ANALOG DEVICES

SHARC+ Dual-Core DSP with ARM Cortex-A55

Silicon Anomaly List

ADSP-SC595/SC596/SC598

ABOUT ADSP-SC595/SC596/SC598 SILICON ANOMALIES

These anomalies represent the currently known differences between revisions of the SHARC+® ADSP-SC595/SC596/SC598 product(s) and the functionality specified in the ADSP-SC595/SC596/SC598 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 TAPC0_IDCODE register can be used to differentiate the revisions as shown below.

Silicon REVISION	TAPC0_IDCODE.REVID
0.0	b#0000

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
04/25/2022	B	PrD	Added Anomalies 20000103, 20000119, 20000120, 20000121, 20000123, 20000124
08/27/2021	A	PrC	Initial Version

SHARC+ and SHARC are registered trademarks of Analog Devices, Inc.

NR004873B

Document Feedback

SUMMARY OF SILICON ANOMALIES

The following table provides a summary of ADSP-SC595/SC596/SC598 anomalies and the applicable silicon revision(s) for each anomaly.

No.	ID	Description	Rev 0.0
1	20000002	Data Forwarding from Rn/Sn to DAG Register May Fail in Presence of Stalls	x
2	20000003	Transactions on SPU and SMPU MMR Regions May Cause Errors	x
3	20000031	GP Timer Generates First Interrupt/Trigger One Edge Late in EXTCLK Mode	x
4	20000062	Writes to the SPI_SLVSEL Register Do Not Take Effect	x
5	20000069	PCSTK and MODE1STK Loads Do Not Occur If Next Instruction Is L2 or L3 Access	x
6	20000072	Floating-Point Computes Targeting F0 Register Can Cause Pipeline Stalls	x
7	20000096	Type 18a USTAT Instructions Fail When Following Specific Code Sequence	x
8	20000103	Unreliable SPDIF Receiver Clock Output Pulse Width at Sample Rates Above 96KHz	x
9	20000114	Circular Buffering in FIR Accelerator may not work properly in "burst access of length 16 words" mode	x
10	20000117	DMC PHY Calibration issue	x
11	20000118	FIR accelerator may produce wrong output for tap length greater than 1024 (multi-iteration mode) with prefetch buffer feature enabled.	x
12	20000119	END bit error may occur during eMSI data transfers due to clock gating	x
13	20000120	eMMC boot may fail for specific boot streams which cause clock gating scenario	x
14	20000121	eMMC device identification may fail during eMMC boot	x
15	20000123	Boot failure with Ignore Block in Page Mode	x
16	20000124	DMC Init routine not usable in Boot ROM	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-SC595/SC596/SC598 including a description, workaround, and identification of applicable silicon revisions.

1. 20000002 - Data Forwarding from Rn/Sn to DAG Register May Fail in Presence of Stalls:

DESCRIPTION:
An instruction involving a DAG operation such as address generation or modify following a type5a instruction may fail under the following conditions:

1. The type5a instruction updates the source register of the subsequent DAG operation.
2. The type5a instruction uses the same source register to both load to the DAG register and store the result of the compute operation.
3. The DAG operation follows within six instructions of the type5a instruction.
4. The pipeline is stalled due to a data/control dependency or an L1 memory bank conflict.

When these conditions are met, the type5a instruction produces the expected result and updates the DAG register correctly; however, the data forwarded to the DAG is incorrect, and the DAG register used as the destination in the subsequent DAG operation is incorrectly updated.

Consider the following type5a instruction sequence:

```
1: r2 = r2 - r13, i4 = r2; // r2 is destination of compute AND source of DAG load
2: if eq jump target1;     // Dependency on previous instruction stalls the pipe
3: nop;
4: nop;
5: nop;
6: nop;
7: i5 = b2w (i4);          // Uses source register (i4) stored to by type5a instruction
```

In the above case, `i5` (line 7) is updated with an incorrect value, even though `i4` (line 1) contains the correct value. The same would be true if the instruction on line 7 appeared anywhere in lines 3 through 6.

WORKAROUND:
There are two potential workarounds for this issue:

1. Split the type5a instruction which conforms to the use case into two separate instructions.
2. Avoid using the relevant DAG register in a DAG operation within six instructions of the type5a instruction.

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):
0.0

2. 20000003 - Transactions on SPU and SMPU MMR Regions May Cause Errors:

DESCRIPTION:
Non-secure reads or writes to the upper half of each SPU and SMPU instance's MMR space will be erroneously blocked and cause a bus error when configured as a non-secure slave.

For each instance of the SPU and SMPU, the affected MMR address range can be calculated as follows:

- Lower bound = Instance Address Offset + `0x800`
- Upper bound = Instance Address Offset + `0xFFF`

WORKAROUND:
Do not access the documented system MMR ranges from a non-secure slave.

APPLIES TO REVISION(S):
0.0

3. 20000031 - GP Timer Generates First Interrupt/Trigger One Edge Late in EXTCLK Mode:

DESCRIPTION:
When any GP Timer is configured in External Clock mode, the first interrupt/trigger should occur when the corresponding **TIMER_DATA_ILAT** bit sets after the **TIMER_TMRn_CNT** register reaches the value programmed in the **TIMER_TMRn_PER** register. Instead, the interrupt/trigger and the setting of the **TIMER_DATA_ILAT** bit occur one signal edge later. At this point, the **TIMER_TMRn_CNT** register will have rolled over to 1. Subsequent interrupts/triggers occur after the correct number of edges.

For example, if **TIMER_TMRn_PER**=7, the first interrupt/trigger will occur after the timer pin samples eight edges. From that point forward, interrupts/triggers will correctly occur every seven signal edges.

WORKAROUND:
For interrupts/triggers to occur every **n** edges detected on the timer pin, the **TIMER_TMRn_PER** register must be configured to **n-1** for the initial event and then reprogrammed to **n** for subsequent events, as shown in the following pseudocode:

```
TIMER_TMRn_PER = n-1;   // Configure PERIOD register with n-1
TIMER_RUN_SET = 1;      // Enable the timer
TIMER_TMRn_PER = n;     // Configure PERIOD register with n
```

The second write to the **TIMER_TMRn_PER** register does not take effect until the 2nd period; therefore, this sequence can be performed when the timer is first enabled.

APPLIES TO REVISION(S):
0.0

4. 20000062 - Writes to the SPI_SLVSEL Register Do Not Take Effect:

DESCRIPTION:
A single write to the **SPI_SLVSEL** register should change the state of the register and cause the modified software-controlled SPI slave selects to assert or de-assert. Instead, a single write to **SPI_SLVSEL** has no effect.

WORKAROUND:
Any write to **SPI_SLVSEL** should be done twice (back-to-back) with the same value in order for the change to take effect.

APPLIES TO REVISION(S):
0.0

5. 20000069 - PCSTK and MODE1STK Loads Do Not Occur If Next Instruction Is L2 or L3 Access:

DESCRIPTION:
Writes to the **PCSTK** and **MODE1STK** registers may not happen correctly if the next instruction is an access to a non-L1 memory location, as in the following code sequence:

```
1: MODE1STK = r0;
2: PCSTK    = dm(0,i6);  // i6 points to L2 or L3 memory space
3: px2      = dm(0,i6);
```

Because **i6** points to non-L1 memory in this sequence, the **MODE1STK** write on line 1 fails due to the use of **i6** on line 2, and the write to **PCSTK** on line 2 also fails because of the same use of **i6** on line 3.

WORKAROUND:
Insert a **NOP;** instruction between the write to the **PCSTK**/**MODE1STK** register and the next memory access instruction.

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):
0.0

6. 20000072 - Floating-Point Computes Targeting F0 Register Can Cause Pipeline Stalls:

DESCRIPTION:
Any floating-point compute instruction with **`F0`** as the destination register will cause pipeline stalls when followed immediately by a no-operand or single-operand compute instruction with **`Rx`** as the unused source register, as in the following code sequence:

```
F0 = PASS F4;
R10 = PASS R11; // Y operand is not used. Flushed to 0 in opcode by assembler.
```

WORKAROUND:
There are two possible workarounds:
1. Do not use the F0 register as the destination in the above code sequence.
2. Ensure that the instruction that immediately follows the compute operation is not of the form described in the code example above.

APPLIES TO REVISION(S):
0.0

7. 20000096 - Type 18a USTAT Instructions Fail When Following Specific Code Sequence:

DESCRIPTION:
Type 18a ISA/VISA register bit manipulation instructions (**`BIT SET`**, **`BIT CLR`**, **`BIT TGL`**, **`BIT TST`**, and **`BIT XOR`**) using either USTAT register can fail when immediately following an external memory (EXT_MEM) or system MMR (SMMR) read-write sequence and a read of a core memory-mapped register (CMMR) involving the same USTAT register. Consider the following pseudo-code sequence:

```
1: USTAT# = dm(EXT_MEM|SMMR);  // EXT_MEM or SMMR read to USTAT1 or USTAT2
2: dm(EXT_MEM|SMMR) = USTAT#;  // EXT_MEM or SMMR write from the same USTAT register
3: USTAT# = dm(CMMR);          // CMMR read to the same USTAT register
4: bit <op> USTAT# <data32>;   // <op> = SET|CLR|TGL|TST|XOR, using the same USTAT register
```

In this code sequence, the type 18a instruction in line 4 erroneously performs the bit operation on the value loaded to the USTAT register in instruction 1 rather than performing the operation on the expected value loaded in instruction 3.

WORKAROUND:
Insert a **`NOP;`** instruction before the type 18a instruction in the above code sequence to avoid the issue.

APPLIES TO REVISION(S):
0.0

8. 20000103 - Unreliable SPDIF Receiver Clock Output Pulse Width at Sample Rates Above 96KHz:

DESCRIPTION:
When the sampling rate of the SPDIF receiver input stream (FS_Rate) is above 96 KHz, the positive pulse width of the SPDIF receiver TDM output clock (SPDIF_RX_TDMCLK_O) can be as low as the period of the SPDIF receiver module clock, and the negative pulse width can be as high as one SPDIF receiver module clock period less than the ideal clock period. As a result, audio peripherals such as the ASRC, SPORT, and DAI pins may not function properly when SPDIF_RX_TDMCLK_O is used as the clock source.

WORKAROUND:
Do not use the SPDIF_RX_TDMCLK_O output clock as the source for external peripherals when the FS rate is above 96 KHz.

The Precision Clock Generator (PCG) can be used to divide the clock down such that audio peripherals like the ASRC, SPORT, and DAI pins may function internally; however, the clock and frame sync outputs from the PCG will still exhibit the duty cycle problem and must not be used to interface with external components.

APPLIES TO REVISION(S):
0.0

9. 20000114 - Circular Buffering in FIR Accelerator may not work properly in "burst access of length 16 words" mode:

DESCRIPTION:

FIR DMA engine must detect burst transfer which may cross circular buffer boundary (limited by B and B+L) in advance and must prevent launching a burst access. Instead it must split and launch individual accesses near the circular buffer boundary. This does not happen when "burst access of length 16" words is enabled because of failure in circular buffer boundary detection in that mode. Due to this the burst access crosses the circular buffer boundary and results in unexpected data being loaded into FIR Accelerator.

The problem is not applicable when burst access of length 16 is disabled. Note that burst 16 mode is enabled by default and the bit FIR_CTL1.BURST_16_DIS should be set to disable this feature.

WORKAROUND:

1. Align end address of the Input buffers to 4KB boundary. In this case 4kB boundary detection takes effect and FIR DMA engine splits the burst access crossing 4KB boundary into individual accesses near the circular buffer boundary.
2. Disable burst 16 feature by setting FIR_CTL1.BURST_16_DIS bit. Disabling it will not affect functional results but may result in loss of overall accelerator performance.

APPLIES TO REVISION(S):

0.0

10. 20000117 - DMC PHY Calibration issue:

DESCRIPTION:
DMC PHY calibration procedure may not work as expected sometimes with the existing DMC PHY programming model. Because of this, the driver impedance and ODT calibration may not happen correctly. This may result in DDR access failures.

WORKAROUND:
The recommended workaround code for PHY calibration requires some additional PHY register programming. The following programming sequence should be used for DMC PHY calibration to avoid the issue:

1. Program 0x00000000 to the DMC_DDR_CA_CTL register.
2. Program 0x00000000 to the DMC_DDR_ROOT_CTL register.
3. Program 0x00000000 to the DMC_DDR_SCRATCH_3 register.
4. Program 0x00000000 to the DMC_DDR_SCRATCH_2 register.
5. Program 0x04010000 to the DMC_DDR_ROOT_CTL register.
6. Wait for 2500 DCLK cycles.
7. Program 0x10000002 to the DMC_DDR_CA_CTL register.
8. Wait for 2500 DCLK cycles.
9. Program 0x00000000 to the DMC_DDR_CA_CTL register.
10. Program 0x00000000 to the DMC_DDR_ROOT_CTL register.
11. Program 0x00001000 to the DMC_DDR_SCRATCH_3 register.
12. Program 0x00000000 to the DMC_DDR_SCRATCH_2 register.
13. Wait for 2500 DCLK cycles.
14. Program 0x04010000 to the DMC_DDR_ROOT_CTL register.
15. Wait for 2500 DCLK cycles.
16. Program 0x10000002 to the DMC_DDR_CA_CTL register.
17. Wait for 2500 DCLK cycles.
18. Program 0x00000000 to the DMC_DDR_CA_CTL register.
19. Program 0x00000000 to the DMC_DDR_ROOT_CTL register.
20. Program 0x00000000 to the DMC_DDR_SCRATCH_3 register.
21. Program the DMC_DDR_SCRATCH_2.IMPWRADD bit field [7:0] with the drive strength of the address and command signals. For example, for programming a drive strength of 100 Ohm, write 0x64 into this bit field.
22. Program the DMC_DDR_SCRATCH_2.IMPWRDQ bit field [15:8] with the drive strength of the DQ, DQS, DM and clock signals. For example, for programming a drive strength of 90 Ohm, write 0x5A into this bit field.
23. Program the DMC_DDR_SCRATCH_2.IMPRTT bit field [23:16] with the adjusted On Die Termination (ODT) for the Data and DQS signals for the read operation. Due to a correction factor, program this field to 80% of the equivalent ODT.DMC_DDR_ZQ_CTL0.IMPRTT value = ODT*2*0.8 For example, if a 50 Ohm terminating resistance is required on the data pads to match the trace impedance to the board impedance, there will be two 50 Ohm resistance data pads in parallel. The value is programmed to 100 0.8 = 80.
24. Program 0x04010000 to the DMC_DDR_ROOT_CTL register.
25. Wait for 2500 DCLK cycles.
26. Program 0x0C000002 to the DMC_DDR_CA_CTL register.
27. Wait for 2500 DCLK cycles.
28. Program 0x00000000 to the DMC_DDR_CA_CTL register.
29. Program 0x00000000 to the DMC_DDR_ROOT_CTL register.
30. Program 0x00000000 to the DMC_DDR_SCRATCH_3 register.
31. Program 0x30000000 to the DMC_DDR_SCRATCH_2 register.
32. Program 0x04010000 to the DMC_DDR_ROOT_CTL register.
33. Wait for 2500 DCLK cycles.
34. Program 0x10000002 to the DMC_DDR_CA_CTL register.
35. Wait for 2500 DCLK cycles.
36. Program 0x00000000 to the DMC_DDR_CA_CTL register.
37. Program 0x00000000 to the DMC_DDR_ROOT_CTL register.
38. Program 0x00000000 to the DMC_DDR_SCRATCH_3 register.
39. Program 0x00000000 to the DMC_DDR_SCRATCH_2 register.
40. Program 0x00000000 to the DMC_DDR_SCRATCH_3 register.
41. Program 0x00000000 to the DMC_DDR_SCRATCH_2 register.
42. Program 0x04010000 to the DMC_DDR_ROOT_CTL register.
43. Wait for 2500 DCLK cycles.
44. Program 0x10000002 to the DMC_DDR_CA_CTL register.
45. Wait for 2500 DCLK cycles.
46. Program 0x00000000 to the DMC_DDR_CA_CTL register.
47. Program 0x00000000 to the DMC_DDR_ROOT_CTL register.
48. Program 0x00000000 to the DMC_DDR_SCRATCH_3 register.
49. Program 0x00000000 to the DMC_DDR_SCRATCH_2 register.
50. Program 0x00000000 to the DMC_DDR_SCRATCH_3 register.

51. Program 0x50000000 to the DMC_DDR_SCRATCH_2 register.
52. Program 0x04010000 to the DMC_DDR_ROOT_CTL register.
53. Wait for 2500 DCLK cycles.
54. Program 0x10000002 to the DMC_DDR_CA_CTL register.
55. Wait for 2500 DCLK cycles.
56. Program 0x00000000 to the DMC_DDR_CA_CTL register.
57. Program 0x00000000 to the DMC_DDR_ROOT_CTL register.
58. Program 0x0C000004 to the DMC_DDR_CA_CTL register.
59. Wait for 2500 DCLK cycles.
60. Program BITM_DMC_DDR_ROOT_CTL_TRIG_RD_XFER_ALL to the DMC_DDR_ROOT_CTL register.
61. Wait for 2500 DCLK cycles.
62. Program 0x00000000 to the DMC_DDR_CA_CTL register.
63. Program 0x00000000 to the DMC_DDR_ROOT_CTL register.
64. Calculate ODT PU and PD values as shown below and 'OR' that to pREG_DMC0_DDR_SCRATCH_2 register.

```
   stat_value = (((*pREG_DMC0_DDR_SCRATCH_7 & 0x0000FFFFu)<<16) |  ((*pREG_DMC0_DDR_SCRATCH_6 & 0xFFFF0000u)>>16));
    drv_pu = stat_value & 0x0000003Fu;
    drv_pd = (stat_value>>12) & 0x0000003Fu;
    odt_pu = (drv_pu * ClkDqsDrvImpedance)/ ROdt;
    odt_pd = (drv_pd * ClkDqsDrvImpedance)/ ROdt;
    *pREG_DMC0_DDR_SCRATCH_2 |= ((1uL<<24) | ((drv_pd & 0x0000003Fu)) | ((odt_pd & 0x0000003Fu)<<6) | ((drv_pu &
0x0000003Fu)<<12) | ((odt_pu & 0x0000003Fu)<<18));
```

Here, ClkDqsDrvImpedance is the value of the drive strength of the DQ, DQS, DM and clock signals programmed to DMC_DDR_SCRATCH_2 register in step #22.
ROdt is the value of the (ODT) for the Data and DQS signals for the read operation programmed to DMC_DDR_SCRATCH_2 register in step #23.

65. Program 0x0C010000 to the DMC_DDR_ROOT_CTL register.
66. Wait for 2500 DCLK cycles.
67. Program 0x08000002 to the DMC0_DDR_CA_CTL register.
68. Wait for 2500 DCLK cycles.
69. Program 0x00000000 to the DMC_DDR_CA_CTL register.
70. Program 0x00000000 to the DMC_DDR_ROOT_CTL register.
71. Program 0x04010000 to the DMC_DDR_ROOT_CTL register.
72. Wait for 2500 DCLK cycles.
73. Program 0x80000002 to the DMC_DDR_CA_CTL register.
74. Wait for 2500 DCLK cycles.
75. Program 0x00000000 to the DMC_DDR_CA_CTL register.
76. Program 0x00000000 to the DMC_DDR_ROOT_CTL register.

Reference C code for DMC PHY calibration:

```
#include <sys/platform.h>

/* Additional Register Address */
#define pREG_DMC0_DDR_SCRATCH_2     ((volatile uint32_t*)0x31071074)
#define pREG_DMC0_DDR_SCRATCH_3     ((volatile uint32_t*)0x31071078)
#define pREG_DMC0_DDR_SCRATCH_6     ((volatile uint32_t*)0x31071084)
#define pREG_DMC0_DDR_SCRATCH_7     ((volatile uint32_t*)0x31071088)

  uint32_t stat_value = 0x0u;
  uint32_t drv_pu , drv_pd, odt_pu, odt_pd;
  uint32_t ROdt, ClkDqsDrvImpedance, AddrDrvImpedance;

  ROdt = 120ul; /* 75 ohms of On Die Termination (ODT) for the Data and DQS signals for the
read operation. (75*2*0.8) */
  ClkDqsDrvImpedance = 90ul; /* The drive strength of the DQ, DQS, DM and clock signals */
  AddrDrvImpedance = 100ul;  /* The drive strength of the address and command signals */


  *pREG_DMC0_DDR_CA_CTL = 0x0ul;
  *pREG_DMC0_DDR_ROOT_CTL = 0x0ul;
  *pREG_DMC0_DDR_SCRATCH_3 = 0x0ul;
  *pREG_DMC0_DDR_SCRATCH_2 = 0x0ul;
```

```
    *pREG_DMC0_DDR_ROOT_CTL = 0x04010000ul;
    dmcdelay(2500u);
    *pREG_DMC0_DDR_CA_CTL = 0x10000002ul;
    dmcdelay(2500u);
    *pREG_DMC0_DDR_CA_CTL = 0x0u;
    *pREG_DMC0_DDR_ROOT_CTL = 0x0u;
    *pREG_DMC0_DDR_SCRATCH_3 = 0x1ul<<12;
    *pREG_DMC0_DDR_SCRATCH_2 = 0x0ul;
    dmcdelay(2500u);
    *pREG_DMC0_DDR_ROOT_CTL = 0x04010000ul;
    dmcdelay(2500u);
    *pREG_DMC0_DDR_CA_CTL = 0x10000002ul;
    dmcdelay(2500u);
    *pREG_DMC0_DDR_CA_CTL = 0x0ul;
    *pREG_DMC0_DDR_ROOT_CTL = 0x0ul;
    *pREG_DMC0_DDR_SCRATCH_3 = 0x0ul;
    *pREG_DMC0_DDR_SCRATCH_2 = ((ROdt & BITM_DMC_DDR_ZQ_CTL0_IMPRTT) <<
  BITP_DMC_DDR_ZQ_CTL0_IMPRTT) | ((ClkDqsDrvImpedance & BITM_DMC_DDR_ZQ_CTL0_IMPWRDQ) <<
  BITP_DMC_DDR_ZQ_CTL0_IMPWRDQ) |
  ((AddrDrvImpedance & BITM_DMC_DDR_ZQ_CTL0_IMPWRADD) << BITP_DMC_DDR_ZQ_CTL0_IMPWRADD) ;

    *pREG_DMC0_DDR_ROOT_CTL = 0x04010000ul;
    dmcdelay(2500u);
    *pREG_DMC0_DDR_CA_CTL = 0x0C000002ul;
    dmcdelay(2500u);
    *pREG_DMC0_DDR_CA_CTL = 0x0ul;
    *pREG_DMC0_DDR_ROOT_CTL = 0x0ul;
    *pREG_DMC0_DDR_SCRATCH_3 = 0x0ul;
    *pREG_DMC0_DDR_SCRATCH_2 = 0x30000000ul;
    *pREG_DMC0_DDR_ROOT_CTL = 0x04010000ul;
    dmcdelay(2500u);
    *pREG_DMC0_DDR_CA_CTL = 0x10000002ul;
    dmcdelay(2500u);
    *pREG_DMC0_DDR_CA_CTL = 0x0ul;
    *pREG_DMC0_DDR_ROOT_CTL = 0x0ul;
    *pREG_DMC0_DDR_SCRATCH_3 = 0x0ul;
    *pREG_DMC0_DDR_SCRATCH_2 = 0x0ul;
    *pREG_DMC0_DDR_SCRATCH_3 = 0x0ul;
    *pREG_DMC0_DDR_SCRATCH_2 = 0x0ul;
    *pREG_DMC0_DDR_ROOT_CTL = 0x04010000ul;
    dmcdelay(2500u);
    *pREG_DMC0_DDR_CA_CTL = 0x10000002ul;
    dmcdelay(2500u);
    *pREG_DMC0_DDR_CA_CTL = 0x0ul;
    *pREG_DMC0_DDR_ROOT_CTL = 0x0ul;
    *pREG_DMC0_DDR_SCRATCH_3 = 0x0ul;
    *pREG_DMC0_DDR_SCRATCH_2 = 0x0ul;
    *pREG_DMC0_DDR_SCRATCH_3 = 0x0ul;
    *pREG_DMC0_DDR_SCRATCH_2 = 0x50000000ul;
    *pREG_DMC0_DDR_ROOT_CTL = 0x04010000ul;
    dmcdelay(2500u);
    *pREG_DMC0_DDR_CA_CTL = 0x10000002ul;
    dmcdelay(2500u);
    *pREG_DMC0_DDR_CA_CTL = 0u;
    *pREG_DMC0_DDR_ROOT_CTL = 0u;
    *pREG_DMC0_DDR_CA_CTL = 0x0C000004u;
    dmcdelay(2500u);
    *pREG_DMC0_DDR_ROOT_CTL = BITM_DMC_DDR_ROOT_CTL_TRIG_RD_XFER_ALL;
    dmcdelay(2500u);
    *pREG_DMC0_DDR_CA_CTL = 0u;
    *pREG_DMC0_DDR_ROOT_CTL = 0u;
    /* calculate ODT PU and PD values */
    stat_value = ((*pREG_DMC0_DDR_SCRATCH_7 & 0x0000FFFFu)<<16);
    stat_value |= (*pREG_DMC0_DDR_SCRATCH_6 & 0xFFFF0000u)>>16;

    drv_pu = stat_value & 0x0000003Fu;
    drv_pd = (stat_value>>12) & 0x0000003Fu;
    odt_pu = (drv_pu * ClkDqsDrvImpedance)/ ROdt;
    odt_pd = (drv_pd * ClkDqsDrvImpedance)/ ROdt;
    *pREG_DMC0_DDR_SCRATCH_2 |= ((1uL<<24)              |
                                ((drv_pd & 0x0000003Fu))     |
                                ((odt_pd & 0x0000003Fu)<<6)  |
                                ((drv_pu & 0x0000003Fu)<<12) |
                                ((odt_pu & 0x0000003Fu)<<18));

    *pREG_DMC0_DDR_ROOT_CTL = 0x0C010000u;
    dmcdelay(2500u);
    *pREG_DMC0_DDR_CA_CTL = 0x08000002u;
    dmcdelay(2500u);
    *pREG_DMC0_DDR_CA_CTL = 0u;
    *pREG_DMC0_DDR_ROOT_CTL = 0u;
    *pREG_DMC0_DDR_ROOT_CTL = 0x04010000u;
    dmcdelay(2500u);
    *pREG_DMC0_DDR_CA_CTL = 0x80000002u;
    dmcdelay(2500u);
    *pREG_DMC0_DDR_CA_CTL = 0u;
    *pREG_DMC0_DDR_ROOT_CTL = 0u;
```

APPLIES TO REVISION(S):

0.0

11. 20000118 - FIR accelerator may produce wrong output for tap length greater than 1024 (multi-iteration mode) with prefetch buffer feature enabled.:

DESCRIPTION:

In case of multi-iteration mode (tap length >1024), the output of current iteration is computed by combining the intermediate output of the previous iteration and the MAC result of the current iteration. If starting bytes of the output buffer fall in the same prefetch line (of size 64 Bytes) as ending bytes of the input or coefficient buffer, these bytes are prefetched during coefficient or data read phase of the previous iteration. During the intermediate output write to memory, it is therefore expected that the prefetch buffer is invalidated. Due to this anomaly, prefetch buffer invalidation doesn't occur which results in stale output data being loaded in the next iteration during intermediate output read which may result in wrong combined output data.

The problem is not applicable when prefetch buffer feature is disabled or when the output buffer is not placed in same prefetch line as either the input data or coefficient buffer when in multi-iteration mode (tap length >1024).

WORKAROUND:

1. Make sure that the separation between Output Buffer and Input/Coefficient Buffer is minimum of two prefetch lines(128 bytes).
2. Disable prefetch buffer feature by clearing FIR_CTL1.PFB_EN bit. Disabling it will not affect functional results but may result in loss of overall accelerator performance.

APPLIES TO REVISION(S):

0.0

12. 20000119 - END bit error may occur during eMSI data transfers due to clock gating:

DESCRIPTION:
During eMSI data transfers if the eMSI clock is stopped in between the data transfer due to FIFO full condition (which may happen for multi-block read commands where data requested is greater than eMSI FIFO size) or due to deliberate programming of the eMSI controller to stop the eMSI clock, the END bit error will occur due to incorrect sampling of the data block END bit by the controller.

WORKAROUND:

1. Use single block read command instead of multi-block read command for SDR and DDR mode of operations.

2. eMSI clock gating in multi-block read transfers for SDR and DDR mode of operations can be avoided in two ways:
   2.1 At eMSI Controller level:
   a. For Pre-defined transfers: If DMA mode is set to ADMA, provide sufficient number of transfer descriptors and if DMA mode is set to SDMA, update the System Address Register (**EMSI_SDMA_ADDR/EMSI_ADMA_ADDR_LO**) as soon as DMA interrupt status bit (**EMSI_ISTAT_DMA_INTERRUPT**) is set.
   b. For Open-ended transfers: If DMA mode is set to ADMA, provide sufficient number of transfer descriptors. If DMA mode is set to SDMA, update the System Address Register (**EMSI_SDMA_ADDR/EMSI_ADMA_ADDR_LO**) as soon as DMA interrupt status bit (**EMSI_ISTAT_DMA_INTERRUPT**) is set and terminate the transfers using **STOP_TRANSMISSION (CMD12)** command as soon as transfer complete status bit (**EMSI_ISTAT_XFER_COMPLETE**) is set for both DMA mode.

   2.2 At System level:
   Increasing eMSI Quality of Service (QoS) priority to **12** in SCB will mitigate the END bit error in the SDR and DDR mode of data transfer operations. eMSI QoS priority can be increased as shown in the below code:

   ```
   SCB0_SDIO0_IB_WRITE_QOS = 0xC;
   ```

   Both (2.1 and 2.2) workarounds will not be applicable if eMSI clock is gated by deliberate programming.

3. In the SDR mode (not applicable to DDR mode operations), multi-block read data transfer operations, enabling the eMSI tuning logic will ensure that END bit error will not occur even if eMSI clock is gated during the data transfers. eMSI tuning logic can be enabled as in shown below code:

   ```
   PADS0_PCFG0 |= BITM_PADS_PCFG0_EMSI_TUNING_EN;
   PADS0_PCFG1 |= ENUM_PADS_PCFG1_INV4;
   ```

Refer to the 'eMSI Controller Timing - SDR Mode (Clock Tunning Logic Enabled)' table in the datasheet for timing requirement specification.

APPLIES TO REVISION(S):
0.0

13. 20000120 - eMMC boot may fail for specific boot streams which cause clock gating scenario:

DESCRIPTION:
eMMC boot may fail from user/boot area partition due to an END bit error occurring during the boot process. Some boot streams where boot data processing takes more time in boot kernel, boot ROM can gate eMSI clock to pause the reading of more data from eMMC device. This clock gating causes END bit error (anomaly 20000119) resulting in boot failure. This is applicable for both power-on reset and boot ROM API calls for eMMC boot.

WORKAROUND:

1. Program the emmcMasterBootCmd parameters in OTP for power-on reset boot:
   a. Update the emmcMasterBootCmd parameters in OTP to enable safe boot from user area partition. emmcMasterBootCmd for enabling safe boot from user area partition:

```
 emmcMasterBootCmd = (0x9u<<BITP_ROM_BCMD_SPIM_DEVICE) | ENUM_ROM_BCMD_EMSI_BUSWID_8BIT
| BITM_ROM_BCMD_EMSI_HISPEEDENABLE | ENUM_ROM_BCMD_EMSI_SDRMODE_DDR |
BITM_ROM_BCMD_EMSI_SAFEBOOT;
```

   b. Don't program emmcMasterBootCmd to enable boot from boot area partition.

2. In eMMC ROM API boot:
   a. For SDR mode (not applicable for DDR mode of boot operations) of eMMC boot operations from user/boot area partition eMSI tuning logic can be enabled before eMMC Boot ROM API call. eMSI tuning logic can be enabled as shown in below code:

```
PADS0_PCFG0 |= BITM_PADS_PCFG0_EMSI_TUNING_EN;
PADS0_PCFG1 |= ENUM_PADS_PCFG1_INV4;
```

   Refer to the 'eMSI Controller Timing - SDR Mode (Clock Tunning Logic Enabled)' table in the datasheet for timing requirement specification.
   b. For SDR/DDR mode of eMMC boot operations, boot in safe boot mode from user area partition.

APPLIES TO REVISION(S):
0.0

14. 20000121 - eMMC device identification may fail during eMMC boot:

DESCRIPTION:
eMMC device identification for some eMMC devices may fail during power-on or ROM API boot operations from user area partition, resulting in eMMC boot failure. This is due to boot ROM not able to provide 74 free-running clock cycles after the clock (of frequency less than or equal to 400 kHz) supplied to eMMC device, as recommended by JEDEC specifications.

WORKAROUND:
Most of the eMMC devices are still able to boot, as they don't need 74 free-running clock cycles after the clock (of frequency less than or equal to 400 kHz) supplied to eMMC device as recommended by JEDEC specifications. Please contact eMMC device vendors for more information.

APPLIES TO REVISION(S):
0.0

15. 20000123 - Boot failure with Ignore Block in Page Mode:

DESCRIPTION:
When page mode (ROM_BFLAG_PAGEMODE) is enabled in device boot, boot process will fail in the presence of ignore (BFLAG_IGNORE) block, when byte count from the target address of the payload, crosses the 1024 bytes page boundary due to loss of synchronization between source pointer and internal temp buffer. Also in non-secure host boot modes, when page mode is enabled, ignore block payload is processed from internal temp buffer instead of discarding it. This makes the boot rom fail to process ignore block as it is intended for. Therefore, the issue is applicable for the following cases:
1. Non Secure device and host boot modes with page mode enabled i.e. ROM_BFLAG_PAGEMODE is set in Global Flags in adi_rom_Boot().
2. Secure device boot.

WORKAROUND:
1. Do not use ignore blocks in the boot-stream for the above-mentioned cases.

APPLIES TO REVISION(S):
0.0

16. 20000124 - DMC Init routine not usable in Boot ROM:

DESCRIPTION:
DMC initialization in Boot ROM can not be performed through OTP programming or Boot ROM API call due to the existing PHY calibration issue as captured in Anomaly ID: 20000117.

WORKAROUND:
1. DMC can be initialized properly through initcode or from user's application.

APPLIES TO REVISION(S):
0.0

ANALOG DEVICES

www.analog.com