ANALOG DEVICES

Blackfin Embedded Processor

Silicon Anomaly List

ADSP-BF606/BF607/BF608/BF609

ABOUT ADSP-BF606/BF607/BF608/BF609 SILICON ANOMALIES

These anomalies represent the currently known differences between revisions of the Blackfin® ADSP-BF606/BF607/BF608/BF609 product(s) and the functionality specified in the ADSP-BF606/BF607/BF608/BF609 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 REVID bits <31:28> of the SDU_IDCODE register can be used to differentiate the revisions as shown below.

Silicon REVISION	SDU_IDCODE.REVID
0.1	0x1
0.2	0x2

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
08/04/2017	I	A	Added Anomalies: 16000048, 16000049
01/25/2016	H	A	Added Anomaly: 16000047
03/28/2015	G	A	Added Anomalies: 16000044, 16000045, 16000046
01/31/2014	F	0	Removed Silicon Revision 0.0 Added Silicon Revision 0.2 Added Anomalies: 16000043
11/08/2013	E	0	Added Anomalies: 16000041, 16000042
05/23/2013	D	PrG	Added Silicon Revision 0.1 Added Anomalies: 16000039, 16000040
01/15/2013	C	PrF	Added Anomalies: 16000024, 16000025, 16000033

NR004212I **Document Feedback**

One Technology Way, P.O.Box 9106, Norwood, MA 02062-9106 U.S.A.
Tel: 781.329.4700 ©2017 Analog Devices, Inc. All rights reserved.
Technical Support www.analog.com

SUMMARY OF SILICON ANOMALIES

The following table provides a summary of ADSP-BF606/BF607/BF608/BF609 anomalies and the applicable silicon revision(s) for each anomaly.

No.	ID	Description	Rev 0.1	Rev 0.2
1	16000003	TRU_STAT.ADDRERR and TRU_ERRADDR.ADDR May Not Reflect the Correct Status	x	x
2	16000007	DDR2 Memory Reads May Fail Intermittently	x	x
3	16000008	Instruction Memory Stalls Can Cause IFLUSH to Fail	x	x
4	16000009	TESTSET Instruction Cannot Be Interrupted	x	x
5	16000010	IFLUSH Instruction at End of Hardware Loop Causes Infinite Stall	x	x
6	16000011	False Hardware Error when RETI Points to Invalid Memory	x	x
7	16000012	Speculative Fetches of Indirect-Pointer Instructions Can Cause False Hardware Errors	x	x
8	16000013	False Hardware Errors Caused by Fetches at the Boundary of Reserved Memory	x	x
9	16000014	False Hardware Error from an Access in the Shadow of a Conditional Branch	x	x
10	16000015	Multi-Issue Instruction with dsp32shiftimm in slot1 and P-reg Store in slot2 Not Supported	x	x
11	16000017	Speculative Fetches Can Cause Undesired External FIFO Operations	x	x
12	16000024	Spurious Fault Signaled After Clearing an Externally Generated Fault	x	x
13	16000025	SPORT May Drive Data Pins During Inactive Channels in Multichannel Mode	x	x
14	16000033	PPI Data Underflow on First Word Not Reported in Certain Modes	x	x
15	16000039	CGU_STAT.PLOCKERR Bit May be Unreliable	x	x
16	16000040	JTAG Emulator Reads of SDU_IDCODE Alter Register Contents	x	x
17	16000041	IFLUSH Instruction Causes Parity Error When Parity Is Enabled	x	.
18	16000042	Instruction Cache Failure When Parity Is Enabled	x	.
19	16000043	Hardware Loop Can Underflow Under Specific Conditions	x	x
20	16000044	In LPDDR Mode, Automatic DMC Initialization after Deep Power Down Mode May Fail	x	x
21	16000045	Lower 16 Bits of CNT_CNTR Register May Be Off by 1 under Certain Circumstances	x	x
22	16000046	DLL in the DMC PHY May Not Lock to New DCLK Frequency	x	x
23	16000047	SPI Master Boot Fails When Block Payload Size Exceeds 65,532 Bytes	x	x
24	16000048	DMC CAS Latency of 2 May Not Work As Expected	x	x
25	16000049	LPDDR DMC Access Failures when DLL Locks to Problematic Range of Values	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-BF606/BF607/BF608/BF609 including a description, workaround, and identification of applicable silicon revisions.

1. 16000003 - TRU_STAT.ADDRERR and TRU_ERRADDR.ADDR May Not Reflect the Correct Status:

DESCRIPTION:
TRU_STAT.ADDRERR and TRU_ERRADDR.ADDR show an error even if a valid address is written to the TRU address range.

WORKAROUND:
Do not depend on the values in TRU_STAT.ADDRERR and TRU_ERRADDR.ADDR.

APPLIES TO REVISION(S):
0.1, 0.2

2. 16000007 - DDR2 Memory Reads May Fail Intermittently:

DESCRIPTION:
DDR2 memory reads may fail intermittently using the DMC_TR2.TRTP (tRTP) minimum setting recommended in the Hardware Reference manual.

WORKAROUND:
Increase DMC_TR2.TRTP to a minimum of 5. If the DMC_TR2.TRTP calculation (using the memory manufacturer tRTP specification and the DCLK frequency) yields a result above 5, this higher DMC_TR2.TRTP value should be used.

APPLIES TO REVISION(S):
0.1, 0.2

3. 16000008 - 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):
0.1, 0.2

4. 16000009 - 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. The same applies to a semaphore that is shared between multiple cores within the same device.

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.

APPLIES TO REVISION(S):
0.1, 0.2

5. 16000010 - 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
```

APPLIES TO REVISION(S):
0.1, 0.2

6. 16000011 - 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):

0.1, 0.2

7. 16000012 - 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;
```

APPLIES TO REVISION(S):
0.1, 0.2

8. 16000013 - 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.

APPLIES TO REVISION(S):
0.1, 0.2

9. 16000014 - 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, 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):
0.1, 0.2

10. 16000015 - 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, 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):
0.1, 0.2

11. 16000017 - 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):

0.1, 0.2

12. 16000024 - Spurious Fault Signaled After Clearing an Externally Generated Fault:

DESCRIPTION:
A spurious fault will be indicated for a single SCLK0 period on SYS_FAULT and $\overline{\text{SYS\_FAULT}}$ under the following conditions:
1. SEC_FCTL.FIEN and SEC_FCTL.FOEN are set.
2. An external device signals a fault on SYS_FAULT or $\overline{\text{SYS\_FAULT}}$.
3. The external fault is cleared by writing 0x00010000 to SEC_FEND.

WORKAROUND:
Clear SEC_FCTL.FIEN before writing to SEC_FEND to clear an externally signaled fault:
1. SEC_FCTL.FIEN and SEC_FCTL.FOEN are set.
2. An external device signals a fault on SYS_FAULT or $\overline{\text{SYS\_FAULT}}$.
3. Clear SEC_FCTL.FIEN
4. Clear the external fault by writing 0x00010000 to SEC_FEND.
5. Set SEC_FCTL.FIEN

APPLIES TO REVISION(S):
0.1, 0.2

13. 16000025 - SPORT May Drive Data Pins During Inactive Channels in Multichannel Mode:

DESCRIPTION:
When a SPORT is operating in multichannel mode, the transmitter tri-states the data pins during the inactive channels. When SPMUX functionality is enabled, under specific conditions, one SPORT half may continue to drive on the inactive channels. This happens when all the below conditions are true.
1. SPORT half "x" is configured as transmitter (SPORT_CTL_x.SPTRAN = 1)
2. Imports Frame sync internally from the pairing half SPORT (SPORT_CTL2_x.FSMUXSEL = 1).
3. Multichannel Frame Delay is zero (SPORT_MCTL_x.MFD = 0)
4. Window Offset is zero (SPORT_MCTL_x.OFFSET = 0)
5. Channel-0 of multichannel frame is enabled for transmission (SPORT_CS0_x.CH0 = 1)
6. Frame sync is active low (SPORT_xCTL.LFS = 1)
7. Frame Sync edge Detect bit is 0 (SPORT_CTL_x.FSED = 0)

If any of these conditions is false, this anomaly does not occur.

When this exact configuration is used, after completion of all the active channels, the SPORT half transmitter drives the first bit of next word to be transmitted once the number of channels specified in WSIZE expires. Therefore the SPORT half may drive on inactive channels which can cause contention when other transmitters configured to drive on these inactive channels.

WORKAROUND:
Avoid any one of the above conditions. For example:
1. Set the Frame Sync edge Detect bit (SPORT_CTL_x.FSED = 1).
2. Use Window Offset other than zero.
3. Use Multichannel Frame Delay other than zero.

APPLIES TO REVISION(S):
0.1, 0.2

14. 16000033 - PPI Data Underflow on First Word Not Reported in Certain Modes:

DESCRIPTION:
In PPI transmit modes with external frame sync(s) PPI_STAT.CFIFOERR and PPI_STAT.YFIFOERR aren't reported reliably if the first data word underflows. If the second data word to be transmitted also underflows, PPI reports the underflow error correctly.

WORKAROUND:
Use the second data word underflow notification to take any required action or use a different PPI mode.

APPLIES TO REVISION(S):
0.1, 0.2

15. 16000039 - CGU_STAT.PLOCKERR Bit May be Unreliable:

DESCRIPTION:
The CGU_STAT.PLOCKERR bit may not be set in the case of the PLL failing to lock at the programmed frequency. However, as expected, CGU_STAT.PLOCKERR will not assert when a PLL lock is successful.

WORKAROUND:
Do not depend on CGU_STAT.PLOCKERR to detect failure of the PLL to lock. In the case of safety-critical systems SYS_CLKOUT or any other internally clocked output may be monitored to ensure that no software or hardware issue has prevented the PLL from locking at the programmed frequency.

APPLIES TO REVISION(S):
0.1, 0.2

16. 16000040 - JTAG Emulator Reads of SDU_IDCODE Alter Register Contents:

DESCRIPTION:
After connecting to the processor using an emulator debug session, the value stored in SDU_IDCODE will be read as 0x00000000 if the JTAG emulator does not reload the correct value after reading.

WORKAROUND:
There are three possible workarounds:

1. Read SDU_IDCODE before connecting with an emulator debug session.
2. Use a development tool chain that reloads the SDU_IDCODE after reading. For tool chains and operating systems supported by Analog Devices (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.
3. Read a different register to distinguish between silicon revisions, if such register differences (shown in this list) exist between the silicon revisions of interest.

APPLIES TO REVISION(S):
0.1, 0.2

17. 16000041 - IFLUSH Instruction Causes Parity Error When Parity Is Enabled:

DESCRIPTION:

Executing IFLUSH with instruction cache and parity enabled (assuming the valid bit is active on the cache line in question) will result in incorrect parity being generated due to the clearing of the valid bit in the cache TAG without changing the parity bit appropriately. The next time that location is accessed, a parity error will be generated.

This anomaly does not apply to data cache.

WORKAROUND:

The ITEST_COMMAND functionality can be used as a substitute for the IFLUSH instruction, as the cache tag parity is recalculated in hardware when the tag is written via the ITEST_COMMAND register. For example, consider the pseudo-code:

```
IFLUSH(flush_addr);
```

The above functionality can be achieved by manually snooping the cache tag arrays to locate where in the cache it resides and invalidating that line:

```
volatile char i;
int itest_cmd, itest_data0;

for(i = 0; i < 4; i++)
{
   itest_cmd = ((i<<26) | ((flush_addr & 0x00003000)<<4) | (flush_addr & 0x000003E0));
         /* ITEST_COMMAND[27:26] = WAY # (loop count i) */
         /* ITEST_COMMAND[17:16] = flush_addr[13:12]    */
         /* ITEST_COMMAND[09:05] = flush_addr[09:05]    */

   *pITEST_COMMAND = itest_cmd;       /* write to command register */
   ssync();                           /* must be followed by ssync */

   itest_data0 = *pITEST_DATA0;       /* Get Cache Tag */
   ssync();

   /* Check for valid bit AND address match (bits 31:14 and bits 11:10) */
   if( (itest_data0 & 1) && ( (itest_data0 & 0xFFFFCC00) == (flush_addr & 0xFFFFCC00) ) )
   {
      /* Clear bit 0 to invalidate the line */
      *pITEST_DATA0 = itest_data0 & 0xFFFFFFFE;
      ssync();

      /* Change command to cache tag write */
      *pITEST_COMMAND = itest_cmd | 0x00000002;
      ssync();
      break;    /* Weï¿½ve done what we need to do, so exit */
   } /* if no tag match or not valid, do nothing */
} /* if no ways hit, do nothing */
```

APPLIES TO REVISION(S):

0.1

18. 16000042 - Instruction Cache Failure When Parity Is Enabled:

DESCRIPTION:

When parity checking is enabled, an instruction cache line fill can be abandoned if a change in control occurs on the processor when the cache line fill is initiated with a particular cycle alignment relative to the cache fill initiation. When this failure occurs, the cache line is marked valid for the corresponding cache TAG, but the cache line fill itself does not happen. As a result, the processor executes the stale cached instruction rather than the one that should have been there had the new cache line been filled properly.

This anomaly does not apply to data cache.

WORKAROUND:

There is no workaround possible to keep both instruction cache and parity enabled. If either is disabled, this anomaly will not occur.

APPLIES TO REVISION(S):

0.1

19. 16000043 - 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):

0.1, 0.2

20. 16000044 - In LPDDR Mode, Automatic DMC Initialization after Deep Power Down Mode May Fail:

DESCRIPTION:
In the LPDDR mode, the DMC controller is expected to automatically reinitialize when exiting deep power down mode (the DMC_CTL.DPDREQ bit is cleared).

Due to this anomaly, the automatic initialization sequence may not work as expected, which may result in unpredictable behavior (e.g., data corruption) when trying to access the LPDDR memory space after exiting deep power down mode.

WORKAROUND:
A new initialization sequence must be performed in software after exiting deep power down mode:
1. Clear the DMC_CTL.DPDREQ bit to exit deep power down mode.
2. Wait for DMC_STAT.DPDACK to clear.
3. Wait for DMC_STAT.MEMINITDONE and DMC_STAT.DLLCALDONE to set, ensuring that the automatic initialization sequence has completed (this sequence does not work as expected, but these status bits get set properly).
4. Set the DMC_CTL.INIT bit to manually initialize the DMC.
5. Wait for DMC_STAT.MEMINITDONE and DMC_STAT.DLLCALDONE to set again, ensuring that the manual initialization sequence has completed.

The following C code can be used to implement the above sequence:

```
*pREG_DMC0_CTL &= ~BITM_DMC_CTL_DPDREQ;              // Step 1

while(*pREG_DMC0_STAT & BITM_DMC_STAT_DPDACK);       // Step 2

while(!(*pREG_DMC0_STAT & BITM_DMC_STAT_DLLCALDONE));   // Step 3
while(!(*pREG_DMC0_STAT & BITM_DMC_STAT_MEMINITDONE));

*pREG_DMC0_CTL |= BITM_DMC_CTL_INIT;                 // Step 4
while(!(*pREG_DMC0_STAT & BITM_DMC_STAT_DLLCALDONE));   // Step 5
while(!(*pREG_DMC0_STAT & BITM_DMC_STAT_MEMINITDONE));
```

APPLIES TO REVISION(S):
0.1, 0.2

21. 16000045 - Lower 16 Bits of CNT_CNTR Register May Be Off by 1 under Certain Circumstances:

DESCRIPTION:
When the CNT_CNTR register is read in the same SCLK cycle that the counter increments, the lower 16 bits of the register are not updated. Under nearly all conditions, the value read from CNT_CNTR will be one less than the correct value. However, in the case of the counter crossing a 16-bit boundary, the value read will be off by 0xffff from the expected value. Consider the case where the previous count value was 0x0000ffff. If the anomaly manifested on the next counter increment, the CNT_CNTR register will read as 0x0001ffff instead of 0x00010000.

WORKAROUND:
There is no workaround for the issue. For the case of the 16-bit boundary condition, software can check the lower 16 bits for 0xffff and read the CNT_CNTR register again to get a more accurate count.

APPLIES TO REVISION(S):
0.1, 0.2

22. 16000046 - DLL in the DMC PHY May Not Lock to New DCLK Frequency:

DESCRIPTION:
The DMC PHY DLL may not lock properly after the DCLK frequency is changed via the CGU. When this fails, the controller does not generate the required phase between the DQS0/DQS1 and DQn signals, which can result in data corruption when trying to access DDR memory.

This anomaly applies to both DDR2 and LPDDR.

WORKAROUND:
To avoid the issue, the DLL must be reset in software when changing the DCLK frequency. The following C code can be used as a workaround:

```
*pREG_DMC0_PHY_CTL0 |= 0x0800uL;
Init_cgu();                     /* Change clock settings */
*pREG_DMC0_PHY_CTL0 &= (~0x0800uL);
```

APPLIES TO REVISION(S):
0.1, 0.2

23. 16000047 - SPI Master Boot Fails When Block Payload Size Exceeds 65,532 Bytes:

DESCRIPTION:
When booting in SPI Master Mode, the boot ROM configures the SPI in 8-bit mode and uses the SPI Receive Counter register (**`SPI_RWC`**) to store the payload byte count. By definition, this 16-bit register can only accommodate payload sizes up to 64KB.

When the boot code transfers a payload greater than this, it is supposed to break the data into blocks of 64KB each by writing the **`SPI_RWC`** register to 64K, but it erroneously sets the count to 0 instead. When this occurs, the SPI port halts and the boot process stops.

WORKAROUND:
Use the **`-MaxBlockSize`** switch to limit the block size to a value less than 64K. As boot code needs to be aligned on a 32-bit boundary, the maximum size of any individual block is 65,532 (0xFFFC); therefore, use **`-MaxBlockSize 0xFFFC`** in the additional options while creating the loader file.

APPLIES TO REVISION(S):
0.1, 0.2

24. 16000048 - DMC CAS Latency of 2 May Not Work As Expected:

DESCRIPTION:
DMC accesses may not work as expected when the DMC is initialized with a CAS latency (CL) of 2. This anomaly applies to both DDR2 and LPDDR modes.

WORKAROUND:
Per the JEDEC specification, supporting a CAS latency of 2 is optional. For both DDR2 and LPDDR modes, use a CAS latency other than 2 to ensure proper DMC operation.

APPLIES TO REVISION(S):
0.1, 0.2

25. 16000049 - LPDDR DMC Access Failures when DLL Locks to Problematic Range of Values:

DESCRIPTION:
The DMC module's internal DLL locks to a particular DLL code, which is dependent on the DCLK frequency and is used to control various important timing relationships among numerous DMC interface signals. In LPDDR mode, the DLL might lock to a range of problematic DLL codes, which can result in DMC access failures.

WORKAROUND:
With the CGU properly programmed to establish the desired DCLK frequency, allow the DMC DLL to first lock. Once the DLL is locked, the DMC updates numerous internal configuration settings, one of which is the DLL code that must be overridden manually to work around this anomaly. Prior to performing the DMC initialization sequence, the following code is required to properly check and optionally override the latched DLL lock code:

```
#define pINTERNAL_REGISTER  (volatile int *) 0XFFC800A0

unsigned short usVal;

// CGU is programmed and DMC DLL is allowed to lock

// Manually override internal DLL lock code
usVal = *pINTERNAL_REGISTER & 0x1FF;  // Bits 8:0 are the field of interest
if(usVal >= 0x1CF)                    // Values > 0x1CF must be trimmed to 0x1CF
   usVal = 0x1CF;

*pREG_DMC0_PHY_CTL0 &= 0xFC00FFFF;                // Clear out field of interest
*pREG_DMC0_PHY_CTL0 |= ((usVal << 17)|(1 << 16)); // Program the new value

// Proceed to the DMC initialization sequence
```

Note: The `INTERNAL_REGISTER` location must only be used as described in this workaround code. Application code must otherwise never access this location.

APPLIES TO REVISION(S):
0.1, 0.2