Hardware Error (External Memory Addressing Error) is generated under the following conditions:

1) A single valid CPLB spans the boundary of the reserved space. For example, a CPLB with a start address at the beginning of L1 instruction memory and a size of 4MB will include the boundary to reserved memory.

2) Two separate valid CPLBs are defined, one that covers up to the byte before the boundary and a second that starts at the boundary itself. For example, one CPLB is defined to cover the upper 1kB of L1 instruction memory before the boundary to reserved memory, and a second CPLB is defined to cover the reserved space itself.

As long as both sides of the boundary to reserved memory are covered by valid CPLBs, the false error is generated. Note that this anomaly also affects the boundary of the L1_code_cache region if instruction cache is enabled. In other words, the boundary to reserved memory, as described above, moves to the start of the cacheable region when instruction cache is turned on.

WORKAROUND:
Leave at least 76 bytes free before any boundary with a reserved memory space. This will prevent false hardware errors from occurring.

This workaround may be built into the development tool chain and/or into the operating system source code. For tool chains and operating systems supported by Analog Devices (VisualDSP++, VDK, CrossCore Embedded Studio, the GNU Tool Chain, and the Linux kernel), please consult the "Silicon Anomaly Tools Support" help page in the applicable documentation and release notes for details.

For all other tool chains and operating systems, see the appropriate supporting documentation for details.

APPLIES TO REVISION(S):
0.2, 0.3

39. 05000312 - Errors when SSYNC, CSYNC, or Loads to LT, LB and LC Registers Are Interrupted:

DESCRIPTION:
When instruction cache is enabled, invalid code may be executed when any of the following instructions are interrupted:

- **CSYNC**
- **SSYNC**
- **LCx =**
- **LTx =** (only when LCx is non-zero)
- **LBx =** (only when LCx is non-zero)

When this problem occurs, a variety of incorrect things could happen, including an illegal instruction exception. Additional errors could show up as an exception, a hardware error, or an instruction that is valid but different than the one that was expected.

WORKAROUND:
Place a **cli** before all **SSYNC**, **CSYNC**, "**LCx =**", "**LTx =**", and "**LBx =**" instructions to disable interrupts, and place an **sti** after each of these instructions to re-enable interrupts. When these instructions are executed in code that is already non-interruptible, the problem will not occur.

In an interrupt service routine that will enable interrupt nesting, be sure to push the LCx, LTx, and LBx registers before pushing RETI, which enables interrupt nesting. Following the inverse during the ISR context restore will guarantee that RETI is popped before the loop registers are loaded, thus disabling nested interrupts and protecting the loads from this anomaly situation. For example:

```
INT_HANDLER:
   [--sp] = astat;
   [--sp] = lc0; // push loop registers before pushing RETI
   [--sp] = lt0;
   [--sp] = lb0;
   [--sp] = lc1;
   [--sp] = lt1;
   [--sp] = lb1;
   [--sp] = reti; // push RETI to enable nested interrupts
   [--sp] = ...
      // body of interrupt handler
   ...  = [sp++];
   reti = [sp++]; // pop RETI to disable interrupts
   lb1 = [sp++];  // it is now safe to load the loop registers
   lt1 = [sp++];
   lc1 = [sp++];
   lb0 = [sp++];
   lt0 = [sp++];
   lc0 = [sp++];
   astat = [sp++];
```

Finally, as the workaround involves Supervisor Mode instructions to disable and enable interrupts, this does not apply to User Mode. In user space, do not use **CSYNC** or **SSYNC** instructions. Also, do not load the loop registers directly. Instead, utilize hardware loops which can be implemented with the **LSETUP** instruction, which limits loop ranges to 2046 bytes.

This workaround may be built into the development tool chain and/or into the operating system source code. For tool chains and operating systems supported by Analog Devices (VisualDSP++, VDK, CrossCore Embedded Studio, the GNU Tool Chain, and the Linux kernel), please consult the "Silicon Anomaly Tools Support" help page in the applicable documentation and release notes for details.

For all other tool chains and operating systems, see the appropriate supporting documentation for details.

APPLIES TO REVISION(S):
0.2, 0.3

40. 05000313 - PPI Is Level-Sensitive on First Transfer In Single Frame Sync Modes:

DESCRIPTION:
When the PPI is configured to trigger on a single external frame sync, all of the transfers require an edge on the frame sync except for the first transfer. For the first transfer only, the frame sync input is level-sensitive. This will make the PPI begin a transfer if the frame sync is at the active state, which can cause the PPI to start prematurely.

This anomaly does not apply when the PPI uses 2 or 3 frame syncs.

WORKAROUND:
When using a single external frame sync with the PPI, ensure that the frame sync is in the inactive state when the PPI is enabled.

APPLIES TO REVISION(S):
0.2, 0.3

41. 05000315 - Killed System MMR Write Completes Erroneously on Next System MMR Access:

DESCRIPTION:
Consider the following sequence:
1) System MMR write is stalled.
2) Interrupt/Exception occurs while the System MMR write is stalled (thus killing the write).
3) Interrupt/Exception Service Routine accesses (either read or write) any system MMR.

In order for this anomaly to happen, the change in program flow must kill the write in one particular stage of the execution pipeline. In this case, the anomaly will cause the MMR logic to think that the killed System MMR access is still valid. The following access (read/write) to the System MMR in the handler will cause the previously stalled write to complete erroneously.

Similarly, if the System MMR write is killed by an instruction itself, such as a conditional branch, the erroneous write can happen if the store buffer is full and emptying out to slow external memory.

```
cc = r0 == r0;   // always true
if cc jump skip;
W[p0] = r1.l;    // System MMR access is fetched and killed
skip: ...
```

NOTE: if the processor is halted in the handler before the next System MMR access via the debugging tools, the processor will stall indefinitely waiting for the write to complete, thus locking out the Emulation event.

WORKAROUND:
The workaround is to reset the MMR logic with another killed System MMR access in the branch's shadow. For example, setting up a read from the System MMR CHIPID register and subsequently killing it will create a killed access that has no other side-effects on the system. Therefore, the following code snippet, executed at the beginning of each handler routine, will work around this anomaly:

```
cc = r0 == r0;   // always true
p0.h = 0xffc0;   // System MMR space CHIPID
p0.l = 0x0014;
if cc jump skip; // always skip System MMR access, but it is fetched and killed
r0 = [p0];       // bogus System MMR read to work around the anomaly
skip: ...        // continue with handler code
```

In the case of System MMR writes being killed by the conditional branches, it is sufficient to insert 2 NOPs or any other non-MMR instructions in the location immediately after the conditional branch.

NOTE: in order to prevent lock-ups during debug sessions, always insert a desired breakpoint *after* the above code snippet if you need to halt the processor in the handler.

APPLIES TO REVISION(S):
0.2

42. 05000316 - EMAC RMII Mode: Collisions Occur in Full Duplex Mode:

DESCRIPTION:
In the full duplex operation of RMII mode, the assertion of both TX_EN and CRS_DV results in a collision detection, which means the outbound data and inbound data are available at the same time. It forces the transmitter to back off for some time and try again. This problem may compromise the transmission bandwidth.

WORKAROUND:
None.

APPLIES TO REVISION(S):
0.2

43. 05000321 - EMAC RMII Mode: TX Frames in Half Duplex Fail with Status "No Carrier":

DESCRIPTION:
In the half duplex operation of RMII mode, the No Carrier bit (CSR) of the EMAC_TX_STAT register is always set (no carrier). This prevents the TX frames from being reported with status TX_OK. As a result, the software may not be aware that its TX frames have been sent. This problem comes from the dilemma that the PHY must deassert CRS_DV to avoid collision while the same signal is used for TX_CRS during transmission. This same reason may also lead to a report of loss of carrier in the TX status word (TX_LOSS bit is set).

WORKAROUND:
None.

APPLIES TO REVISION(S):
0.2

44. 05000322 - EMAC RMII Mode at 10-Base-T Speed: RX Frames Not Received Properly:

DESCRIPTION:
The Ethernet MAC programmed to operate in RMII mode at 10-Base-T speed is unable to decode RX Frames properly.

WORKAROUND:
Use high speed (100-Base-T) and/or a MII connection for networking.

APPLIES TO REVISION(S):
0.2, 0.3

45. 05000341 - Ethernet MAC MDIO Reads Do Not Meet IEEE Specification:

DESCRIPTION:
When reading data from PHY registers via the Ethernet MAC (EMAC) MII interface, the EMAC latches the MDIO signal sourced by the PHY at the rising edge of the System Clock following the falling edge of MDC. The IEEE 802.3 standard stipulates that the PHY drive the MDIO at the rising edge of MDC with a 0 - 300ns output delay. In certain combinations of SCLK/MDC clocks and circuit routing, the EMAC may not be able to latch the correct MDIO data value.

WORKAROUND:
The following condition must be met to ensure the EMAC latches the correct MDIO data:

2 * Tprop + 300ns (or the actual PHY delay) < 0.5 * MDC period + tck

where:
- Tprop is the propagation time between the PHY and the Blackfin MII.
- tck is the delay between MDC falling edge and SCLK rising edge (-0.5 * SCLK period + 4ns).

To achieve this requirement, SCLK and MDC can be adjusted by programming the the PLL_DIV and EMAC_SYSCTL registers, respectively.

APPLIES TO REVISION(S):
0.3

46. 05000350 - UART Gets Disabled after UART Boot:

DESCRIPTION:
The boot kernel disables the UART port after a successful boot sequence over the UART completes. Consequently, the UART0_DLL and UART0_DLH registers are reset and, therefore, the application does not have access to the bit rate information obtained during the autobaud sequence at boot time. This behavior prohibits the processor from sending an acknowledgement to the host that the UART boot has completed successfully, which is common practice.

WORKAROUND:
The application must again perform the UART baud detection routine to re-establish a link to the UART host.

APPLIES TO REVISION(S):
0.2

47. 05000355 - Regulator Programming Blocked when Hibernate Wakeup Source Remains Active:

DESCRIPTION:
After the processor is placed into the hibernate state, a subsequent peripheral wakeup event can take the part out of hibernate. If that wakeup source remains asserted throughout the initiated reset sequence, writes to the VR_CTL register will not get latched into the regulator when the **IDLE** sequence is executed. Latching into the regulator circuit will be blocked until the wakeup source de-asserts.

The write will effectively be lost, however, the written value will go to the VR_CTL register's physical memory-mapped address. If no further writes to VR_CTL are performed, the "lost" write will be latched into the regulator hardware when the next **IDLE** instruction is executed.

WORKAROUND:
Ensure that the peripheral hibernate wakeup source de-asserts before attempting to program the regulator via the **IDLE** sequence required after the write to VR_CTL.

APPLIES TO REVISION(S):
0.2, 0.3

48. 05000357 - Serial Port (SPORT) Multichannel Transmit Failure when Channel 0 Is Disabled:

DESCRIPTION:
When configured in multi-channel mode with channel 0 disabled, DMA transmit data will be sent to the wrong SPORT channel if all of the following criteria are met:

1) External Receive Frame Sync (IRFS = 0 in SPORTx_RCR1)
2) Window Offset = 0 (WOFF = 0 in SPORTx_MCMC1)
3) Multichannel Frame Delay = 0 (MFD = 0 in SPORTx_MCMC2)
4) DMA Transmit Packing Disabled (MCDTXPE = 0 in SPORTx_MCMC2)

When this specific configuration is used, the multi-channel transmit data gets corrupted because whatever is in the channel 0 placeholder in non-packed mode gets sent first, even though channel 0 is disabled. The result is a one-word data shift in the output window, which repeats for each subsequent window in the serial stream. For example, if the non-packed transmit buffer is {0, 1, 2, 3, 4, 5, 6, 7}, and the window size is 8 channels with channel 0 disabled and channels 1-7 enabled to transmit, the expected data sequence in a series of output windows is:

1234567--1234567--1234567--1234567

With this anomaly, the output looks like this instead:

0123456--7012345--6701234--5670123

WORKAROUND:
There are several possible workarounds to this:

1) Disable Multichannel Mode
2) Use Internal Receive Frame Syncs
3) Use a Multichannel Frame Delay > 0
4) Use a Window Offset > 0
5) Enable DMA Transmit Packing
6) Do not disable Channel 0

APPLIES TO REVISION(S):
0.2, 0.3

49. 05000366 - PPI Underflow Error Goes Undetected in ITU-R 656 Mode:

DESCRIPTION:
If the PPI port is configured in ITU-R 656 Output Mode, the FIFO Underrun bit (UNDR in PPI_STATUS) does not get set when a PPI FIFO underrun occurs. An underrun can happen due to limited bandwidth or the PPI DMA failing to gain access to the bus due to arbitration latencies.

WORKAROUND:
None.

APPLIES TO REVISION(S):
0.2, 0.3

50. 05000371 - Possible RETS Register Corruption when Subroutine Is under 5 Cycles in Duration:

DESCRIPTION:

The RTS instruction can fail to return correctly if placed within four execution cycles of the beginning of a subroutine. For example:

```
CALL STUB_CODE;
...
...
...
STUB_CODE:
   RTS;
```

When this happens, potential bit failures in RETS will cause the processor to vector to the wrong address, which can cause invalid code to be executed.

WORKAROUND:

If there are at least four execution cycles in the subroutine before the RTS, the CALL and RTS instructions can never align in the manner required to encounter this problem. Since a NOP is a 1-cycle instruction, the following is a safe workaround for all potential failure cases:

```
CALL STUB_CODE;
...
...
...
STUB_CODE:
   NOP;      // These 4 NOPs can be any combination of instructions
   NOP;      // that results in at least 4 core clock cycles.
   NOP;
   NOP;
   RTS;
```

Branch prediction does not factor into this scenario. Conditional jumps within the subroutine that arrive at the RTS instruction inside of 4 cycles will not result in the scenario required to cause this failure. Asynchronous events (interrupts, exceptions, and NMI) are also not susceptible to this failure.

This workaround may be built into the development tool chain and/or into the operating system source code. For tool chains and operating systems supported by Analog Devices (VisualDSP++, VDK, CrossCore Embedded Studio, the GNU Tool Chain, and the Linux kernel), please consult the "Silicon Anomaly Tools Support" help page in the applicable documentation and release notes for details.

For all other tool chains and operating systems, see the appropriate supporting documentation for details.

APPLIES TO REVISION(S):

0.2, 0.3

51. 05000402 - SSYNC Stalls Processor when Executed from Non-Cacheable Memory:

DESCRIPTION:
Executing an SSYNC instruction from non-cacheable L2 memory with interrupts disabled can cause the processor to stall.

WORKAROUND:
If any interrupts are enabled, the stall will still occur, but it will be broken by the asynchronous event. If no interrupts are enabled or no interrupts are being generated, the stall is indefinite and the processor must be reset.

To avoid the stall condition, the following conditions must be met.

1) The SSYNC is in L1 memory or in cacheable L2 memory.
2) The SSYNC is not at a loop bottom where the loop top is located in non-cacheable L2 memory.
3) If the SSYNC is located in a cacheable L2 page, it is at least eight 64-bit words away from the bottom of the page (as specified by a CPLB) if the following (address sequential) page is either L1 or non-cacheable L2 memory.

If any of the above conditions is not met, another workaround would be to configure one of the timers prior to the SSYNC instruction with a time-out period to generate an interrupt and break the stall.

APPLIES TO REVISION(S):
0.2

52. 05000403 - Level-Sensitive External GPIO Wakeups May Cause Indefinite Stall:

DESCRIPTION:
When level-sensitive GPIO events are used to wake the processor from the low-power sleep mode of operation, the processor may stall indefinitely if the width of the wakeup pulse is too short. When this occurs, the PLL begins transitioning from the sleep mode due to the level sensed on the GPIO pin, but then reverts back to the sleep mode if the trigger level is removed before the core has had sufficient time to break the idle state to resume execution.

As a result, the processor does not wake up properly, at which point only a hardware reset can exit the resulting stall condition.

WORKAROUND:
There are two ways to avoid this anomaly:

1) Use edge-sensitivity for the pin(s) being used to generate the wakeup event.
2) Ensure that the edge on the wakeup signal is clean and held at the trigger level for at least 3 system clock (SCLK) cycles.

APPLIES TO REVISION(S):
0.2, 0.3

53. 05000416 - Speculative Fetches Can Cause Undesired External FIFO Operations:

DESCRIPTION:

When an external FIFO device is connected to an asynchronous memory bank, memory accesses can be performed by the processor speculatively, causing improper operations because the FIFO will provide data to the Blackfin, and the data will be dropped whenever the fetch is made speculatively or if the speculative access is canceled. "Speculative" fetches are reads that are started and killed in the pipeline prior to completion. They are caused by either a change of flow (including an interrupt or exception) or when performing an access in the shadow of a branch. This behavior is described in the Blackfin Programmer's Reference.

Another case that can occur is when the access is performed as part of a hardware loop, where a change of flow occurs from an exception. Since exceptions can't be disabled, the following example shows how an exception can cause a speculative fetch, even with interrupts disabled:

```
CLI R3;                              /* Disable Interrupts */
LSETUP( loop_s, loop_e) LC0 = P2;
   loop_s: R0 = W[P0];               /* Read from a FIFO Device  */
   loop_e: W[P1++] = R0;             /* Write that Generates a Data CPLB Page Miss */
STI R3;                              /* Enable Interrupts */
RTS;
```

In this example, the read inside the hardware loop is made to a FIFO with interrupts disabled. When the write inside the loop generates a data CPLB exception, the read inside the loop will be done speculatively.

WORKAROUND:

First, if the access is being performed with a core read, turn off interrupts prior to doing the core read. The read phase of the pipeline must then be protected from seeing the read instruction before interrupts are turned off:

```
CLI R0;
NOP; NOP; NOP;  /* Can Be Any 3 Instructions */
R1 = [P0];
STI R0;
```

To protect against an exception causing the same undesired behavior, the read must be separated from the change of flow:

```
CLI R3;                              /* Disable Interrupts */
LSETUP( loop_s, loop_e) LC0 = P2;
   loop_s: NOP;                      /* 2 NOPs to Pad Read */
           NOP;
           R0 = W[P0];
   loop_e: W[P1++] = R0;
STI R3;                              /* Enable Interrupts */
RTS;
```

The loop could also be constructed to place the NOP padding at the end:

```
LSETUP( .Lword_loop_s, .Lword_loop_e) LC0 = P2;
   .Lword_loop_s: R0 = W[P0];
                  W[P1++] = R0;
                  NOP;          /* 2 NOPs to Pad Read */
   .Lword_loop_e: NOP;
```

Both of these sequences prevent the change of flow from allowing the read to execute speculatively. The 2 inserted NOPs provide enough separation in the pipeline to prevent a speculative access. These NOPs can be any two instructions.

Reads performed using a DMA transfer do not need to be protected from speculative accesses.

APPLIES TO REVISION(S):

0.2, 0.3

54. 05000425 - Multichannel SPORT Channel Misalignment Under Specific Configuration:

DESCRIPTION:
When using the Serial Port in Multi-Channel Mode, the transmit and receive channels can get misaligned if a very specific configuration for the SPORT is met, as follows:

1) Window Offset (WOFF) = 0.
2) Window Size is an odd multiple of 8 (i.e., WSIZE is an even number > 0).
3) The time between RFS pulses is exactly equal to the window duration.

Note: The anomaly does NOT apply when WSIZE = 0.

When this exact configuration is used, the multi-channel mode channel enable registers are mislatched after the first window concludes, which results in the TDV signal being driven according to incorrect channel assignments and receive data being sampled on the wrong channels. So, the first window will send and receive properly, but all windows after the first will be misaligned, and data sent and received will be corrupted.

This error occurs for external and internal clocks and RFS.

WORKAROUND:
There are several workarounds possible:

1) Use a window offset other than 0.
2) Use a window size that is an even multiple of 8.
3) For internal RFS, make sure that SPORTx_RFSDIV is at least equal to the window size (# of enabled channels * SLEN).

APPLIES TO REVISION(S):
0.2, 0.3

55. 05000426 - Speculative Fetches of Indirect-Pointer Instructions Can Cause False Hardware Errors:

DESCRIPTION:

A false hardware error is generated if there is an indirect jump or call through a pointer which may point to reserved or illegal memory on the opposite control flow of a conditional jump to the taken path. This commonly occurs when using function pointers, which can be invalid (e.g., set to -1). For example:

```
    CC = P2 == -0x1;
    IF CC JUMP skip;
    CALL (P2);
  skip:
    RTS;
```

Before the IF CC JUMP instruction can be committed, the pipeline speculatively issues the instruction fetch for the address at -1 (0xffffffff) and causes the false hardware error. It is a false hardware error because the offending instruction is never actually executed. This can occur if the pointer use occurs within two instructions of the conditional branch (predicted not taken), as follows:

```
    BRCC X [predicted not taken]
    Y: JUMP (P-reg);  // If either of these two p-regs describe non-existent
       CALL (P-reg);  // memory, such as external SDRAM when the SDRAM
    X: RTS;           // controller is off, then a hardware error will result.
```

WORKAROUND:

If instruction cache is on or the ICPLBs are enabled, this anomaly does not apply.

If instruction cache is off and ICPLBs are disabled, the indirect pointer instructions must be 2 instructions away from the branch instruction, which can be implemented using NOPs:

```
    BRCC X [predicted not taken]
    Y: NOP;          // These two NOPs will properly pad the indirect pointer
       NOP;          // used in the next line.
       JUMP (P-reg);
       CALL (P-reg);
    X: RTS;
```

This workaround may be built into the development tool chain and/or into the operating system source code. For tool chains and operating systems supported by Analog Devices (VisualDSP++, VDK, CrossCore Embedded Studio, the GNU Tool Chain, and the Linux kernel), please consult the "Silicon Anomaly Tools Support" help page in the applicable documentation and release notes for details.

For all other tool chains and operating systems, see the appropriate supporting documentation for details.

APPLIES TO REVISION(S):

0.2, 0.3

56. 05000443 - IFLUSH Instruction at End of Hardware Loop Causes Infinite Stall:

DESCRIPTION:
If the IFLUSH instruction is placed on a loop end, the processor will stall indefinitely. For example, the following two code examples will never exit the loop:

```
P1 = 2;
LSETUP (LOOP1_S, LOOP1_E) LC1 = P1;
LOOP1_S: NOP;
LOOP1_E: IFLUSH[P0++];

LSETUP (LOOP2_S, LOOP2_E) LC1 = P1;
LOOP2_S: NOP; NOP; NOP; NOP;       // Any number of instructions...
LOOP2_E: IFLUSH[P0++];
```

WORKAROUND:
Do not place the IFLUSH instruction at the bottom of a hardware loop. If the IFLUSH is padded with any instruction at the bottom of the loop, the problem is avoided:

```
LSETUP (LOOP_S, LOOP_E) LC1 = P1;
LOOP_S: IFLUSH[P0++];
LOOP_E: NOP;                    // Pad the loop end
```

This workaround may be built into the development tool chain and/or into the operating system source code. For tool chains and operating systems supported by Analog Devices (VisualDSP++, VDK, CrossCore Embedded Studio, the GNU Tool Chain, and the Linux kernel), please consult the "Silicon Anomaly Tools Support" help page in the applicable documentation and release notes for details.

For all other tool chains and operating systems, see the appropriate supporting documentation for details.

APPLIES TO REVISION(S):
0.2, 0.3

57. 05000461 - False Hardware Error when RETI Points to Invalid Memory:

DESCRIPTION:

When using CALL/JUMP instructions targeting memory that does not exist, a hardware error condition will be triggered. If interrupts are enabled, the Hardware Interrupt (IRQ5) will fire. Since the RETI register will have an invalid location in it, it must be changed before executing the RTI instruction, even if servicing a different interrupt. Consider the following sequence:

```
P2.L = LO (0xFFAFFFFC);  // Load Address in Illegal Memory to P2
P2.H = HI (0xFFAFFFFC);
CALL(P2);                // Call to Bad Address Generates Hardware Error IRQ5
....

IRQ5_code:               // Hardware Error Interrupt Routine
RAISE 14;                // (1)
RTI;                     // (2)

IRQ14_code:
[--SP] = ( R7:0, P5:0 ); // (3)
[--SP] = RETI;           // (4)
....
```

When the hardware error occurs, the program counter points to the invalid location 0xFFAFFFFC, which is loaded into the RETI register during the service of the IRQ5 hardware error event. When the RTI instruction (2) is executed, a fetch of the instruction pointed to by the RETI register, which is an illegal address, is requested before hardware sees the level 14 interrupt pending. This fetch causes another hardware error to be latched, even though this instruction is not executed. Execution will go to IRQ14 (3). As soon as interrupts are re-enabled (4), the pending hardware error will fire.

WORKAROUND:

1) Ensure that code doesn't jump to or call bad pointers.
2) Always set the RETI register when returning from a hardware error to something that will not cause a hardware error on the memory fetch.

APPLIES TO REVISION(S):

0.2, 0.3

58. 05000462 - Synchronization Problem at Startup May Cause SPORT Transmit Channels to Misalign:

DESCRIPTION:
When the SPORT is configured in multichannel mode with an external SPORT clock, a synchronization problem may occur when the SPORT is enabled. This synchronization issue manifests when the skew between the external SPORT clock and the Blackfin processor's internal System Clock (SCLK) causes the channel counters inside the SPORT to get out-of-sync. When this occurs, a "dead" channel is inserted at the beginning of the window, and the rest of the transmit channels are right-shifted one location throughout the active window. The last channel data will be sent as the first enabled transmit channel data in the second window after another "dead" channel is inserted. All data will be sent sequentially and in its entirety, but it is transmitted on the wrong channels with respect to the frame sync and will never recover.

WORKAROUND:
When this error occurs, the SPORT must be restarted and checked again for this error. The failure is extremely rare to begin with, so the probability of seeing consecutive restarts showing the failure is infinitesimally small.

A software solution is possible based on the timing of the SPORT interrupt. In the SPORT ISR, the CYCLES register can be set to zero the first time the interrupt occurs and then read back the second time the interrupt occurs. This will provide a time reference in core clocks for the frequency of the SPORT interrupt itself. If the value read the second time exceeds the duration of the multichannel window (in core clocks), then a "dead" channel was inserted into the stream, and the SPORT must be restarted.

Hardware workarounds are going to be heavily dependent on how the multichannel mode SPORT is configured. In multichannel mode, TFS functions as a Transmit Data Valid (TDV) signal and will always be driven to the active state (as governed by the LTFS bit in the SPORTx_TCR1 register) during transmit channels. Therefore, the TDV signal can be routed to one of the GPIO pins configured to generate an interrupt upon detection of the TDV pin changing states, based upon how the application configures the channels within the active frame, to detect the "dead" channel. If all the channels in the window are configured as transmit channels and there is no window offset and no multichannel frame delay, then TDV should go active as soon as the RFS pulse is received. If the period of the RFS pulse is exactly the window size (i.e., there are no extra clocks after the active window before the next RFS is detected), then TDV will remain active throughout operation. Therefore, if TDV goes inactive while the SPORT is on, the failure happened and the SPORT must be restarted and run again with this test in place until the failure is not detected.

For applications that have a window offset, a multichannel frame delay, extra clocks between the end of the active window and the next frame sync, and/or non-transmit channels inside the active window, the first TDV assertion would need to be tracked manually to detect the "dead" channel. One idea might be to do the following:

1) Connect TFS (TDV) to a GPIO interrupt and configure the interrupt to occur when TDV goes active.
2) Connect RFS to a GPIO interrupt and configure the interrupt to occur when RFS goes active.
3) Connect the SPORT receive clock to a TMRx pin configured in EXT_CLK mode.

When the GPIO interrupt for the active RFS pulse signifying the start of the window occurs, enable the Timer that is being used to track the SPORT receive clock. When the GPIO interrupt for the TDV signal transition occurs, check the TIMERx_COUNTER register to determine how many SPORT clocks have passed since the frame started. If it is one channel's worth over the expected value, the error occurred and the SPORT must be restarted and tested again. The GPIO interrupts should also be disabled if the startup condition is not detected.

APPLIES TO REVISION(S):
0.2, 0.3

59. 05000473 - Interrupted SPORT Receive Data Register Read Results In Underflow when SLEN > 15:

DESCRIPTION:
A SPORT receive underflow error can be erroneously triggered when the SPORT serial length is greater than 16 bits and an interrupt occurs as the access is initiated to the 32-bit SPORTx_RX register. Internally, two accesses are required to obtain the 32-bit data over the internal 16-bit Peripheral Access Bus, and the anomaly manifests when the first half of the access is initiated but the second is held off due to the interrupt. Application code vectors to service the interrupt and then issues the read of the SPORTx_RX register again when it subsequently resumes execution after the interrupt has been serviced. The previous read that was interrupted is still pending awaiting the second half of the 32-bit access, but the SPORT erroneously sends out two requests again. The first access completes the previous transaction, and the second access generates the underflow error, as it is now attempting to make a read when there is no new data present.

WORKAROUND:
The anomaly does not apply when using valid serial lengths up to 16 bits, so setting SLEN < 16 is one workaround.

When the length of the serial word is 17-32 bits (16 <= SLEN < 32), accesses to the SPORTx_RX register must not be interrupted, so interrupts must be disabled around the read. In C:

```
int temp_IMASK;

temp_IMASK = cli();
RX_Data = *pSPORT0_RX;
sti(temp_IMASK);
```

In assembly:

```
P0.H = HI(SPORT0_RX);
P0.L = LO(SPORT0_RX);

CLI R0;
R1 = [P0];
STI R0;
```

APPLIES TO REVISION(S):
0.2, 0.3

60. 05000475 - Possible Lockup Condition when Modifying PLL from External Memory:

DESCRIPTION:
Synchronization logic in the EBIU can get corrupted if PLL alterations are made by code that resides in external memory. When this occurs, an infinite stall will occur, and the part will need to be reset. The lockup is dependent on what the original ratio was, what the new ratio is, and other factors, thus making it impossible to specify any cases where this is safe.

WORKAROUND:
The CCLK::SCLK ratio should not be changed via the external interface, whether it's from asynchronous memory or SDRAM. Only make modifications to the PLL_CTL and PLL_DIV registers from code executing in on-chip memory.

APPLIES TO REVISION(S):
0.2, 0.3

61. 05000477 - TESTSET Instruction Cannot Be Interrupted:

DESCRIPTION:
When the TESTSET instruction gets interrupted, the write portion of the TESTSET may be stalled until after the interrupt is serviced. After the ISR completes, application code continues by reissuing the previously interrupted TESTSET instruction, but the pending write operation is completed prior to the new read of the TESTSET target data, which can lead to deadlock conditions.

For example, in a multi-threaded system that utilizes semaphores, thread A checks the availability of a semaphore using TESTSET. If this original TESTSET operation tested data with a low byte of zero (signifying that the semaphore is available), then the write portion of TESTSET sets the MSB of the low byte to 1 to lock the semaphore. When this anomaly occurs, the write doesn't happen until TESTSET is re-issued after the interrupt is serviced. Therefore, thread A writes the byte back out with the lock bit set and then immediately reads that value back, now erroneously indicating that the semaphore is locked. Provided the semaphore was actually still free when TESTSET was reissued, this means that the semaphore is now permanently locked because thread A thinks it was locked already, and any other threads that subsequently pend on the same semaphore are being locked out by thread A, which will now never release it.

WORKAROUND:
The TESTSET instruction must be made uninterruptible to avoid this condition:

```
CLI R0;
TESTSET(P0);
STI R0;
```

There is no workaround other than this, so events that cannot be made uninterruptible, such as an NMI or an Emulation event, will always be sensitive to this issue. Additionally, due to the need to disable interrupts, User Mode code cannot implement this workaround.

This workaround may be built into the development tool chain and/or into the operating system source code. For tool chains and operating systems supported by Analog Devices (VisualDSP++, VDK, CrossCore Embedded Studio, the GNU Tool Chain, and the Linux kernel), please consult the "Silicon Anomaly Tools Support" help page in the applicable documentation and release notes for details.

For all other tool chains and operating systems, see the appropriate supporting documentation for details.

APPLIES TO REVISION(S):
0.2, 0.3

62. 05000480 - Multiple Simultaneous Urgent DMA Requests May Cause DMA System Instability:

DESCRIPTION:
In a system with several DMA channels active, a race condition exists where a channel may go urgent and cause a granted DMA channel to be overridden by the urgent channel after the granted channel has begun a transaction. When this occurs, one word from the granted channel's data erroneously goes to the urgent channel's peripheral. This can be a single word of data shuffled between the two peripherals or a single word from the granted peripheral's DMA data inserted as the first word in the urgent channel's DMA descriptor fetch. When the descriptor fetch gets corrupted by this word, the rest of the descriptor fetch is completed offset by one word, and all of the DMA registers associated with the descriptor are written with incorrect values.

The data swap between two channels may be difficult to detect, but the descriptor fetch failure can lead to several failure conditions that are easily seen, such as DMA errors for accessing illegal memory and/or having an illegal configuration register.

WORKAROUND:
Program the DMA Traffic Control Period to a non-zero value. This forces the DMA block to group accesses together rather than allow arbitration for each piece of data placed on the internal DMA bus.

APPLIES TO REVISION(S):
0.2

63. 05000481 - Reads of ITEST_COMMAND and ITEST_DATA Registers Cause Cache Corruption:

DESCRIPTION:
Reading the ITEST_COMMAND or ITEST_DATA registers will erroneously trigger a write to these registers in addition to reading the current contents of the register. The erroneous write does not update the read state of the register, however, the data written to the register is acquired from the most recent MMR write request, whether the most recent MMR write request was committed or speculatively executed. The bogus write can set either register to perform unwanted operations that could result in:

1) Corrupted instruction L1 memory and/or instruction TAG memory.
and/or
2) Garbled instruction fetch stream (stale data used in place of new fetch data).

WORKAROUND:
Never read ITEST_COMMAND or ITEST_DATA. The only exception to this strict workaround is in the case of performing the read atomically and immediately after a write to the same register. In this case, the erroneous write will still occur, but it will be with the exact same data as the intentional write that preceded it.

APPLIES TO REVISION(S):
0.2, 0.3

64. 05000489 - PLL May Latch Incorrect Values Coming Out of Reset:

DESCRIPTION:
It is possible that the PLL can latch incorrect SSEL and CSEL values during reset when VDDINT is powered before VDDEXT. If this problem occurs, the PLL_DIV register will show the correct default value when read via software, but the actual SSEL and CSEL values being provided to the PLL may be incorrect. This results in different values for the core and system clocks from what the default values would be coming out of reset. If this problem occurs, the most likely result will be system and core clocks that are not the default (CCLK = 10xCLKIN, SCLK = 2xCLKIN), which will be corrected when the application programs the PLL to the desired frequencies. However, the random nature of the values latched could lead to the PLL getting illegally programmed, which can cause the boot process to fail.

WORKAROUND:
There are a few workarounds for this issue. Any one of the following will avoid the issue:

1) Use the on-chip regulator.
2) Issue a second hardware reset after the power-on reset.
3) Ensure that VDDEXT reaches at least the Vddext minimum specification before turning on VDDINT.
4) If powering VDDINT first, keep $\overline{\text{RESET}}$ de-asserted until after VDDEXT has been established, then assert $\overline{\text{RESET}}$ per the power-on reset specification.

It is extremely unlikely that this anomaly will occur. If it has not been observed in existing designs, it is recommended that one of the above workarounds be implemented at the next logical point of the design cycle. For systems in development, implementing one of the above workarounds is strongly encouraged.

APPLIES TO REVISION(S):
0.2, 0.3

65. 05000491 - Instruction Memory Stalls Can Cause IFLUSH to Fail:

DESCRIPTION:
When an instruction memory stall occurs when executing an IFLUSH instruction, the instruction may fail to invalidate a cache line. This could be a problem when replacing instructions in memory and could cause stale, incorrect instructions in cache to be executed rather than initiating a cache line fill.

WORKAROUND:
Instruction memory stalls must be avoided when executing an IFLUSH instruction. By placing the IFLUSH instruction in L1 memory, the prefetcher will not cause instruction cache misses that could cause memory stalls. In addition, padding the IFLUSH instruction with NOPs will ensure that subsequent IFLUSH instructions do not interfere with one another, and wrapping SSYNCs around it ensures that any fill/victim buffers are not busy. The recommended routine to perform an IFLUSH is:

```
SSYNC;          // Ensure all fill/victim buffers are not busy
LSETUP (LS, LE)
LS:  IFLUSH;
     NOP;
     NOP;
LE:  NOP;
SSYNC;          // Ensure all fill/victim buffers are not busy
```

Since this loop is four instructions long, the entire loop fits within one loop buffer, thereby turning off the prefetcher for the duration of the loop and guaranteeing that successive IFLUSH instructions do not interfere with each other.

APPLIES TO REVISION(S):
0.2, 0.3

66. 05000494 - EXCPT Instruction May Be Lost If NMI Happens Simultaneously:

DESCRIPTION:

A software exception raised by issuing the EXCPT instruction may be lost if an NMI event occurs simultaneous to execution of the EXCPT instruction. When this precise timing is met, the program sequencer believes it is going to service the EXCPT instruction and prepares to write the address of the next sequential instruction after the EXCPT instruction to the RETX register. However, the NMI event takes priority over the Exception event, and this address erroneously goes to the RETN register. As such, when the NMI event is serviced, program execution incorrectly resumes at the instruction after the EXCPT instruction rather than at the EXCPT instruction itself, so the software exception is lost and is not recoverable.

WORKAROUND:

Either do not use NMI or protect against this lost exception by forcing the exception to be continuously re-raised and verified in the exception handler itself. For example:

```
EXCPT 0;
JUMP -2;  // add this jump -2 after every EXCPT instruction
```

Then, in the exception handler code, read the EXCAUSE field of the SEQSTAT register to determine the cause of the exception.
If EXCAUSE < 16, the handler was invoked by execution of the EXCPT instruction, so the RETX register must then be modified to skip over the JUMP -2 that was inserted in the workaround code:

```
R2 = SEQSTAT;
R2 <<= 0x1A;
R2 >>= 0x1A;   // Mask Everything Except SEQSTAT[5:0] (EXCAUSE)
R1 = 0xF (Z);
CC = R2 <= R1; // Check for EXCAUSE < 16
IF !CC JUMP CONTINUE_EX_HANDLER;
R2 = RETX;
R2 += 2;       // Modify RETX to Point to Instruction After Inserted JUMP -2;
RETX = R2;
JUMP END_EX_HANDLER;

CONTINUE_EX_HANDLER: // Rest of Exception Handler Code Goes Here
   .
   .
   .
END_EX_HANDLER: RTX;
```

In this fashion, the JUMP -2 guarantees that the soft exception is re-raised when this anomaly occurs. When the NMI does not occur, the above exception handler will redirect the application code to resume after the JUMP -2 workaround code that re-raises the exception.

A workaround may be built into the development tool chain and/or into the operating system source code. For tool chains and operating systems supported by Analog Devices (VisualDSP++, VDK, the GNU Tool Chain, and the Linux kernel), please consult the "Silicon Anomaly Tools Support" help page in the applicable documentation and release notes for details.

For all other tool chains and operating systems, see the appropriate supporting documentation for details.

APPLIES TO REVISION(S):

0.2, 0.3

67. 05000501 - RXS Bit in SPI_STAT May Become Stuck In RX DMA Modes:

DESCRIPTION:
When in SPI receive DMA modes, the RXS bit in SPI_STAT can get set and erroneously get stuck high if the SPI port is disabled as hardware is updating the status of the RXS bit. When in RX DMA mode, RXS will set as a word is transferred from the shift register to the internal FIFO, but it is then automatically cleared immediately by the hardware as DMA drains the FIFO. However, there is an internal 2 system clock (SCLK) latency for the status register to properly reflect this. If software disables the SPI port in exactly this window of time before RXS is cleared, the RXS bit doesn't get cleared and will remain set, even after the SPI is disabled. If the SPI port is subsequently re-enabled, the set RXS bit will cause one of two problems to occur:

1) If enabled in core RX mode, the SPI RX interrupt request will be raised immediately even though there is no new data in the SPI_RDBR register.
2) If enabled in RX DMA mode, DMA requests will be issued, which will cause the processor to DMA data from the SPI FIFO even though there is actually no new data present.

In master mode, the SPI will continue issuing clocks after RX DMA is completed until the SPI port is disabled. If any SPI word is received exactly as software disables the SPI port, the problem will occur.

In slave mode, the host would have to continue providing clocks and the chip-select for this possibility to occur.

WORKAROUND:
Reading the SPI_RDBR register while the SPI is disabled will clear the stuck RXS condition and not trigger any other activity. If using RX DMA mode, be sure to include this dummy read after the SPI port disable.

APPLIES TO REVISION(S):
0.2, 0.3

68. 05000503 - SPORT Sign-Extension May Not Work:

DESCRIPTION:
In multichannel receive mode, the SPORT sign-extension feature (RDTPYE=b#01 in SPORTx_RCR1) is not reliable for channel 0 data when configured for MSB-first data reception. This is regardless of any channel offset and/or multichannel frame delay.

WORKAROUND:
1) If possible, use receive bit order of LSB-first.
2) Do not use channel 0.
3) Ignore channel 0 data.
4) Use software to manually apply sign extension to the channel 0 data before processing.

APPLIES TO REVISION(S):
0.2, 0.3

69. 05000506 - Hardware Loop Can Underflow Under Specific Conditions:

DESCRIPTION:
When two consecutive hardware loops are separated by a single instruction, and the two hardware loops use the same loop registers, and the first loop contains a conditional jump to its loop bottom, the first hardware loop can underflow. For example:

```
P0 = 16;
LSETUP(loop_top1, loop_bottom1) LC0 = P0;
   loop_top1:    nop;
                 if CC JUMP loop_bottom1;
                 nop;
                 nop;
   loop_bottom1: nop;

nop;                          // Any single instruction

LSETUP(loop_top2, loop_bottom2) LC0 = P0;
   loop_top2:    nop;
   loop_bottom2: nop;
```

If a stall occurs on the instruction that is between the two loops, the top loop can decrement its loop count from 0 to 0xFFFFFFFF and continue looping with the incorrect loop count.

WORKAROUND:
There are several workarounds to this issue:

1) Do not use the same loop register set in consecutive hardware loops.
2) Ensure there is not exactly one instruction between consecutive hardware loops.
3) Ensure the first loop does not conditionally jump to its loop bottom.

APPLIES TO REVISION(S):
0.2, 0.3