ANALOG DEVICES

SHARC Processor

Silicon Anomaly List

ADSP-21477/21478/21479

ABOUT ADSP-21477/21478/21479 SILICON ANOMALIES

These anomalies represent the currently known differences between revisions of the SHARC® ADSP-21477/21478/21479 product(s) and the functionality specified in the ADSP-21477/21478/21479 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 (see the data sheet for information on reading part branding). The silicon revision can also be electronically determined by reading the information contained in the **REVPID** and **ROMID** registers either via JTAG or DSP code.

The following DSP code can be used to read the registers:

```
UREG = dm(REVPID);
UREG = dm(ROMID);
```

Silicon REVISION	REVPID[7:4]	ROMID[3:0]
0.2	b#0000	b#0001
0.1*	b#0000	b#0000
0.0	b#0000	b#0000

* - See anomaly 15000019

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-21477	ADSP-21477W	ADSP-21478	ADSP-21478W	ADSP-21479	ADSP-21479W	Anomalies
MediaLB				x		x	15000014

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
01/26/2018	I	D	Added Anomaly: 15000033 Modified Anomalies: Improved content throughout without affecting technical details. New technical information in 15000014.
03/05/2014	H	C	Added Anomaly: 15000031
07/03/2013	G	B	Added Anomaly: 15000029
05/17/2013	F	B	Added Anomaly: 15000028 Modified Anomaly: 15000014
07/20/2012	E	B	Added Anomaly: 15000024

NR004019I **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-21477/21478/21479 anomalies and the applicable silicon revision(s) for each anomaly.

No.	ID	Description	Rev 0.0	Rev 0.1	Rev 0.2
1	15000002	Incorrect Popping of Stacks Possible When Exiting $\overline{\text{IRQx}}$/Timer Interrupts with DB Modifier	x	x	x
2	15000003	IOP Register Access Immediately Following an External Memory Access May Not Work	x	x	x
3	15000004	Effect Latency of Some System Registers May Be Two Cycles for External Data Accesses	x	x	x
4	15000005	Internal Memory Loads to Loop Registers May Fail when DMA Block Conflict Occurs	x	x	x
5	15000010	Enhanced MODIFY/BITREV Instruction Results Cannot Be Used in Next Instruction	x	x	x
6	15000012	External FLAG-Based Conditional Instructions Using DAG Register Post-Modify May Fail	x	x	x
7	15000014	MediaLB DMA-Driven Transfer Mode Requires Special PLL Initialization Sequence	x	x	x
8	15000016	PM Access Instruction Corruption when Fetching from Conflict Cache	x	x	.
9	15000018	SPORT DMA Failures when Grouped SPORTs Target Both Internal and External Memory	x	x	x
10	15000019	Incorrect Values in REVPID and ROMID Registers	.	x	.
11	15000020	Documented PLL Programming Sequence Is Insufficient for All Operating Conditions	x	x	x
12	15000021	Core Timer Remains Halted When Code Execution Resumes after Emulator Halt	x	x	x
13	15000023	VISA Mode Three-Column DM Accesses Following DAG2 Indirect Delayed Branches May Fail	x	x	x
14	15000024	Writes to Internal VISA Code Space May Cause VISA Instruction Corruption	x	x	x
15	15000028	Internal Memory Write Failure for Type-1a Instructions Including External Memory Read	x	x	x
16	15000029	Under Specification UART Stop Bit Timing for Divisor Values Greater than 2	x	x	x
17	15000031	Emulation Read and Write Data Breakpoints Are Unreliable in External Memory	x	x	x
18	15000033	Conditional External Memory Access Failure when Bit FIFO Status Flag Is Used	x	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-21477/21478/21479 including a description, workaround, and identification of applicable silicon revisions.

1. 15000002 - Incorrect Popping of Stacks Possible When Exiting $\overline{\text{IRQx}}$/Timer Interrupts with DB Modifier:

DESCRIPTION:
If a delayed branch modifier **(DB)** is used to return from the interrupt service routine of any of the $\overline{\text{IRQx}}$ (hardware) or timer interrupts, the automatic popping of the **ASTATx**, **ASTATy**, and **MODE1** registers from the status stack may not work correctly.

The specific instructions affected by this anomaly are **RTI(DB);** and **JUMP(CI)(DB);**.

This anomaly applies only to the $\overline{\text{IRQx}}$ and timer interrupts because these are the only interrupts that cause the sequencer to push these registers to the status stack, and it can be encountered while executing from both internal and external memory.

WORKAROUND:
Do not use the **(DB)** modifier in instructions exiting $\overline{\text{IRQx}}$ or timer ISRs. Instructions in the delay slot must be moved to a location prior to the branch.

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 ADI, such as VisualDSP++ and VDK, please consult the "Silicon Anomaly Tools Support" help page in the applicable documentation and release notes for details.

APPLIES TO REVISION(S):
0.0, 0.1, 0.2

2. 15000003 - IOP Register Access Immediately Following an External Memory Access May Not Work:

DESCRIPTION:
If an instruction making an access to an IOP register immediately follows another instruction that performs an access to external memory, the IOP register access may not occur correctly.

WORKAROUND:
Separate the two instructions by inserting another instruction between them, such as a **NOP;**.

APPLIES TO REVISION(S):
0.0, 0.1, 0.2

3. 15000004 - Effect Latency of Some System Registers May Be Two Cycles for External Data Accesses:

DESCRIPTION:

Certain system registers (**MODE1**, **MODE2**, **MMASK**, **SYSCTL**, **BRKCTL**, **ASTATx**, **ASTATy**, **STKYx**, and **STKYy**) incur additional effect latency when any of their bits impact an instruction containing an external data access. Effect latency is the maximum number of instructions it takes for a write to take effect to a register, and these registers normally have an effect latency of 1. For example, consider the following typical code sequence to enable bit-reversed addressing for DAG2 registers:

```
bit set MODE1 BR8; // Write to MODE1 register to enable DAG2 bit-reversing
nop;               // Accommodate effect latency of 1
pm(i8,m12)=f9;     // i8 should be accessed in bit-reversed addressing mode
```

If **i8** points to internal memory, this sequence works as expected; however, due to this anomaly, if **i8** points to external memory, the effect latency of the write to the **MODE1** register is 2 instead of 1, and the PM access will not employ bit-reversed addressing as intended.

This anomaly is independent of whether the instruction itself resides in internal or external memory; rather, the anomaly is encountered if there are external memory data accesses within the two instructions immediately following the register modification.

WORKAROUND:

Besides those listed above, no other registers are affected by this anomaly. For the identified registers, PM and DM accesses to external memory must be avoided within the two instructions immediately following the register modification. For example, applying the workaround to the above:

```
bit set MODE1 BR8; // Write to MODE1 register to enable DAG2 bit-reversing
nop;               // Accommodate effect latency of 1
nop;               // Additional NOP to work around anomaly
pm(i8,m12)=f9;     // i8 is accessed in bit-reversed addressing mode
```

This sequence will work as expected whether **i8** points to internal or external memory.

APPLIES TO REVISION(S):

0.0, 0.1, 0.2

4. 15000005 - Internal Memory Loads to Loop Registers May Fail when DMA Block Conflict Occurs:

DESCRIPTION:

If PM or DM accesses are used to load to the **LCNTR**, **CURLCNTR** or **LADDR** loop registers from either internal memory or from a memory-mapped I/O register, the write to the loop register may fail when a DMA transfer to/from the same block occurs in the same cycle. For example:

```
CURLCNTR = dm(i0,m0);     // Load loop register via DM bus read of address i0
```

If DMA accesses the same memory block as that pointed to by address **i0**, this DM access may align in such a way that the DMA transfer occurs in the same cycle, thus causing the load operation to the **CURLCNTR** register to fail.

WORKAROUND:

1. Change the DMA to source/target a different internal memory block than the one being used to load the loop register, thereby avoiding any potential DMA block conflict.
2. Rather than loading loop registers directly from memory in a single instruction, load them indirectly through a data register:

```
r0 = dm(i0,m0);
CURLCNTR = r0;
```

APPLIES TO REVISION(S):

0.0, 0.1, 0.2

5. 15000010 - Enhanced MODIFY/BITREV Instruction Results Cannot Be Used in Next Instruction:

DESCRIPTION:
When a **MODIFY** or **BITREV** instruction is followed immediately by an instruction that uses its results, the sequence may fail. Consider the following pseudo-code:

```
INSTR1: Ia = <immediate load | register load | memory load>;
INSTR2: Ia = MODIFY|BITREV (Ib, Mx);
INSTR3: <memory load | register load>  = Ia;
```

Due to this anomaly, the value in the **Ia** DAG register from **INSTR1** is what is stored in the **INSTR3** operation instead of the expected results from **INSTR2**.

This anomaly is only applicable when **Ia** and **Ib** are different.

WORKAROUND:
Do not immediately follow enhanced **MODIFY** or **BITREV** instructions with instructions that utilize their results.

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 ADI, such as VisualDSP++ and VDK, please consult the "Silicon Anomaly Tools Support" help page in the applicable documentation and release notes for details.

APPLIES TO REVISION(S):
0.0, 0.1, 0.2

6. 15000012 - External FLAG-Based Conditional Instructions Using DAG Register Post-Modify May Fail:

DESCRIPTION:
External FLAG-based conditional instructions involving DAG register post-modify operation must not be followed immediately by an instruction that uses the same index register. For example, consider the following pseudo-code:

```
INSTR1: IF COND dm(Ia,Mb); // any instruction that involves DAG post-modify operation
INSTR2: dm(Ia,Mc);         // any instruction that depends on updated Ia value
```

The value in the **Ia** DAG index register used in **INSTR2** will either be **Ia** or **Ia+Mb** after **INSTR1** executes, depending on whether the condition is met or not. When **COND** is an external **FLAG** condition (e.g., **FLAG2_IN**, which is set asynchronously by an external source or event), the internal stalls required to allow for proper execution of this sequence are not inserted. As a result, the value of the **Ia** DAG index register used in **INSTR2** may not be correct, which leads to incorrect application code execution.

WORKAROUND:
Separate the instructions in the above sequence by at least two **NOP;** instructions to manually insert the erroneously omitted stalls.

APPLIES TO REVISION(S):
0.0, 0.1, 0.2

7. 15000014 - MediaLB DMA-Driven Transfer Mode Requires Special PLL Initialization Sequence:

DESCRIPTION:
The MediaLB interface's DMA clock may lose synchronization with the processor's internal clock if the procesor's PLL is placed in bypass mode as part of the initialization sequence.

WORKAROUND:
This anomaly does not apply when using the MediaLB interface in core-driven mode. When using it in DMA-driven transfer mode, the MediaLB clock must be disabled before PLL reprogramming and subsequently re-enabled after PLL programming is complete, per the following programming model:

1. Disable the clock to the MediaLB interface.
2. Place the PLL in bypass mode.
3. Program the PLL parameters and provide the required delays for the changes to take effect, per the programming model.
4. Bring the PLL out of bypass mode.
5. Re-enable the clock to the MediaLB interface.

This sequence must be implemented in assembly language and executed from internal memory with interrupts disabled and no active DMA ongoing. When programming the PLL parameters, the assembly code sequence defined in the anomaly 15000020 workaround **must** be followed and modified to integrate the above requirements regarding the MediaLB clock. Specifically:

1. Prior to making PLL adjustments, the MediaLB clock must be disabled. As such, steps 1 and 7 of the 15000020 workaround code must insert this code immediately before the code shown in the workaround:

```
/* Insert at Top of Steps 1 and 7 of 15000020 Workaround */
ustat4 = dm(PMCTL1);
bit set ustat4 MLBOFF;  // Set bit to disable MediaLB clock
R2=dm(MLB_VCCR);        // read off-core IOP register to sync core/peripheral clocks
dm(PMCTL1) = ustat4;    // Disable MediaLB clock

// Documented step 1 or step 7 code from 15000020 workaround goes here
```

2. Exiting PLL bypass mode requires replacement of steps 5 and 9 in the 15000020 sequence to re-enable the MediaLB clock after the PLL adjustments are made. As these two steps are the same instruction sequence, both steps 5 and 9 are replaced by this code:

```
/* Replaces Steps 5 and 9 of the 15000020 Workaround Code */
ustat1 = dm(PMCTL);
bit clr ustat1 PLLBP;   // Clear the bypass bit to exit PLL bypass mode
ustat4 = dm(PMCTL1);
bit clr ustat4 MLBOFF;  // Clear the bit to enable the MLB clock
dm(PMCTL) = ustat1;     // Exit PLL bypass mode
dm(PMCTL1) = ustat4;    // Enable MLB clock
```

APPLIES TO REVISION(S):
0.0, 0.1, 0.2

8. 15000016 - PM Access Instruction Corruption when Fetching from Conflict Cache:

DESCRIPTION:

A PM access instruction itself or the instructions immediately following it may get corrupted when the PM access:

1. conflicts with a core/DMA access to the same memory block,
2. accesses any memory-mapped (IOP) register, or
3. accesses external memory space.

In both VISA and non-VISA modes, if the PM access instruction is in a counter-based or non-counter-based single-instruction loop, the instruction itself may get corrupted while being fetched from the conflict cache. For counter-based loops, the corruption can only occur when the loop count value is greater than four. For example:

```
lcntr=x, do (pc,1) until lce;    // x > 4
dm(I0,M0)=R10, pm(I12,M10)=R10;  // I12 is IOP or external memory or core/DMA conflict occurs
```

Unique to VISA mode, if the second of two consecutive PM access instructions is compressed, the instructions following it may get corrupted while being fetched from the conflict cache.

```
pm(I12,M10)= <imm_data>;         // PM access 1, <imm_data> - can be 16-bit/32-bit
                                 // immediate value compressed or uncompressed
dm(I0,M0)=R10, pm(I12,M10)=R10;  // PM access 2 compressed
...                              // instructions may get corrupted
...
```

This anomaly is **not** encountered when the consecutive PM access instructions are the result of a loop wrap condition (i.e., the first and last instructions of a hardware loop contain PM access instructions).

WORKAROUND:

For all manifestations of this anomaly, DM accesses can be used instead of PM accesses. If PM accesses are required, avoid memory block conflict stalls by moving either the core/DMA access or the PM access to a different memory block and disable the cache prior to any PM access instruction to IOP or external memory space. For example:

```
BIT SET MODE2 CADIS;
nop; nop;
lcntr=x, do (pc,1) until lce;    // x > 4
dm(I0,M0)=R10, pm(I12,M10)=R10;  // I12 is IOP or external memory or core/DMA conflict occurs
BIT CLR MODE2 CADIS;
nop; nop;
```

As disabling cache is not recommended and likely not preferred in the application, alternate workarounds to disabling the cache are:

1. For both VISA and non-VISA mode code, do not use an instruction containing a PM access as a single-instruction loop.
2. For VISA mode code, either:
   a. avoid consecutive PM access instructions altogether by inserting any non-PM access instruction between them, OR
   b. protect any length sequence of consecutive PM access instructions by utilizing the **.NOCOMPRESS** assembler directive to disable instruction compression from the second PM access instruction to the end of the sequence:

```
pm(I12,M10)= <imm_data>;         // PM access 1, <imm_data> - can be 16-/32-bit
                                 // immediate value
.NOCOMPRESS;                     // Disable compression
dm(I0,M0)=R10, pm(I12,M10)=R10;  // PM access 2
...                              // Any number of consecutive PM access instructions
.COMPRESS;                       // Re-enable compression
```

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 ADI, such as VisualDSP++ and VDK, please consult the "Silicon Anomaly Tools Support" help page in the applicable documentation and release notes for details.

APPLIES TO REVISION(S):

0.0, 0.1

9. 15000018 - SPORT DMA Failures when Grouped SPORTs Target Both Internal and External Memory:

DESCRIPTION:
The SPORT DMA channels are grouped as follows:

Group	DMA Channels Associated With
1	SPORT0 (SP0A, SP0B) and SPORT1 (SP1A and SP1B)
2	SPORT2 (SP2A, SP2B) and SPORT3 (SP3A and SP3B)
3	SPORT4 (SP4A, SP4B) and SPORT5 (SP5A and SP5B)
4	SPORT6 (SP6A, SP6B) and SPORT7 (SP7A and SP7B)

When SPORTs within a single group are enabled in DMA mode with both read and write transactions accessing both internal and external memory **and** any other SPORT DMA is also active, the SPORT DMA operations may fail. For transmit DMA, the data output may be incorrect. For receive DMA, the data written to the memory will be correct, but additional latency is incurred between two successive writes. If transmit and/or receive DMA chaining is utilized, the associated TCBs may also not load correctly to the registers, thus leading to unpredictable DMA behavior and/or errors.

WORKAROUND:
Resolve all data buffers and any associated TCBs (if chaining is used) for SPORT DMA channels of the same group to **either** internal **or** external memory, not to a mix of both. For example, if SPORTs 0 and 1 use chaining and SPORTs 4 and 5 do not, but all 4 are enabled for bidirectional operation, the following use of memory is an example of what is required:

Memory Space	Resolved Content
Internal Memory	SP0A/SP0B/SP1A/SP1B TCBs and transmit/receive data buffers
External Memory	SP4A/SP4B/SP5A/SP5B transmit/receive data buffers

APPLIES TO REVISION(S):
0.0, 0.1, 0.2

10. 15000019 - Incorrect Values in REVPID and ROMID Registers:

DESCRIPTION:
The **`REVPID[7:4]`** and **`ROMID[4:0]`** register bit fields were not updated in this revision of silicon and erroneously contain the same values as those of the previous revision 0.0 silicon.

WORKAROUND:
None.

APPLIES TO REVISION(S):
0.1

11. 15000020 - Documented PLL Programming Sequence Is Insufficient for All Operating Conditions:

DESCRIPTION:

The documented PLL programming sequence is not robust in terms of when the writes to the PMCTL register occur relative to various combinations of internal clock alignment across all application conditions, which can result in two failure scenarios:

1. A newly programmed PLLM multiplier value may not take effect, even when the PLL properly enters the bypass mode during the recommended programming sequence. When this occurs, fVCO is unchanged and will remain that which is associated with the previous PLLM value. Consequently, any clocks derived from fVCO are similarly affected, whether the divisor values were changed as a part of the sequence or not (i.e., any new divisor will properly update, but the associated output clock frequency is still not as expected because fVCO didn't change as a result of the PLLM update not occurring).
2. PLL bypass mode may not be properly transitioned to during the sequence. When this occurs, both the newly written PLLM clock multiplier *and* divisors can possibly not take effect, resulting in the core and peripheral clocks being other than those expected. Furthermore, core hangs can also be encountered when the application later attempts any external port access after this anomaly occurs. Additionally, any newly written PLL parameters that did not take effect immediately due to encountering the anomaly during a previous PLL programming attempt may unexpectedly take effect during a subsequent PLL programming sequence that does properly enter the PLL bypass mode.

Due to the numerous operating condition and timing dependencies, it is very difficult to reproduce failures across devices, and failures will be extremely intermittent in nature on devices proven to be sensitive to this issue.

WORKAROUND:

When programming the PLL to change the **PMCTL.PLLM** and/or **PMCTL.INDIV** settings, the following sequence must be used to avoid the internal clock alignment required for this anomaly to manifest:

1. Set the **PMCTL.DIVEN** bit, but do not yet change the **PMCTL.PLLM** value. If a change to **PMCTL.SDCKR** is also needed **and** the final values of both **PMCTL.PLLD** and **PMCTL.SDCKR** are 4, set **PMCTL.PLLD** to 8; otherwise set **PMCTL.PLLD** to 4.
2. Wait at least 16 CCLK cycles.
3. Clear the **PMCTL.DIVEN** bit while setting both the **PMCTL.INDIV** and **PMCTL.PLLBP** bits. The PLL now goes to bypass mode for the first time.
4. Wait at least 4096 CCLK cycles.
5. Clear the **PMCTL.PLLBP** bit to take the PLL out of bypass mode.
6. Wait at least 16 CCLK cycles.
7. Initiate the documented PLL programming sequence of setting the new **PMCTL.PLLM** and **PMCTL.INDIV** values while clearing the **PMCTL.DIVEN** bit. In this write, also set the **PMCTL.PLLBP** bit to again put the PLL into bypass mode.
8. Wait at least 4096 CCLK cycles.
9. Clear the **PMCTL.PLLBP** bit to take the PLL out of bypass mode.
10. Wait at least 16 CCLK cycles.
11. If the required settings in step 1 are not the final settings for the application, set the new **PMCTL.PLLD** and **PMCTL.SDCKR** values while also setting the **PMCTL.DIVEN** bit.
12. Wait at least 16 CCLK cycles.

The assembly code sequence for the above programming model is:

```
#include <def21479.h>          // ADSP-2147x header file

// #define PLLD4_SDCKR4        // Uncomment if final PLLD = SDCKR = 4

// Step 1
ustat1 = dm(PMCTL);
bit clr USTAT1 PLLD16 ;        // Clear PLLD field
#ifdef PLLD4_SDCKR4            // If final PLLD = SDCKR = 4...
bit set USTAT1 PLLD8|DIVEN;    // Set PLLD = 8 and enable output divider
#else                          // Otherwise...
bit set USTAT1 PLLD4|DIVEN ;   // Set PLLD = 4 and enable output divider
#endif
dm(PMCTL) = ustat1;

// Step 2 (Wait at least 16 CCLK cycles)
lcntr=16, do first_div_delay until lce;
first_div_delay: nop;

// Step 3
bit clr ustat1 DIVEN;          // Disable output divider
bit set ustat1 INDIV|PLLBP;    // Enable input divider and go to bypass
dm(PMCTL) = ustat1;

// Step 4 (Wait at least 4096 CCLK cycles)
lcntr=4096, do first_bypass_delay until lce;
```

```
first_bypass_delay:nop;

// Step 5
ustat1 = dm(PMCTL);
bit clr ustat1 PLLBP;          // Take PLL out of bypass
dm(PMCTL) = ustat1;

// Step 6 (Wait at least 16 CCLK cycles)
lcntr=16, do second_div_delay until lce;
second_div_delay:nop;

// Step 7
ustat1 = dm(PMCTL);
bit clr ustat1 INDIV|PLLM63;   // Clear PLLM field and disable input divider
bit set ustat1 PLLM18|PLLBP;   // Set new PLLM and input divider values and go to bypass
dm(PMCTL) = ustat1;            // (PLLM = 18 and INDIV = 0 in this example)

// Step 8 (Wait at least 4096 CCLK cycles)
lcntr=4096, do second_bypass_delay until lce;
second_bypass_delay:nop;

// Step 9
ustat1 = dm(PMCTL);
bit clr ustat1 PLLBP;          // Take PLL out of bypass
dm(PMCTL) = ustat1;

// Step 10 (Wait at least 16 CCLK cycles)
lcntr=16, do third_div_delay until lce;
third_div_delay:nop;

// Step 11 (0x1C0000 is the mask for the SDCKR field)
ustat1 = dm(PMCTL);
bit clr ustat1 PLLD16|0x1C0000;  // Clear PLLD and SDCKR fields
bit set ustat1 SDCKR2|PLLD2|DIVEN;  // Set new PLLD and SDCKR values
dm(PMCTL) = ustat1;

// Step 12 (Wait at least 16 CCLK cycles)
lcntr=16, do fourth_div_delay until lce;
fourth_div_delay: nop;
```

The equivalent C Code sequence is:

```
#include <def21479.h>             // ADSP-2147x header files
#include <cdef21479.h>

// #define PLLD4_SDCKR4           // Uncomment if final PLLD = SDCKR = 4

int temp, i;

// Step 1
temp = *pPMCTL & ~PLLD16;         // Clear PLLD field
#ifdef PLLD4_SDCKR4               // If final PLLD = SDCKR = 4...
    temp |= (PLLD8|DIVEN);        // Set PLLD = 8 and enable output divider
#else                             // Otherwise...
    temp |= (PLLD4|DIVEN);        // Set PLLD = 4 and enable output divider
#endif
    *pPMCTL = temp;

// Step 2
for(i=0; i<16; i++);              // Wait at least 16 CCLK cycles

// Step 3
temp &= ~DIVEN;                   // Disable output divider
temp |= (INDIV|PLLBP);            // Enable input divider and go to bypass
*pPMCTL = temp;

// Step 4
```

```
for(i=0; i<4096; i++);            // Wait at least 4096 CCLK cycles

// Step 5
temp = *pPMCTL & ~PLLBP;          // Exit bypass
*pPMCTL = temp;

// Step 6
for(i=0; i<16; i++);              // Wait at least 16 CCLK cycles

// Step 7
temp = *pPMCTL & ~(INDIV|PLLM63); // Disable input divider and clear PLLM field
temp |= (PLLM18|PLLBP);           // Set input divider and PLLM and go to bypass
*pPMCTL = temp;                   // (PLLM = 18, DIVEN = 0 in this example)

// Step 8
for(i=0; i<4096; i++);            // Wait at least 4096 CCLK cycles

// Step 9
temp = *pPMCTL & ~PLLBP;          // Exit bypass
*pPMCTL = temp;

// Step 10
for(i=0; i<16; i++);              // Wait at least 16 CCLK cycles

// Step 11 (0x1C0000 is the mask for the SDCKR field)
temp = *pPMCTL;
temp &= ~(PLLD16|0x1C0000);         // Clear PLLD and SDCKR fields
temp |= (SDCKR2|PLLD2|DIVEN); // Set new PLLD and SDCKR values
*pPMCTL = temp;

// Step 12
for(i=0; i<16; i++);              // Wait at least 16 CCLK cycles
```

One special use case involves changing only the **PMCTL.SDCKR** ratio specifically to 4 (with no changes to the **PMCTL.PLLM** and **PMCTL.INDIV** values). If a subsequent PLL reprogramming includes a change to **PMCTL.PLLM** and/or **PMCTL.INDIV** (as above), then this programming model must be followed:

1. Set **PMCTL.PLLD** to something other than the original value while also setting the **PMCTL.DIVEN** bit.
2. Wait at least 16 CCLK cycles.
3. Set **PMCTL.PLLD** back to the original value while also setting **PMCTL.SDCKR** to 4 and setting the **PMCTL.DIVEN** bit.
4. Wait at least 16 CCLK cycles.

The assembly code sequence for this programming model is:

```
#include <def21479.h>               // ADSP-2147x  header files

// Step 1
ustat1 = dm(PMCTL);                 // PLLD reads as 4 in this example
bit clr USTAT1 PLLD16;              // Clear PLLD field. Whatever the value was, set PLLD
bit set USTAT1 PLLD8|DIVEN;         // to something else and enable output divider
dm(PMCTL) = ustat1;

// Step 2 (Wait at least 16 CCLK cycles)
lcntr=16, do div_delay1 until lce;
div_delay1: nop;

// Step 3
ustat1 = dm(PMCTL);
bit clr USTAT1 PLLD16|0x1C0000;  // Clear PLLD and SDCKR (mask = 0x1C0000) fields
bit set USTAT1 PLLD4|DIVEN|SDCKR4;  // Reset PLLD, set SDCKR, and enable output divider
dm(PMCTL) = ustat1;

// Step 4 (Wait at least 16 CCLK cycles)
lcntr=16, do div_delay2 until lce;
div_delay2: nop;
```

The equivalent C code sequence is:

```
#include <def21479.h>            // ADSP-2147x header file
```

```
#include <cdef21479.h>

#pragma optimize_off            // Disabling optimization is REQUIRED

int temp, i;

// Step 1
temp = *pPMCTL & ~PLLD16;       // PLLD reads as 4. Clear it while also setting
temp |= (PLLD8|DIVEN);          // PLLD = 8 and enabling output divider
*pPMCTL = temp;

// Step 2 (Wait at least 16 CCLK cycles)
for(i=0; i<16; i++);

// Step 3
temp = *pPMCTL & ~(PLLD16|0x1C0000);  // Clear PLLD and SDCKR (mask = 0x1C0000) fields
temp |= (SDCKR4|PLLD4|DIVEN);  // Reset PLLD to 4, set SDCKR, and enable output divider

*pPMCTL = temp;

// Step 4 (Wait at least 16 CCLK cycles
for(i=0; i<16; i++);

#pragma optimize_as_cmd_line    // Re-enable optimization
```

All of the above code sequences are interruptible and are valid for both VISA and non-VISA modes.

Note that there is **no** workaround to the case where the PLL is programmed with a new PLLM value when fVCO > fVCO(max)/2 and INDIV=1. Since INDIV=0 at reset, this restriction applies only when the PLL is reprogrammed more than once, in which case fVCO must be less than fVCO(max)/2 during all PLL programming attempts other than the last one if INDIV is set. For fVCO >= fVCO(max)/2, use INDIV=0 and halve the PLLM value that would be used when INDIV=1. This may, however, result in fCCLK precision loss.

Note also that fVCO may be dropped to below the fVCO(min) specification during execution of this workaround. This is allowed **only** during the execution of the workaround code and provided fVCO >= 150 MHz. When the workaround code completes, fVCO must be returned to within specification before executing further application code.

APPLIES TO REVISION(S):
0.0, 0.1, 0.2

12. 15000021 - Core Timer Remains Halted When Code Execution Resumes after Emulator Halt:

DESCRIPTION:
When the processor is halted at a breakpoint in an emulator session, the core timer correctly stops decrementing and restarts when code execution is resumed. However, if the emulator breakpoint is placed on the instruction immediately before an idle instruction, the core timer remains halted even after the code execution is resumed. For example:

```
Bit set MODE1 TIMEN;         // Timer enabled
...
...
Instruction1;                // Breakpoint on this instruction
idle;
```

After the core halts at **`Instruction1`** due to the breakpoint in a debug session, the core timer will not resume whether the program is run from the breakpoint or single-stepped to the **`idle;`** instruction.

WORKAROUND:
None

APPLIES TO REVISION(S):
0.0, 0.1, 0.2

13. 15000023 - VISA Mode Three-Column DM Accesses Following DAG2 Indirect Delayed Branches May Fail:

DESCRIPTION:

When in VISA mode, three-column accesses over the DM bus may not work as expected when immediately following an indirect delayed branch instruction that uses DAG2 registers. For example:

Scenario 1: 40-bit data access using RF registers - the following sequence may encounter the failure if the corresponding **SYSCTL.IMDWx** bit is set to enable 40-bit data access:

```
jump(m8,i8) (db);     // Indirect delayed branch instruction using DAG2 registers
r0=dm(i1,m1);         // Read access may fail
nop;
```

Scenario 2: 48-bit data access using PX register - the following sequence may encounter the failure regardless of whether the corresponding **SYSCTL.IMDWx** bit is set or not:

```
call(m8,i8) (db);     // Indirect delayed branch instruction using DAG2 registers
dm(i1,m1)=px;         // Write access may fail
nop;
```

Scenario 3: 40-bit data access using `rframe;` instruction - because the **`rframe;`** instruction is comprised of the sequence **`i7=i6, i6=dm(0,i6)`**, it may encounter the failure if the corresponding **SYSCTL.IMDWx** bit is set.

```
jump(m8,i8) (db);     // Indirect delayed branch instruction using DAG2 registers
rframe;               // i6=dm(0,i6) read access within rframe may fail
nop;
```

The above applies to both direct and indirect reads and writes, and the occurrence of the failure depends upon the relationship between the target address of the indirect branch (e.g., **`i8+m8`** in the above examples) and the address of the three-column data access.

This anomaly does **not** apply to IOP accesses, external memory data accesses, nor PC-relative branch instructions.

WORKAROUND:

1. Do not perform 3-column DM accesses immediately after indirect delayed branch instructions that use DAG2 registers. If possible, move the access to the second slot of the delayed branch. For example, putting the **`rframe;`** instruction in the second slot of the delayed branch in the scenario 3 sequence avoids the failure:

   ```
   jump(m8,i8) (db);     // Indirect delayed branch instruction using DAG2 registers
   nop;
   rframe;               // 40-bit read access "i6=dm(0,i6)"  moved to second slot
   ```

2. Use a direct branch instruction instead of an indirect branch instruction.
3. Use a normal indirect branch instruction instead of a delayed indirect branch instruction; however, this may result in additional branch latency, and this workaround may not be helpful if the delayed branch was intended to atomically execute the two instructions in the delayed branch slots.
4. Use a PM access instead of a DM access. For example, replacing the DM access by a PM access (**`i1`** replaced by **`i9`**) in the scenario 1 sequence avoids the failure:

   ```
   jump(m8,i8) (db);
   r0=pm(i9,m9);       // DM replaced by PM, i1 (DAG1) replaced by i9 (DAG2) for PM access
   nop;
   ```

   However, this workaround may result in an additional core clock cycle due to a PM bus conflict with the instruction fetch.
5. Do not use VISA mode for such data accesses.

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 ADI, such as VisualDSP++ and VDK, please consult the "Silicon Anomaly Tools Support" help page in the applicable documentation and release notes for details.

APPLIES TO REVISION(S):

0.0, 0.1, 0.2

14. 15000024 - Writes to Internal VISA Code Space May Cause VISA Instruction Corruption:

DESCRIPTION:

When any internal DMA or core write occurs to a location either containing or within two 16-bit words following a VISA instruction, subsequent execution of that VISA instruction sequence may fail. If the write was one of the two most recent writes to that block of internal memory, the sequencer's VISA instruction fetch mechanism erroneously does not check the internal write FIFO for uncompleted writes and will fetch the not updated data from the source memory, which can result in core hangs or otherwise unexpected application execution.

For example, the **`r1=0xABCDEF12;`** instruction (with an opcode of 0x0F01ABCDEF12) is mapped to a short word location starting at address 0x124314. If a short word write is performed to address 0x124315 such that 0xABCD is changed to 0x0000, the instruction aligned to location 0x124314 should be changed to opcode 0x0F010000EF12 (i.e., **`r1=0xEF12;`**), as follows:

```
0x124310: r0=0;            // Code in block 0
0x124312: dm(0x124315)=r0; // This write should modify the next instruction
0x124314: r1=0xABCDEF12;   // Affected instruction
```

Because of the anomaly, the write instruction at address 0x124312 does not propagate to memory, and the program sequencer fetches the now stale **`r1=0xABCDEF12;`** instruction at address 0x124314 instead of the updated **`r1=0xEF12;`** instruction; consequently, **`R1`** will contain 0xABCDEF12 instead of the expected 0xEF12 after this sequence executes.

This anomaly can also manifest when the internal memory is initialized with VISA instructions for the first time during the boot process.

WORKAROUND:

If such a write occurs, it must be followed by either:

1. at least 2 writes to a location in the same memory block that is not a VISA instruction nor the two 16-bit locations following one:

```
r0=0;                  // Code in block 0
dm(0x124315)=r0;       // Offending write instruction
dm(0x124300)=r0;       // Dummy write to safe (non-VISA) addresses 0x124300-0x124303
dm(0x124300)=r0;       // 2nd dummy write to the same or another "safe" location
r1=0xABCDEF12;         // Affected instruction at address 0x124314 will be updated
```

2. at least 2 cycles with no core, DMA, nor program sequencer accesses to that memory block. In addition to disabling all DMA channels that access the same block, the above code must be modified to:

```
.section/sw seg_code_block0;
r0=0;
dm(0x124315)=r0;       // Offending write instruction to block 0
call Dummy_block1_code; // Call dummy code in block 1
r1=0xABCDEF12;         // Affected instruction at address 0x124314 will be updated

.section/sw seg_code_block1;
Dummy_block1_code:     // Sequencer fetches/executes from block 1
nop; nop;              // At least 2 instructions that do not access block 0
rts;                   // Return to application code
```

The standard interrupt setup functions in the VisualDSP and CrossCore Embedded Studio C/C++ libraries are self-modifying and must not be used in VISA space. Starting with VisualDSP++ 5.0 Update 9 (including all revisions of CrossCore Embedded Studio), use the non-self-modifying **`interruptnsm()`**, **`interruptcbnsm()`**, **`interruptfnsm()`**, **`interruptsnsm()`**, and **`interruptssnsm()`** versions instead. For older tools versions, add the *{VisualDSP Install Path}\214xx\lib\src\libc_src* include path to the assembler options, then copy the *{VisualDSP Install Path}\214xx\lib\src\libc_src\irptl.asm* source file to the project and modify it as follows:

```
.GLOBAL modify_instr, modify_two_instrs;
nop;                   // Insert NOP here
modify_two_instrs:
BIT SET IRPTL 0x00000000;
```

Specific to initialization, the default processor boot kernel initializes the non-VISA interrupt vector table (IVT) at the end of the loading process, so subsequent execution of application code in any block is not exposed to this anomaly.

APPLIES TO REVISION(S):

0.0, 0.1, 0.2

15. 15000028 - Internal Memory Write Failure for Type-1a Instructions Including External Memory Read:

DESCRIPTION:

If a conditional branch instruction immediately follows a Type-1a (compute + dual data memory move) instruction comprised of a multiply compute, an internal memory write, and an external memory access, the internal memory write portion of the Type-1a instruction may fail if the conditional branch depends on the multiplication result, whether or not the condition itself is met. For the anomaly to manifest, the source register for the internal memory write must be the same as the destination register used by **either** the multiply or the external memory access. When the anomaly occurs, the destination register is updated *before* the internal memory write executes, thereby writing the result of the read or multiply operation to internal memory instead of the data that was in the source register before the Type-1a instruction began executing.

The anomaly applies only to the following four versions of the Type-1a instruction when followed immediately by a conditional branch that depends on the multiply result:

1. Rx = Multiply Compute, Internal Memory Write = Rx, External Memory Access;

   ```
   R8 = R1 * R2 (UUI), dm(Ix, Mx) = R8, R1 = pm(Iy, My);
   ```

   If **Ix** points to internal normal- or short-word address space and **Iy** points to external memory, the internal memory will be written with the result of the multiplication.

2. Multiply Compute, Rx = External Memory Read, Internal Memory Write = Rx;

   ```
   R5 = R2 * R7 (UUI), R4 = dm(Iy, My), pm(Ix, Mx) = R4;
   ```

   If **Ix** points to internal normal- or short-word address space and **Iy** points to external memory, the internal memory will be written with the data read from external memory.

3. Multiply Compute, Rx = ALU Operation, External Memory Access, Internal Memory Write = Rx;

   ```
   R5 = R2 * R7 (SSFR), R1 = R11 + R15, dm(Iy,My) = R4, pm(Ix,Mx) = R1;
   ```

   If **Ix** points to internal normal- or short-word address space and **Iy** points to external memory, the internal memory will be written with the result of the ALU operation.

4. Rx = Multiply Compute, External Memory Access, Internal Long-Word Memory Write = Ry;

   ```
   R1 = R3 * R7 (SSFR), R2 = pm(Ix,My), dm(Iz,Mz) = R0;
   ```

   If **Ix** points to external memory and **Iz** points to internal long-word address space, the **R0:R1** register pair is modified by the multiplication, and the internal memory will be written with the portion of the multiply result that is stored to **R1**.

The branch instruction that must immediately follow the above Type-1a instruction can be any type (**jump**, **call**, **rts**, or **rti**) and must be conditional on multiplier status bits. For example:

```
R8=R1*R2(UUI), dm(Ix,Mx)=R8, R1=pm(Iy,My);  // Type-1a format #1 above
If not ms call label2;                      // Branch depends on multiplier status bit
```

WORKAROUND:

1. Avoid this sequence by placing a **NOP;** instruction between the Type-1a instruction and the conditional branch instruction.
2. Do not target both internal and external memory in the Type-1a instruction in the above sequence.

APPLIES TO REVISION(S):

0.0, 0.1, 0.2

16. 15000029 - Under Specification UART Stop Bit Timing for Divisor Values Greater than 2:

DESCRIPTION:

When the UART clock divisor established by the combination of the **UART0DLL** and **UART0DLH** registers is 3 or greater, the stop bit width is 14 PCLK cycles shorter than the expected full stop bit width. This applies to both one and two stop bit modes, as governed by the **UART0LCR.UARTSTB** configuration bit.

This phenomenon becomes especially problematic at very low divisor values with one stop bit, where the 14 PCLK difference results in a violation of the EIA-404 standard specification requiring that the UART stop bit width be at least 80% of the expected width. Specifically, this anomaly causes the actual stop bit width for divisors of 3 (70.8%) and 4 (78.125%) to violate this specification.

WORKAROUND:

To prevent the 14 PCLK shortening of the width of the stop bit, use only UART clock divisor values of 1 or 2.

When using UART clock divisor values of 3 or 4 with one stop bit, instead configure the UART for two stop bits (**UART0LCR.UARTSTB** = 1) to ensure compliance to the EIA-404 stop bit width specification.

When using UART clock divisor values of 5 or greater with one stop bit, consider the 14 PCLK shortening of the stop bit width when configuring the clock divisor such that the actual stop bit width is as near to 100% of the expected width as possible.

APPLIES TO REVISION(S):

0.0, 0.1, 0.2

17. 15000031 - Emulation Read and Write Data Breakpoints Are Unreliable in External Memory:

DESCRIPTION:

For both "read access" and "write access" triggering modes, the emulation PM and DM data access breakpoints do not work properly for address ranges associated with external memory. A write breakpoint associated with an external memory address range will **only** work as expected when the instruction immediately following the matching write access *also* contains a PM/DM write operation to any address. If the instruction immediately following the matching write access does not contain a PM/DM write operation, not only is the write breakpoint missed, but a false read breakpoint will be hit if the written address that should have hit the write breakpoint is also within the defined read breakpoint address range. Finally, a read breakpoint associated with an external memory address range will work in all cases **except** when the instruction immediately following the matching read access contains a PM/DM write operation to any address.

WORKAROUND:

For emulation data access breakpoints associated with external memory, do not rely on the "read access" or "write access" triggering modes. Configure the **BRKCTL.DxMODE** and **EMUCTL.DxMODE** bit fields for "any access" (0b11) to enable a functional breakpoint for all read and write accesses to the external memory address range of interest.

APPLIES TO REVISION(S):

0.0, 0.1, 0.2

18. 15000033 - Conditional External Memory Access Failure when Bit FIFO Status Flag Is Used:

DESCRIPTION:

The bit FIFO status flag (**ASTATx.SF** and **ASTATy.SF**) is typically available for use in conditional instructions in the cycle immediately following any instruction that updates it. If, however, the immediately following conditional instruction contains an external memory access, the update does not occur properly. For example:

```
r1 = 0xadadadad;
pm(I8, M8) = r1;                        // I8 points to external memory address
bffwrp = 32;                            // Sets SF bit
if sf r2 = BITEXT 1, r0 = pm(I8, M8);   // Conditional use of SF flag
```

Due to this anomaly, there is an unexpected 1-cycle effect latency for the update to the **SF** flag, which results in the **`if sf`** condition being evaluated as not met (despite the **SF** flag being set by execution of the immediately preceding **`bffwrp`** instruction). As such, the conditional instruction that should have executed is instead skipped.

WORKAROUND:

As the effect latency for the update to the **SF** flag is one cycle, it is sufficient to separate SF-conditional external memory accesses from instructions that may manipulate the bit FIFO status flag by one instruction:

```
r1 = 0xadadadad;
pm(I8, M8) = r1;                        // I8 points to external memory address
bffwrp = 32;                            // Sets SF bit
nop;                                    // Any instruction that does not affect SF
if sf r2 = BITEXT 1, r0 = pm(I8, M8);   // Conditional use of SF flag
```

APPLIES TO REVISION(S):

0.0, 0.1, 0.2