ANALOG DEVICES

SHARC Processor

Silicon Anomaly List

ADSP-21363

ABOUT ADSP-21363 SILICON ANOMALIES

These anomalies represent the currently known differences between revisions of the SHARC® ADSP-21363 product(s) and the functionality specified in the ADSP-21363 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 read by reading the REVPID register either via JTAG or DSP code.

The following DSP code can be used to read the register:
<UREG> = REVPID;

Silicon REVISION	REVPID[7:4]
0.5	0101
0.3	0011

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
04/08/2011	K	G	Added anomalies: 07000024 & 07000025
10/14/2010	J	F	Added anomalies: 07000022 & 07000023
08/13/2009	I	D	Modified anomalies: 07000009, Added common note on Tools action for all the core related anomalies
02/25/2009	H	D	Added anomalies: 07000021
07/19/2007	G	0	Modified anomalies: 07000010 - Removed some special characters
12/28/2006	F	0	Removed anomalies: 07000020-This information is added to the HRM
11/24/2006	E	0	Added anomalies: 07000020
10/17/2006	D	0	Document Format Update
06/21/2006	C	0	Modified anomalies: 07000011
03/14/2006	B	0	Added anomalies: 07000011
06/08/2005	A	PrA	Initial release

NR003082K

One Technology Way, P.O.Box 9106, Norwood, MA 02062-9106 U.S.A.
Tel: 781.329.4700 **www.analog.com**
Fax: 781.461.3113 **©2011 Analog Devices, Inc. All rights reserved.**

SUMMARY OF SILICON ANOMALIES

The following table provides a summary of ADSP-21363 anomalies and the applicable silicon revision(s) for each anomaly.

No.	ID	Description	0.3	0.5
1	07000008	Instruction Cache needs to be initialized	x	.
2	07000009	Some Core Stalls not executed properly	x	x
3	07000010	Memory write operations can fail under certain conditions while DMA to internal memory is in progress	x	x
4	07000011	SPI can generate spurious clock for an additional word when used in receive DMA mode at maximum SPICLK frequency	x	x
5	07000021	Incorrect Popping of stacks possible when exiting IRQx/Timer Interrupts with DB modifiers	x	x
6	07000022	Conditional FLAG instructions involving DAG index registers must not be followed immediately by an instruction that uses the same index register	x	x
7	07000023	Hardware /IRQx interrupts may cause unpredictable behavior under some conditions	x	x
8	07000024	VDSP Statistical profiler may provide incorrect profile information for idle instructions	x	x
9	07000025	After an emulator halt at the instruction before ‘idle' instruction, the Core Timer stops decrementing even after code execution restarts	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-21363 including a description, workaround, and identification of applicable silicon revisions.

1. 07000008 - Instruction Cache needs to be initialized:

DESCRIPTION:

The instruction cache in the ADSP-2136x processor is not properly initialized upon power-up. As a result of this the first data read (cache hit) from the cache may be incorrect, possibly causing the processor to execute an invalid instruction. This sort of scenario can apply in code that includes PM data accesses within a loop. (In the second iteration of the loop, the PM data access instruction will be read from cache by the program sequencer, and due to this issue the instruction from cache may be invalid.

WORKAROUND:

The workaround for this issue is to place the code snippet below into the processor's initialization routine. It will force the cache to be initialized properly, and after executing this code snippet the instruction cache will function reliably. It is important to execute this code before the instruction cache is to be relied upon.

```
BIT SET MODE2 CADIS; // Disable the cache
NOP;                 // One nop for effect latency
READ CACHE 0;        // Read cache instruction
BIT CLR MODE2 CADIS; // Re-enabling the cache.
```

Note: 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):

0.3

2. 07000009 - Some Core Stalls not executed properly:

DESCRIPTION:
Under certain specific conditions outlined below, 3 types of stalls that are normally incurred do not get executed. In certain cases this can cause unexpected code operation.

1.Multiplier stalls:
In normal operation, if both operands in a multiplier instruction are produced by either the multiplier or the ALU in the previous instruction, the pipeline is stalled for 1 cycle.

```
[1] F0=F0+F4, F1=F0-F4, R14 = DM(I0,M0);
[2] F4=F0*F1; <---- this should stall
```

Anomalous behavior:
If: The memory access in instruction 1 is to a memory-mapped IOP register.
-Or-
DMA is simultaneously accessing the same bank as the memory access in instruction 1.

Then: Stall does not occur, and this results in the product in F4 being incorrect.

2.Register writes mode1 / mode2:
In normal operation, a stall cycle is added after any write to MODE1 or MODE2 registers.

-A stall is also added when the content of MODE1 and MODE2 registers are modified through the bit manipulation instruction.

-The value of MODE1 also changes when PUSH STS or POP STS instructions are executed, or when the sequencer branches to or returns from an ISR involving push/pop of status stack (such as any of the hardware interrupts like IRQ and core timer interrupts). In such cases the pipeline is stalled for a cycle.

```
Example1:
[1] MODE1= DM(I0,M0); /*Enable bit reverse addressing for I8 */
[2] PM(I8,M8)=R14; /* stalls for a cycle but unaffected by mode setting */
[3] PM(I8,M8)=R14; /* performs bit reversed mode of addressing */
[4] …

Example2:
[1] MODE1= DM(I0,M0); /*Enable bit reverse addressing for I8 */
[2] r0 = r1 + r2; /* stalls for a cycle */
[3] PM(I8,M8)=R14; /* performs bit reversed mode of addressing */
[4] …
```

Anomalous behavior:
If: The memory access in instruction 1 is to a memory-mapped IOP register.
-Or-
DMA is simultaneously accessing the same bank as the memory access in instruction 1.

Then: Stall does not occur, and this results in instruction 3 not being affected by the mode change.

3.PCSTK load & RTS/RTI combination:
When PCSTK is loaded, and an RTS/RTI is executed immediately afterward, there is a stall as the return waits for a writeback of PCSTK before the return.

```
Example1:
[1] PCSTK = DM(I0,M0);
[2] RTS;

Example2:
[1] PCSTK = DM(I0,M0);
[2] RTI;
```

Anomalous behavior:
If: The memory access in instruction 1 is to a memory-mapped IOP register.
-Or-

DMA is simultaneously accessing the same bank as the memory access in instruction 1.

Then: Stall does not occur, and this results in the RTS/RTI vectoring to some unknown location instead of the value from PCSTK.

Note: Multiplier stalls case is fixed for the 0.4 and 0.5 revision of the silicon.

WORKAROUND:

For cases 1 and 3, add a nop between instructions 1 and 2.
For case 2 add a nop between instructions 2 and 3.

Note: 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):

0.3, 0.5

3. 07000010 - Memory write operations can fail under certain conditions while DMA to internal memory is in progress:

DESCRIPTION:
If an instruction modifies a register and the same register is the source to a memory write, and a DMA happens to same block as the memory write in the next cycle, under the following conditions, the modified value of the register is written to memory instead of the old one.

Examples of instructions that modify a register and the same register are written to memory are:

```
1. R0 = R1-R2, DM(I0,M0) = R0; // The memory access can be either DM/PM
2. DM(I0,M0) = R0, R0 = PM(I8,M8);
```

The conditions under which we see the issue are:
1. The failing instruction is the first instruction in a 1,2, or 4 instructions long loop.

Example:

```
lcntr = 8, do (pc,1) until lce;
DM(I0,M0) = R0, R0 = PM(I8,M8); //and DMA occurs in next cycle.
// The new value of R0 (fetched through PM) is written into DM memory
lcntr=0x7, do ST2_IN_BFLY_T2 until lce; //2 instr long loop
f4=f2+f4, dm(i3,m6)=r0, r0=pm(i11,m11);
ST2_IN_BFLY_T2: f4=pass f2, dm(i4,m6)=r4, r2=pm(i11,m11);
```

2. When the compute of the failing instruction generates both the operands of multiplier in the next instruction
Example:

```
F0=F0+F4, F1=F0-F4, DM(I0,M0) = R1; // and DMA occurs in next cycle.
// The new value of R1 (output of compute) is written into DM memory
F4=F0*F1;
```

3. When the failing instruction is followed by a conditional branch and any of the following two happens
a. A compute in the failing instruction affects the condition of the branch
Example:

```
F0=F0+F4, PM(I10,M10) = R0; // and DMA occurs in next cycle.
//The new value of R0 (output of compute) is written into PM memory
IF EQ JUMP(PC,0x12);
```

b. A compute in the instruction preceding the failing instruction affects the condition of branch
Example:

```
F0=F0+F4;
DM(I0,M0) = R0, R0 = PM(I8,M8); // and DMA occurs in next cycle.
//The new value of R0 (fetched through PM) is written into DM memory
IF EQ JUMP(PC,0x12);
```

4. When the failing instruction contains a floating point multiplication and is followed by compute operation involving any fixed point operand register executing in ALU or shifter.

Example:

```
F0=F0*F4, DM(I0,M0) = R0;
F5=FLOAT R1;
//The new value of R0 (output of multiply) is written into PM memory
```

WORKAROUND:
Move DMA to another block of memory OR,
1. For case1, unroll the loop to make the loop length more than 4.

2. For case 2, 3a, 3b and 4 insert an unrelated instruction between the failing instruction and the instruction following it.

Note: 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):
0.3, 0.5

4. 07000011 - SPI can generate spurious clock for an additional word when used in receive DMA mode at maximum SPICLK frequency:

DESCRIPTION:
SPI can generate spurious clock for an additional word when the following three conditions are met:-

1. The SPI is configured for 32-bit, receive master DMA mode and performs back to back DMA transfers.
2. The SPI DMA is disabled and re-enabled between back to back DMA transfers without disabling the SPI itself.
3. The SPI is operated at the maximum SPICLK frequency (SPIBAUD = 2).

When the above three conditions are met, SPI can generate clock for an additional word even after the FIFO and the receive buffer is full. This clock for another word can cause the receive buffer overflow error which will result in loss of the additional received data word.

WORKAROUND:
1. Do not disable the SPI DMA between back to back DMA transfers.

2. If the application demands disabling SPI DMA between back to back DMA transfers then the sequence given below must be followed:
   - Disable the SPI DMA
   - Disable the SPI
   - Clear the FIFO and the receive buffer
   - Configure the DMA descriptors and Enable the SPI
   - Enable the SPI DMA

APPLIES TO REVISION(S):
0.3, 0.5

5. 07000021 - Incorrect Popping of stacks possible when exiting IRQx/Timer Interrupts with DB modifiers:

DESCRIPTION:
If a delayed branch modifier (DB) is used to return from the interrupt service routines of any of IRQx (hardware) or timer interrupts, the automatic popping of ASTATx/ASTATy/MODE1 registers from the status stack may go wrong.

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

This anomaly affects only IRQx and Timer Interrupts as these are the only interrupts that cause the sequencer to push an entry onto the status stack.

WORKAROUND:
Do not use (DB) modifiers in instructions exiting IRQx or Timer ISRs. Instructions in the delay slots should be moved to a location prior to the branch.

Note: 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):
0.3, 0.5

6. 07000022 - Conditional FLAG instructions involving DAG index registers must not be followed immediately by an instruction that uses the same index register:

DESCRIPTION:
In the following instruction sequence shown below:

```
INSTR1: Compute;
INSTR2: If COND DM (Ia,Mb); //COND maybe based on the INSTR1, or any other condition
INSTR3: DM (Ia,Mc);
```

The value of the DAG index register in **`INSTR3`** will either be **`Ia`** or **`(Ia+Mb)`** depending on whether **`INSTR2`** was aborted or executed.

If **`COND`** is based on an external **`FLAG`** condition (for example, say **`FLAG2_IN`**) which is set asynchronously by an external source or event, the necessary internal stalls which would result in the DAG index register getting the correct value do not take effect, and consequently the value of the DAG index register may not contain the correct and expected value.

WORKAROUND:
Separate the second and third instructions in the above sequence by at least two NOP's.

Note: 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):
0.3, 0.5

7. 07000023 - Hardware /IRQx interrupts may cause unpredictable behavior under some conditions:

DESCRIPTION:
The /IRQx interrupts may lead to an unpredictable behavior of the processor under some conditions. The problem happens under a specific timing window between the /IRQx signal and the internal clock references. At this point, a couple of malfunctions (out of many possible ones) that may occur are - the fetch address corresponding to the /IRQx interrupt from the IVT is corrupted, this can lead to corrupted instruction and/or the sequencer branching to unknown/invalid location. This problem is applicable for both edge-sensitive and level-sensitive configurations of the /IRQx interrupts and is seen only when the processor is operated with the core-clock frequency above 285MHz.

WORKAROUND:
1) DAI interrupts should be used for external interrupts, instead of /IRQx interrupts.
2) If possible, the core clock frequency can be reduced to 285MHz or lower.

APPLIES TO REVISION(S):
0.3, 0.5

8. 07000024 - VDSP Statistical profiler may provide incorrect profile information for idle instructions:

DESCRIPTION:
The VisualDSP++ statistical profiler may indicate incorrect processor time expenditure when idle instructions are used. This occurs if the EMUPC register does not get updated when the program sequencer is executing the idle instruction.

In the example code below, the ISR_func1 is executed at the ISR's interrupt rate and most of the processor time is spent in executing the idle instruction. However, the statistical profiler may show incorrectly that most of the processor time is spent executing instructions in the ISR_func1. This occurs when the program sequencer branches to execute the IDLE instruction from the RTI instruction. When this occurs, the EMUPC register may still retain the previously executed ISR_func1's Program Counter value.

```
_main:
……
Test:
Idle;
Jump Test;
_main.end:

ISR_func1:
……
……
Instruction_x;
Instruction _y;
RTI;
```

Note: This issue impacts only the Statistical Profiling information.

WORKAROUND:
NOP instruction can be used instead of IDLE instruction wherever possible to get correct statistical profile information.

APPLIES TO REVISION(S):
0.3, 0.5

9. 07000025 - After an emulator halt at the instruction before 'idle' instruction, the Core Timer stops decrementing even after code execution restarts:

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 at an instruction just before an idle instruction, the core timer remains halted even after the code execution is resumed.

In the example code below, the core timer remains halted after code execution is resumed following the core halt at Instruction1. The same behavior is seen if Instruction1 is executed by 'single stepping'.

```
Enable_Core_timer:
……
Bit set MODE1 TIMEN;
……
……
Instruction1;                // Placing breakpoint here causes the anomaly
Idle;
Instruction2;
Instruction3;
```

Note: This issue impacts only Emulator session debug.

WORKAROUND:
None

APPLIES TO REVISION(S):
0.3, 0.5

This page intentionally left blank.