TMS320C55x DSP CPU

Programmer's Reference

Supplement

SPRU652G
November 2002
Revised February 2005

REVISION HISTORY

This revision history highlights the technical changes made to SPRU652E to generate SPRU652F. It also highlights the technical changes made to SPRU652F to generate SPRU652G; these changes are marked by "**[Revision G]**" in the Revision History table below.

Scope: Added Advisory CPU_118 and updated Section 1.1, etc.
Added Advisory CPU_119 **[Revision G]**

PAGE(S) NO.	ADDITIONS/CHANGES/DELETIONS
6	Table 1, Quick Reference Table: − added CPU_118, CPU May Halt After Returning From an Interrupt Service Routine When Operating in Emulation (Debug) Mode
10	Updated Section 1.1, Device and Development-Support Tool Nomenclature
47	CPU_116, Interrupted Nesting of Loops May Stop CPU Execution: − Details section: corrected hierarchy of bulleted list
50	Added CPU_118, CPU May Halt After Returning From an Interrupt Service Routine When Operating in Emulation (Debug) Mode
51	Added CPU_119, Due to Improper Update of the DBSTAT Register, the Debugger May Halt at Code Locations Where No Breakpoints are Set **[Revision G]**

Contents

1 Introduction

This document describes functional exceptions to the CPU behavior described in the *TMS320C55x DSP CPU Reference Guide* (literature number SPRU371). Non-CPU issues are described in the device-specific silicon errata. [For advisories on the OMAP5910 dual-core processor, see the *OMAP5910 Dual-Core Processor Silicon Errata* (literature number SPRZ016).] See Section 4 for a listing of related documentation.

A quick reference table (Table 1) is included so that advisory descriptions may be quickly located from the short description of each advisory. The advisory number in the first column of the table is referenced in the title of each advisory description that follows in Section 3. The columns on the right of the table indicate whether the advisory is present on the indicated silicon revision. The CPU advisories can be grouped into three primary categories:

- Parallel Execution Issues
  These issues are related to specific combinations of instructions executed in parallel. In most cases, these issues can be avoided by not executing the instructions in parallel.
- Pipeline-protection Issues
  These issues are cases where the pipeline is not properly automatically protected for specific instruction sequences. In most cases, these problems can be corrected by rearranging the instruction sequences, or by adding NOP instructions.
- Other Issues
  These are the issues that do not fall into one of the above two categories.

The legend following indicates which symbols are used for each case.

‖	The advisory **is** present on this silicon revision and is related to parallel instruction execution.
P	The advisory **is** present on this silicon revision and is related to pipeline protection.
X	The advisory **is** present on this silicon revision and not related to either of the above categories.
fixed	The advisory **is not** present on this silicon revision.
N/A	The advisory **is not** applicable to this device.

Table 1. Quick Reference Table

Advisory Number	Device: / Silicon Revision:	5501 All revs	5502 All revs	5503 All revs	5507 All revs	5509A All revs	5510 2.1 / 2.2	OMAP5910 All revs	OMAP5912 All revs
CPU_72	C54CM Bit Update and *CDP With T0 Index is not Pipeline-Protected	N/A	N/A	N/A	N/A	N/A	P	N/A	N/A
CPU_73	Certain Instructions not Pipeline-Protected From Resets	N/A	N/A	N/A	N/A	N/A	P	P	P
CPU_76	DELAY Smem Does not Work With Circular Addressing	N/A	N/A	N/A	N/A	N/A	X	X	X
CPU_79	IDLE Cannot Copy the Content of ICR to ISTR	N/A	N/A	N/A	N/A	N/A	X	N/A	N/A
CPU_80	Nested Local Repeat Corrupted After C54CM Bit Reset	N/A	N/A	N/A	N/A	N/A	P	N/A	N/A
CPU_81	WHILE Instruction in Slot #2 is not Protected	N/A	N/A	N/A	N/A	N/A	P	N/A	N/A
CPU_82	'if(cond true) goto' at the End of Local Repeat Fails	N/A	N/A	N/A	N/A	N/A	X	X	X
CPU_83	BRAF Updated Incorrectly in Certain Cases of Conditional Execution	N/A	N/A	N/A	N/A	N/A	X	X	X
CPU_84	SP/SSP Access Followed by a Conditional Execute is not Protected Against Interrupts	N/A	N/A	N/A	N/A	N/A	P	P	P
CPU_85	Local Repeat With C54CM = 1 may be Corrupted on its Last Iteration	N/A	N/A	N/A	N/A	N/A	X	X	X
CPU_86	Corruption of CSR or BCRx Register Read When Executed in Parallel With Write	N/A	N/A	N/A	N/A	N/A	‖	‖	‖
CPU_87	Context Restore Just Before Return Instruction Sometimes Fails	N/A	N/A	N/A	N/A	N/A	P	P	P
CPU_88	Incorrect Context Store of BRAF During Interrupt Servicing	N/A	N/A	N/A	N/A	N/A	‖	‖	‖
CPU_89	Internal Overflow not Detected When Using the Left Shift Command	N/A	N/A	N/A	N/A	N/A	X	X	X

Table 1. Quick Reference Table (Continued)

Advisory Number	Device: / Silicon Revision:	5501 All revs	5502 All revs	5503 All revs	5507 All revs	5509A All revs	5510 2.1 / 2.2	OMAP5910 All revs	OMAP5912 All revs
CPU_90	CPU Bypass Can Cause Corruption of a Read Following a Write	N/A	N/A	N/A	N/A	N/A	fixed	fixed	fixed
CPU_91	C16, XF, and HM Bits not Reinitialized by Software Reset	N/A	N/A	N/A	N/A	N/A	X	X	X
CPU_92	Consecutive C-Bus Accesses may not Work	N/A	N/A	N/A	N/A	N/A	X	N/A	N/A
CPU_93	Interrupted Conditional Execution After Memory Write may Execute Unconditionally in the D Unit	N/A	N/A	N/A	N/A	N/A	X	N/A	N/A
CPU_94	Interrupted Conditional Execution After Long Memory-Mapped Register Write is Executed Unconditionally in the D Unit / AD Unit	N/A	N/A	N/A	N/A	N/A	X	X	X
CPU_95	BRCx Decrement may not Work When gotoP24 is put at End of Blockrepeat With C54CM = 0	N/A	N/A	N/A	N/A	N/A	X	X	X
CPU_96	gotoP24 Within Blockrepeat Exits the Loop	N/A	N/A	N/A	N/A	N/A	X	X	X
CPU_97	RETA = Lmem || Lmem = RETA may not Work	N/A	N/A	N/A	N/A	N/A	‖	‖	‖
CPU_98	BANZ at the End of Inner Loop in Native Mode may Corrupt Program Flow	N/A	N/A	N/A	N/A	N/A	X	X	X
CPU_99	Return_int (Under a Fast Return Configuration) may Cause Improper Operation of Single Repeats and Conditional Executions	N/A	N/A	N/A	N/A	N/A	X	X	X
CPU_100	Interrupted Single Repeat is not Resumed After RETI	N/A	N/A	N/A	N/A	N/A	X	X	X
CPU_102	Page Register Update and CPU Bypass Corrupts Following Memory Read	N/A	N/A	N/A	N/A	N/A	X	X	X
CPU_103	C54x Instruction, FRET[D] is not Protected Against Prior C54CM Bit Update	X	X	X	X	X	‖	N/A	N/A

Table 1. Quick Reference Table (Continued)

Advisory Number	Device: / Silicon Revision:	5501 All revs	5502 All revs	5503 All revs	5507 All revs	5509A All revs	5510 2.1 / 2.2	OMAP5910 All revs	OMAP5912 All revs
CPU_104	Blockrepeat Corrupted if Preceded by Localrepeat With C54CM = 1 and BRC0 = 0	X	X	X	X	X	X	X	X
CPU_106	Move (Shift and Store) Instructions Incompatible With C54x When C54CM Bit = 1 and SST Bit = 1	X	X	X	X	X	X	X	X
CPU_107	Conditional Call With False Condition Corrupts RETA	X	X	X	X	X	X	X	X
CPU_108	Long (32-Bit) Read From MMR Gets Corrupted	X	X	X	X	X	X	X	X
CPU_109	Bus Error Issued on Byte Access to I/O Space With Address Range 0x0 to 0x5f	X	X	X	X	X	X	X	X
CPU_110	Relative Branch in ISR Corrupts Program Flow When Localrepeat With C54CM = 1 is Interrupted	X	X	X	X	X	X	X	X
CPU_111	C54CM Bit Modification Followed by a mar Instruction Not Pipeline-Protected	P	P	P	P	P	P	P	P
CPU_112	Data Page Register and Stack Pointer Update Not Pipeline-Protected Against Data Move Instructions	P	P	P	P	P	P	P	P
CPU_114	ST2 Update and Dual-Memory Access With Circular Qualifier Not Pipeline-Protected	P	P	P	P	P	P	P	P
CPU_116	Interrupted Nesting of Loops May Stop CPU Execution	P	P	P	P	P	P	P	P
CPU_117	Updating BRC Prior to a Loop That Contains Only Single Repeats Incorrectly Decrements the RPTC	X	X	X	X	X	X	X	X

Table 1. Quick Reference Table (Continued)

Advisory Number	Device: / Silicon Revision:	5501 All revs	5502 All revs	5503 All revs	5507 All revs	5509A All revs	5510 2.1 / 2.2	OMAP5910 All revs	OMAP5912 All revs
CPU_118	CPU May Halt After Returning From an Interrupt Service Routine When Operating in Emulation (Debug) Mode	X	X	X	X	X	X	X	X
CPU_119	Due to Improper Update of the DBSTAT Register, the Debugger May Halt at Code Locations Where No Breakpoints are Set	X	X	X	X	X	X	X	X

1.1 Device and Development-Support Tool Nomenclature

To designate the stages in the product development cycle, TI assigns prefixes to the part numbers of all DSP devices and support tools. Each DSP commercial family member has one of three prefixes: TMX, TMP, or TMS (e.g., **TMS**320VC5502GZZ). Texas Instruments recommends two of three possible prefix designators for its support tools: TMDX and TMDS. These prefixes represent evolutionary stages of product development from engineering prototypes (TMX/TMDX) through fully qualified production devices/tools (TMS/TMDS).

Device development evolutionary flow:

TMX Experimental device that is not necessarily representative of the final device's electrical specifications

TMP Final silicon die that conforms to the device's electrical specifications but has not completed quality and reliability verification

TMS Fully qualified production device

Support tool development evolutionary flow:

TMDX Development-support product that has not yet completed Texas Instruments internal qualification testing.

TMDS Fully qualified development-support product

TMX and TMP devices and TMDX development-support tools are shipped against the following disclaimer:

"Developmental product is intended for internal evaluation purposes."

TMS devices and TMDS development-support tools have been characterized fully, and the quality and reliability of the device have been demonstrated fully. TI's standard warranty applies.

Predictions show that prototype devices (TMX or TMP) have a greater failure rate than the standard production devices. Texas Instruments recommends that these devices not be used in any production system because their expected end-use failure rate still is undefined. Only qualified production devices are to be used.

2 Important Notices About CPU Advisories

2.1 Prototype Silicon Advisory Information

The list of advisories included in this document may include all prototype versions of a device. As prototype silicon revisions become obsolete, they will be removed from this document. Please consult your local sales representative if you need information concerning previous silicon revisions not listed in this document.

2.2 Useful Information Regarding Assembler Diagnostic Messages

The TMS320C55x Assembler will generate three types of diagnostic messages when it detects a potential or probable Silicon Exception.

2.2.1 ERROR Diagnostics

The assembler generates ERROR diagnostics in cases where it can fully determine that the code will cause a silicon exception to occur on hardware.

2.2.2 WARNING Diagnostics

The assembler generates WARNING diagnostics in cases where it can fully determine that the code will cause a silicon exception to occur on hardware, but which, under certain circumstances, may not be an issue for the user.

2.2.3 REMARK Diagnostics

The assembler generates REMARK diagnostics in conditions where it can fully determine that the code may cause a silicon exception to occur on hardware, but the exception itself also depends on non-visible trigger conditions that the assembler has no knowledge of, such as whether interrupts are enabled.

Since the assembler cannot determine the state of these trigger conditions, it cannot know that the exception will affect this code. Therefore, it generates a REMARK to instruct the user to examine the code and evaluate whether this is a potential silicon exception situation. (Please see the following sections for how to suppress remarks in situations where you have determined that the other trigger conditions do not exist.)

Intended Treatment of REMARK Diagnostics

The intent of generating REMARK diagnostics is to inform the user that the code could potentially cause a silicon exception and that it should be reviewed by the user side by side with the trigger conditions and a determination be made whether the code is a potential silicon exception situation.

If the code is determined to be a potential silicon exception situation, users should modify their code to prevent that exception from occurring.

If users determine that their code will not cause a silicon exception based on the trigger conditions, then the REMARK that the assembler generates can be suppressed. There are two methods of doing so; please see the "Suppressing REMARK Diagnostics" section.

Suppressing REMARK Diagnostics

Once the user determines that a silicon exception REMARK diagnostic is not appropriate for the code as written, the REMARK diagnostic can be suppressed in one of the following ways.

REMARK Directives:

The .noremark/.remark directives can be used to suppress the generation of a REMARK diagnostic for particular regions of code. The .noremark directive turns off the generation of a particular REMARK diagnostic. The .remark directive re-enables the generation of a particular REMARK diagnostic.

A '.noremark ##' (where ## is the remark id) directive is placed at the beginning of the region, and a '.remark ##' directive is placed at the end of the region.

> **NOTE:** The .noremark/.remark directive combination should always be placed around the entire region of code that participates in the potential silicon exception. Otherwise, spurious diagnostics may still be generated.

Additionally, the user has the option of disabling a silicon exception diagnostic for the entire file by placing just the .noremark directive at the top of the assembly file. However, this may be dangerous if, during inevitable code maintenance, the code is modified by someone not familiar with all the exception conditions. Please take great care when using the directives in this manner.

REMARK Command-Line Options:

The compiler shell (cl55) supports a command line option to suppress a particular REMARK diagnostic. The shell option −ar# (where # is the assembler's silicon exception id as described above) will suppress the named REMARK for the entire scope of all assembly files compiled with that command. Using the option −ar without a number will suppress all REMARK diagnostics.

Again, this may be dangerous if, during inevitable code maintenance, the code is modified by someone not familiar with all the silicon exception conditions. Please take great care when using the command-line REMARK options. Using the .noremark/.remark directives covering the shortest possible range of source lines is much safer.

PENDING Assembler Notification Status:

In the advisory descriptions, an assembler notification status marked "Pending" indicates the current version of the code generation tools do not yet detect the condition. As versions of the tools are released, known issues are included.

3 C55x CPU Known Design Advisories to Functional Specifications

Advisory CPU_72	*C54CM Bit Update and *CDP With T0 Index is not Pipeline-Protected*

Revision(s) Affected: See Table 1

Details: When the C54CM bit in status register 1 (ST1_55) is set to 1, the T0 index for single/dual/coefficient memory accesses should be replaced with AR0 for 54x compatibility. Therefore, if a C54CM bit update is followed by an instruction utilizing the Data Address Generator and T0 index, a stall should be generated to postpone the Data Address Generator until the C54CM bit update is complete. In the following cases, the stall is not created and the incorrect index is used (AR0/T0):

Case 1

C54CM bit update by bit instruction
0−4 cycles
B-bus access with 'coef(*CDP+T0)' using address modifier: *ABS16(#k) or *(#k).

Case 2

C54CM bit update by MMR write
0−5 cycles
B-bus access with 'coef(*CDP+T0)' using address modifier: *ABS16(#k) or *(#k).

Algebraic example

```
bit(ST1,#5) = #1                        ; set C54CM (=1)
AC0 = *(#60h)*coef(*(CDP+T0))           ; T0 incorrectly used as index
```

Mnemonic example

```
BSET #5, ST1_55                         ; set C54CM (=1)
MOV (#60h)*coef(*(CDP+T0)), AC0         ; T0 incorrectly used as index
                                        ; 0xaaaa
```

Assembler Notification: Assembler (versions 2.00 and later) will generate a REMARK when this condition is found.

Workaround(s):

1. In the case where the C54CM bit is updated by a bit instruction, maintain at least 5 cycles (useful code or NOPs) between the C54CM bit update and the Data Address Generator instruction.
2. In the case where the C54CM bit is updated by a MMR write of ST1_55, maintain at least 6 cycles (useful code or NOPs) between the C54CM bit update and the Data Address Generator instruction.

Advisory CPU_73 — *Certain Instructions not Pipeline-Protected From Resets*

Revision(s) Affected: See Table 1

Details: In the following cases, instructions may not execute properly due to insufficient pipeline protection from reset conditions:

Case 1

The following instruction(s) is not executed properly when closely preceded by a hardware or software reset:

```
DP = #K16     ;OR
Data Address Generator operation affected by any status bit ;OR
if (cond) execute (AD Unit)
```

These instructions (which depend on ST0_55, ST1_55 and ST2_55) will not execute correctly if they are located in the first four instructions following the reset (including the delay slot in the reset vector).

Case 2

IFR0/1 or ST1 MMR read instructions may return invalid read data when followed by a software reset.

Case 3

The BRAF bit is not cleared correctly by a software reset which follows the bit (ST1, #BRAF) = #1 instruction.

Assembler Notification: None

Workaround: Use the appropriate workaround, based on the Case.

Case 1

Do not put the following instruction(s) in the delay slot (last four bytes after the interrupt vector). Also do not use the following instruction(s) as the first, second, or third instructions at beginning of program space:

```
DP = #K16     ;OR
DAGEN-operation affected by any status bit     ;OR
if (cond) execute (AD Unit)
```

Case 2

Ensure at least 3 cycles between IFR0/1 or ST1 MMR read and a software reset.

Case 3

Ensure at least 5 cycles between bit(ST1, #BRAF) = #1 and a software reset

Advisory CPU_76 — *DELAY Smem Does not Work With Circular Addressing*

Revision(s) Affected: See Table 1

Details: When using circular addressing mode with the 'DELAY Smem' instruction in the following case:

smem = (end address of a circular buffer)

the incorrect destination address is used for the delay instruction. The destination address used is (end of circular buffer)+1, which is outside of the circular buffer. The correct functionality would be for the destination address to wrap around to the beginning address of the circular buffer.

Assembler Notification: Assembler (version 2.3 and later) will detect the use of delay (Smem) and generate a REMARK.

Workaround: Do not use circular addressing mode with the 'DELAY' instruction.

Advisory CPU_79 — *IDLE Cannot Copy the Content of ICR to ISTR*

Revision(s) Affected: See Table 1

Details: When an IDLE instruction is decoded, the content of the Idle Configuration Register (ICR) is supposed to be copied to the Idle Status Register (ISTR) when the instruction preceding the IDLE completes its write phase. However, during the following sequence, the ICR to ISTR copy does not happen:

1. IDLE is decoded.
2. A wakeup interrupt condition (NMI or any maskable interrupt which is enabled in the IER0/IER1 registers) is captured or is currently pending.
3. The ICR to ISTR would normally happen here, but does not occur.

Assembler Notification: None

Workaround: Make sure all interrupts are masked (disabled) via the IER0/IER1 registers before IDLE instruction is decoded. This workaround does not work for the NMI interrupt.

Advisory CPU_80 — *Nested Local Repeat Corrupted After C54CM Bit Reset*

Revision(s) Affected: See Table 1

Details: When the following conditions occur:

- A local repeat follows another local repeat (nested local repeats).
- The first local repeat is stalled in the address phase due to a C54CM bit update from 1 to 0,

The CPU jumps to the wrong instruction address when leaving the outer repeat loop.

Algebraic example

```
bit(ST1,#5) = #0
nop
nop                        ;insert additional NOP here to implement workaround
localrepeat {              ;instruction stalled in address phase
       localrepeat {
              <instruction>
              <instruction>
              <instruction>
              :
       }
       :
}                          ;instruction pointer is incorrect upon leaving outer
                            loop
```

Assembler Notification: Assembler (version 2.30 and greater) will generate a WARNING when these conditions occur.

Workaround(s):

1. In the case of a C54CM bit update by a bit instruction to register ST1. Ensure that the C54CM bit update occurs a least four (4) cycles before the first local repeat.
2. In the case of a C54CM bit update by an MMR instruction. Ensure that the C54CM bit update occurs a least five (5) cycles before the first local repeat.

Advisory CPU_81 — *WHILE Instruction in Slot #2 is not Protected*

Revision(s) Affected: See Table 1

Details: When WHILE instruction is located as second slot and single repeated instruction follows after one null slot that is caused by any pipeline discontinuity, the WHILE instruction is not pipeline protected. If the single repeat instruction follows WHILE instruction immediately or there are more than one null slot, it works.

Assembler Notification: Assembler (version 1.83 and greater) will generate an ERROR when a WHILE operation is found in the second position of a parallel pair.

Workaround: Do not use the WHILE instruction in the second slot of a parallel instruction pair.

Advisory CPU_82

'if (cond true) goto' at the End of Local Repeat Fails

Revision(s) Affected: See Table 1

Details: Within any local repeat block if a conditional branch instruction is placed at the second to last position, and the branch target is at the last position of the loop, the program flow is corrupted. This is the case regardless of whether the local repeat is the outer loop or a nested inner loop.

Algebraic example

```
localrepeat{
.
.
.
if (cond true) goto TARGET

TARGET
nop
}
.
```

Assembler Notification: None

Workaround: Do not use this instruction sequence.

Advisory CPU_83

BRAF Updated Incorrectly in Certain Cases of Conditional Execution

Revision(s) Affected: See Table 1

Details: When C54CM=1 and one of the following cases occurs, the BRAF bit is modified regardless of the condition.

- if(cond=false)Execute(D_unit) || bit(ST1, @BRAF) = #0/1
- while(cond=false && (RPTC < k8)) bit(ST1, @BRAF) = #0/1

Assembler Notification: Assembler (version 2.3 and greater) will attempt to detect the cases above and generate a WARNING.

Workaround(s):

1. Use the AD-unit instead of the D-unit in the conditional execution instruction
   OR
   Do not use parallelism (use conditional execute of next instruction as opposed to conditional execute of parallel instruction).
2. Do not use a bit instruction that modifies BRAF within the WHILE instruction

Advisory CPU_84 *SP/SSP Access Followed by a Conditional Execute is not Protected Against Interrupts*

Revision(s) Affected: See Table 1

Details: Any of the following instructions are not protected against interrupts when followed by a AD-unit conditional execute instruction for which the condition is false. (This exception only applies to conditional execution of the next instruction and not a conditional execute of a parallel instruction):

- MMR-read access to SP/SSP
- dst = XSP/XSSP
- dbl(Lmem) = XSP/XSSP
- push_both(XSP/XSSP)
- XSP/XSSP = pop()
- MMR-write access to SP/SSP

Algebraic example

```
...{
nop
SP = SP - #1
if (TC1) execute (AD_Unit) ;where TC1=0, condition is false.
<interrupt occurs>
AR6 -= #1
...
```

Assembler Notification: Assembler (version 2.3 and greater) will attempt to identify a code sequence that may cause the exception, and will generate a REMARK.

Workaround(s):

1. When SP/SSP is read in the read phase, insert two (2) NOPs between the SP/SSP instruction and the conditional execute instruction.
2. When SP/SSP is read or written in the execute phase, insert three (3) NOPs between the SP/SSP instruction and the conditional execute instruction.
3. When SP/SSP is written in the write phase, insert four (4) NOPs between the SP/SSP instruction and the conditional execute instruction.

Advisory CPU_85 — *Local Repeat With C54CM = 1 may be Corrupted on its Last Iteration*

Revision(s) Affected: See Table 1

Details: Under the following conditions during a local repeat loop:

- C54CM = 1
- The program fetch is occurring to restart the last iteration of the local repeat loop
- The program fetch is stalled

The local repeat may be overwritten even though the last iteration has not been completed.

Assembler Notification: Assembler (version 2.3 and greater) will generate a WARNING when a .C54CM_ON directive is seen and a local repeat is encountered.

Workaround: Do not use local repeat loops with C54CM = 1.

Advisory CPU_86 — *Corruption of CSR or BCRx Register Read When Executed in Parallel With Write*

Revision(s) Affected: See Table 1

Details: Under the following conditions:

- CSR, BRC0, or BRC1 register is read in the EXE phase in parallel with a write to the same register
- The instruction is stalled due to a previous write access

The register read may be corrupted, returning the new value from the register write instruction. The possible parallel instruction pairs which may cause this condition are as follows:

```
Smem  = CSR   ||  CSR = TAx     ;Smem should be updated by old register value, but
Smem  = CSR   ||  CSR = Smem    ;updated to TAx value instead

Smem  = BRC0  ||  BRC0 = TAx
Smem  = BRC0  ||  BRC0 = Smem
TAx   = BRC0  ||  BRC0 = TAx
TAx   = BRC0  ||  BRC0 = Smem

Smem  = BRC1  ||  BRC1 = TAx
Smem  = BRC1  ||  BRC1 = Smem
TAx   = BRC1  ||  BRC1 = TAx
TAx   = BRC1  ||  BRC1 = Smem
```

Assembler Notification: Assembler (version 2.3 and greater) will detect the above parallel pairs and generate a WARNING.

Workaround: Do not execute these instructions in parallel.

Advisory CPU_87 *Context Restore Just Before Return Instruction Sometimes Fails*

Revision(s) Affected: See Table 1

Details: A context restore just before the return instruction sometimes fails. There are two cases in which this condition may occur:

Case 1: When the C54CM bit in ST1_55 is updated via MMR write just before the return instruction, a failure may occur. In the following sequence:

```
*(ST1_55) = <value>
return
```

the new value of the C54CM bit is not used by the return instruction. This may eventually lead to a BRAF recovery error. When C54CM=1, BRAF is not recovered by return. When C54CM=0, BRAF is recovered.

This failure occurs under the following conditions:

- C54CM bit is modified by ST1_55 context restore, AND
- the return condition is either 'return' with slow-return configuration, OR, 'if() return' with fast or slow return configuration.

Case 2: Altering the BRAF bit just before 'return_int' instruction. In the following sequence:

```
C54CM = #1
...
any BRAF update
return_int
```

In the fast-return configuration, BRAF is recovered immediately after return_int is decoded (along with return address). Due to lack of pipeline protection, the BRAF contents recovered by 'return_int' is overwritten by the instruction preceding 'return_int'.

This failure occurs under the following conditions:

- C54CM = 1, AND
- the return condition is either 'return' with fast-return configuration.

Assembler Notification: Assembler (version 2.3 and greater) will generate a REMARK when it detects the above instruction sequences.

Workaround: Use one of the following workarounds.

Case 1: Insert at least one NOP between the MMR access and the return instruction.

Case 2: Do not recover the BRAF context with an instruction that accesses BRAF. Instead, let the return instruction recover the BRAF content.

Advisory CPU_88 *Incorrect Context Store of BRAF During Interrupt Servicing*

Revision(s) Affected: See Table 1

Details: When an interrupt is serviced while a blockrepeat loop is active, the context pushed onto the stack incorrectly stores the BRAF bits as 0. Upon returning from the interrupt service routine, the CPU acts as if no loop is active. The program execution will continue sequentially past the end of the active loop. in other words, the blockrepeat loop is not re-activated upon return from an interrupt.

This condition occurs when the second instruction of a parallel instruction pair is a call (only call L16 is legal for such an instruction pair). The condition can occur when these parallel instructions are placed before the loop as well as within the loop.

Algebraic example

```
...
<instruction 1> || call L16
...
blockrepeat{
...
<interrupt decoded>
...                        ; upon return from interrupt, loop becomes inactive.
}
```

OR

```
...
blockrepeat{
...
<instruction 1> || call L16
...
<interrupt decoded>
...                        ; upon return from interrupt, loop becomes inactive.
}
```

Assembler Notification: Assembler (version 2.3 and greater) will detect any instruction with a parallel call L16 and generate a REMARK.

Workaround: Since interrupts are asynchronous, the only workaround is NOT to utilize the following parallel instruction pair.

```
<instruction 1> || call L16
```

Advisory CPU_89 *Internal Overflow not Detected When Using the Left Shift Command*

Revision(s) Affected: See Table 1

Details: In native 55x mode (C54CM=0) when performing left shifts using 32-bit computational mode (M40=0) with the sign extension mode bit set to UNSIGNED (SXMD=0), any overflow in ACx should result in a saturate 40-bit value of 0x 00 7FFF FFFF. However, if ACx[39..32] = 0xFF and a left shift occurs with the shift value ≥ 0x8, then ACx gets zeroed.

Example

```
SXMD = #0
M40 = #0

AC2 = FF 0000 0000h
DR1 = 0x0008h
TARGET*AR4 = HI(saturate(AC2 << DR1))

; *AR4 = 0x0000

; Expected value should be *AR4 = 0x7FFF.
```

Assembler Notification: Assembler will emit a REMARK for any instruction containing a left shift of an accumulator (ACx) by DRx or a constant ≥ 8.

Workaround: None

Advisory CPU_90 *CPU Bypass Can Cause Corruption of a Read Following a Write*

Revision(s) Affected: See Table 1

Details: When the CPU writes data to memory there is typically a pipeline based latency that occurs before that value actually gets written. So when the same memory address is read immediately after it is written, the 55x incorporates a data bypass that reads that value directly from the internal write bus while the memory write is still occurring. However, if a pipeline stall occurs between memory write followed by read to the same address, then the read data may become corrupt. The following scenarios describe when this corruption may occur:

Case 1:

Write to memory
Read from same memory || anything causing CPU stall

Case 2:

Write to memory
Anything
Read from same memory || anything causing CPU stall

Case 3:

Write to memory
Anything
Anything
Read from same memory || anything causing CPU stall

Case 4:

Write to memory
Anything causing CPU stall
Read from same memory

Case 5:

Write to memory
Anything
Anything causing CPU stall
Read from same memory

Assembler Notification: Simulator c55xsimBBD.ccs is available to detect this bug.

Workaround: Avoid the CPU bypass. Insert at least three non-memory related instructions between memory write followed by corresponding memory read instructions to the same address. Three NOPs between write and read will work.

Example

```
Memory Write Operation
NOP
NOP
NOP
Memory Read Operation (to the same address)
```

Advisory CPU_91

C16, XF, and HM Bits not Reinitialized by Software Reset

Revision(s) Affected: See Table 1

Details: According to the specification, the software reset only affects IFR0/1, STO_55, ST1_55, and ST2_55. In this case, the reset value should be the same as those forced by a hardware reset (C16=0, HM=0, XF=1). Instead, the software reset does not affect the C16, XF, and HM bits and they retain their previous values.

Assembler Notification: Assembler will emit a REMARK on any "reset" instruction.

Workaround: Always initialize these bits as desired following reset.

Advisory CPU_92

Consecutive C-Bus Accesses may not Work

Revision(s) Affected: See Table 1

Details: When two C-bus accesses are performed consecutively, as show below, and the second instruction's C-bus memory access wait state is different from that of the D-bus, the second as well as forthcoming C-bus based instructions may read corrupted data from the C-bus.

Instruction 1

Memory read − 32-bit data read addressed by "Lmem".

Instruction 2

Dual memory read − Two 16-bit data reads by one of the following:

- Implicit dual memory read instruction, "Xmem" and "Ymem".
- Paralleled single memory read instructions, "Smem || Smem".
- Dual stack read instructions shown below.

```
dst1,dst2 = pop()
ACx = dbl(pop())
dst = s_pop()
dbl(Lmem) = pop()
dst,Smem = pop()
```

Instruction 3

Long or Dual memory access (i.e. C-bus use)

```
if (cond) return
return
return_int
```

Consecutive C-Bus Accesses may not Work (Continued)

Instruction 4

Long or Dual memory access (i.e. C-bus use)

NOTE: This problem can also occur when a when a dual memory read instruction is executed at the top of a local repeat followed by a long memory read at the end of the repeat.

Algebraic example

```
localrepeat{
  Dual memory read
       .
       .
       .
  Long memory read
}
```

Assembler Notification: Assembler will detect Lmem read followed by Dual mem read and REMARK. Assembler will also detect when this situation arises around the boundaries of a loop (blockrepeat or localrepeat − with Lmem read at the end of the loop in combination with a Dual mem read at the start of the loop), and emit a REMARK.

Workaround: Insert a NOP, or any other instruction that does not use C-bus, between first and second instruction.

Advisory CPU_93 *Interrupted Conditional Execution After Memory Write may Execute Unconditionally in the D Unit*

Revision(s) Affected: See Table 1

Details: When a memory write instruction is executed just before a conditional statement in the D Unit and an interrupt is asserted between the conditional execute and the next instruction to be executed based on the conditional's result, the next instruction may get executed regardless of the conditional's result as shown in the following examples.

Example 1

```
Memory Write
If (Conditional) Execute (D Unit)
<Hardware Interrupt Asserted>
Instruction to be executed based on the Conditional gets executed
regardless of the Conditional
```

Example 2

```
Localrepeat {
If (Conditional) Execute (D Unit)
<Hardware Interrupt Asserted>
Instruction to be executed based on the Conditional gets executed
regardless of the Conditional
```

Assembler Notification: Assembler will detect when a memory write is followed by a conditional execute on the D-unit and emit a REMARK. The assembler will also detect when this situation arises at the boundaries of a loop (blockrepeat or localrepeat − with memory write at the end of the loop and the conditional execute in the D-unit at the start of the loop) and emit a REMARK.

Workaround: Insert a NOP between the memory write instruction and the conditional execution.

Example

```
Memory Write
NOP
If (Conditional) Execute (D Unit)
<Hardware Interrupt Asserted>
Instruction to be executed based on the Conditional
```

Or, replace the:

```
If (Conditional) Execute (D Unit)
```

instruction with:

```
If (Conditional) Execute (AD Unit)
```

Advisory CPU_94

Interrupted Conditional Execution After Long Memory-Mapped Register Write is Executed Unconditionally in the D Unit / AD Unit

Revision(s) Affected: See Table 1

Details: When a long memory-mapped register (MMR)† write instruction is executed just before or during a conditional statement in the D unit / AD unit and:

- an interrupt is asserted between the conditional execute and the next instruction to be executed
- no single MMR write follows before or during the return from interrupt

then, the instruction to be executed based on the conditional gets executed regardless of the conditional's value as shown in the following examples.

Example 1

```
       long MMR Write
       .                                                ;No single MMR write
       If (Conditional) Execute (AD Unit / D Unit)      ;No single MMR write
       <Hardware Interrupt Asserted>
       Instruction to be executed based on the Conditional gets executed
       regardless of the Conditional
       .
       .
ISR    .                                                ;No single MMR write
       .                                                ;No single MMR write
       Return_Int                                       ;No single MMR write
```

† Long memory mapped register (MMR): Any of the following instructions that point to 0x0 − 0x5F with "Lmem". Such as:

dbl(Lmem) = pop()
dbl(Lmem) = ACx, copr()
dbl(Lmem) = LCRPV
dbl(Lmem) = src
dbl(Lmem) = ACx
dbl(Lmem) = saturate(uns(ACx))
Lmem = pair(DAx)
HI(Lmem) = HI(ACx) >> #1, LO(Lmem) = LO(ACx) >> #1
Lmem = pair(HI(ACx))
Lmem = pair(LO(ACx))
Lmem = dbl(coeff)

Interrupted Conditional Execution After Long Memory-Mapped Register Write is Executed Unconditionally in the D Unit / AD Unit (Continued)

Example 2

```
       If (Conditional) Execute (AD Unit / D Unit)    ;No single MMR write
       <Hardware Interrupt Asserted>
       Instruction to be executed based on the Conditional gets executed
       regardless of the Conditional
       .
       .
ISR    .
       .
       long MMR write
       .                                              ;No single MMR write
       Return_Int                                     ;No single MMR write
```

Example 3

```
       If (Conditional) Execute (AD Unit / D Unit)    ;No single MMR write
       <Hardware Interrupt Asserted>
       Instruction to be executed based on the Conditional gets executed
       regardless of the Conditional
       .
       .
ISR    .
       .
       long MMR write || Return_Int
```

Assembler Notification: Assembler will emit a REMARK on any return-from-interrupt instruction. It will avoid emitting this remark if it is able to determine that a single memory write occurs before the return-from-interrupt and there is no long MMR write between the single memory write and the return-from-interrupt instruction.

Workaround: Put a dummy single memory write (i.e., @#0x1F = AR0 || mmap() : 0x1F is a reserved space.) in front of all "Return_int" and ensure that no long memory writes are in parallel with a "Return_int."

Example

```
       long MMR Write
       .                                              ;No single MMR write
       If (Conditional) Execute (AD Unit / D Unit)    ;No single MMR write
       <Hardware Interrupt Asserted>
       Instruction to be executed
       .
       .
ISR    .                                              ;No single MMR write
       .                                              ;No single MMR write
       Single MMR write
       Return_Int                                     ;No single MMR write
```

Advisory CPU_95	*BRCx Decrement may not Work When gotoP24 is put at End of Blockrepeat With C54CM = 0*

Revision(s) Affected: See Table 1

Details: When a branch, such as a gotoP24, is performed at the end of blockrepeat with C54CM = 0, then the corresponding BRCx may not get decremented. This bug occurs in both outer and inner blockrepeats. See the following example.

Example

```
            BRC = x
            blockrepeat{

            goto tgt  ; assembled to gotoP24

            .      .
      tgt:  BRC == x or (x-1) ?
            }
```

NOTE: If the destination of the goto is within a 16-bit range (i.e., gotoL16) is assigned, this problem does not occur.

Assembler Notification: The assembler will emit a WARNING if a goto P24 instruction occurs at the end of a blockrepeat loop.

Workaround: Do not put a gotoP24 instruction at the end of a blockrepeat.

Advisory CPU_96 — *gotoP24 Within Blockrepeat Exits the Loop*

Revision(s) Affected: See Table 1

Details: When a branch, such as a gotoP24, occurs within a blockrepeat with C54CM = 0 and its target is within the same loop, the loop ends immediately. If a nested loop starts after the branch, it is handled as a non-nested one with the 1st-level (RSA0/REA0 utilized, BRC0 decremented). See the following examples.

Example 1

```
              blockrepeat{
                .
                goto tgt
                .
       tgt:     .
                .
              }              ; Exit from the loop regardless BRCx value.
```

Example 2

```
              blockrepeat{
                .
                goto tgt
                .
       tgt:     blockrepeat { ; Regarded as outer loop, use of RSA0/REA0/BRC0
                      .
                    }
                .
              }
```

NOTE: If the destination of the goto is within a 16-bit range, (i.e., gotoL16) is used, this problem does not occur. This implies that the size of the blockrepeat must be greater than 0x8000.

Assembler Notification: The assembler will emit an ERROR when a goto P24 instruction occurs in a blockrepeat loop *and* its target is also defined in the same loop.

Workaround: Do not put a goto instruction, in which the target is within the same loop, in a blockrepeat which is greater than 0x8000 in size.

Advisory CPU_97	*RETA = Lmem || Lmem = RETA may not Work*

Revision(s) Affected: See Table 1

Details: RETA = Lmem || Lmem = RETA can be used to swap the data between Lmem and RETA as shown below:

New RETA <− Old Lmem

Old RETA −> New Lmem

However, when this store operation is stalled during a parallel execution, the content of the old RETA is lost as shown below:

New RETA <− Old Lmem

New RETA −> New Lmem

Example

Before execution : RETA is 0x00123456, Lmem is 0xffffffff

After execution : RETA is 0xffffffff, Lmem is 0xffffffff (Should be 0x00123456)

Assembler Notification: The assembler will emit a WARNING if "RETA = Lmem || Lmem = RETA" is specified.

Workaround: Do not use this parallel execution.

Advisory CPU_98

BANZ at the End of Inner Loop in Native Mode may Corrupt Program Flow

Revision(s) Affected: See Table 1

Details: When all of the following conditions are met:

- − C54CM=0 (Native mode),
- − Two blockrepeats (not localrepeat) are nested,
- − the instruction at end of inner loop is BANZ with a false condition,
- − the size of inner loop is less than 32 bytes.
- − the distance between the end of the two loops is greater than 0 and less than 24 bytes.

The program flow may be corrupted. The instruction immediately after the inner loop, although outside of the inner loop, gets executed during first iteration of the inner loop. See the example below.

Example

```
"INST-A" is executed at the first iteration of the inner loop.
bit(ST1,@C54CM) = #0
blockrepeat{
  .
  blockrepeat{                          ┐
    .                                   │ Less than 32 bytes
    .                                   │
    BANZ with false condition           ┘
  }                                     ┐
  INST-A                                │
  .                                     │ Greater 0 and less than 24 bytes
  }                                     ┘
```

Assembler Notification: The assembler will emit a REMARK if the following conditions are met:

- BANZ instruction is the last instruction in a nested block repeat
- size of inner block repeat loop is < 32 bytes
- the distance between the end of the outer block repeat loop and the end of the inner blockrepeat loop must be > 0, but < 24 bytes

Workaround: Put a NOP immediately after the BANZ within the inner loop

Advisory CPU_99 *Return_int (Under a Fast Return Configuration) may Cause Improper Operation of Single Repeats and Conditional Executions*

Revision(s) Affected: See Table 1

Details: Under a fast return configuration, when an interrupt is asserted during any of the following:

- Single repeat
- The single repeat is executed more than expected and if it is located at the end of blockrepeat / localrepeat, the BRCx may be not get decremented.

And if the corresponding Return_Int is stalled at an ADDRESS or ACCESS1 phase, then the following may occur:

- Just before a conditional execute instruction
- The instruction to be executed conditionally gets executed UNconditionally.

See the following examples.

Example 1

```
            AR0 = #0
            repeat(#15)
            AR0 = AR0 + #1              ; An interrupt is asserted here.
            AR0 = AR0 ^ #16             ; AR0 is expected to be 0 but not.
            if(AR0 != #0 ) goto ERROR
            .
      ISR:  .
            AR1 = AR1 – #1
            mar(*AR1+) || return_int    ; Stalled at ADDRESS phase.
```

Example 2

```
            .
            << An interrupt is asserted here >>
            if(cond=false)Execute(AD_Unit/D_Unit)
            Instruction to be executed conditionally always gets executed.
            .
      ISR:  .
            AR1 = AR1 – #1
            mar(*AR1+) || return_int    ; Stalled at ADDRESS phase.
```

Assembler Notification: The assembler will emit a REMARK on any return-from-interrupt instruction that does not have at least six NOP instructions preceding it (to avoid stall in ADDRESS or ACCESS1 phase of the pipeline).

Workaround: If the ”hold” feature, which can cause the CPU to stall, is not used, place 6 NOPs immediately before the return_int to avoid it from stalling.

Example

```
            nop
            nop
            nop
            nop
            nop
            nop
            return_int    ; No stalling during an ADDRESS or ACCESS1 phase.
```

Or don't use the fast return configuration

Advisory CPU_100 — *Interrupted Single Repeat is not Resumed After RETI*

Revision(s) Affected: See Table 1

Details: When an interrupt is asserted during any of the following single repeat instructions:

- while (cond && (RPTC < k8)) repeat
- repeat (k16)
- repeat (CSR)
- repeat (CSR) , CSR += DAx
- repeat (CSR) , CSR += k4
- repeat (CSR) , CSR −= k4
- repeat (k8)

The single repeat doesn't resume after returning from the interrupt under all of the following conditions:

- the restore of the repeat counter(RPTC) by MMR write in ISR is close(*) to the "return_int".
- the RPTC is 0 before the restore.

(*) if the instruction between restore RPTC and return_int is less than

- six bytes for the fast return configuration.
- two bytes for the slow return configuration.

Assembler Notification: The assembler will emit a REMARK on any return-from-interrupt instruction which has a RPTC register write within six instructions before it.

Workaround: Insert six nops between the restore RPTC and return_int for the fast return configuration. Insert two nops between restore RPTC and return_int for the slow return configuration.

Example

```
ISR:   .                    ; Fast return configuration.
       .
       @RPTC_L=pop()||mmap()
       nop
       nop
       nop
       nop
       nop
       nop
       return_int.
```

Advisory CPU_102

Page Register Update and CPU Bypass Corrupts Following Memory Read

Revision(s) Affected: See Table 1

Details: In the following sequence:

INST0: Any instruction

INST1: Any instruction

INST2: Any instruction

INST3: Write to a memory

INST4: Read from the same memory with a CPU STALL

INST5: Read from any memory (Does *not* have to be same address as INST3 or INST4)

INST5 may get wrong data from memory if the corresponding page register for the data read address generation has been updated with:

- a MMR write at INST0 or INST1 or INST2
- an EXE phase instruction at INST1 or INST2

The following table shows all Page registers with MMR address, instructions to update in EXE phase, and the events (data read) to be used.

Page Register (MMR Address)	EXE Phase Instruction	Used by (Candidate of INST5)
DPH (2Bh)	XDP = xsrc XDP = dbl(Lmem) XDP = popboth( )	−Direct addressing (CPL = 0)
SPH, SSPH (4Eh)	XSP = xsrc XSP = dbl(Lmem) XSP = popboth( ) XSSP = xsrc XSSP = dbl(Lmem) XSSP = popboth( )	−Direct addressing (CPL = 1) −All kinds of return INST. −All kinds of pop INST.
CDPH (4Fh)	XCDP = xsrc XCDP = dbl(Lmem) XCDP = popboth( )	−Indirect addressing with CDP pointer
ARx_H (None)	XARx = xsrc XARx = dbl(Lmem) XARx = popboth( )	−Indirect addressing with ARx pointer

Page Register Update and CPU Bypass Corrupts Following Memory Read (Continued)

In the following example, DR1 gets corrupted value.

Example

```
XAR1 = XAR3                  ; The page of AR1 is updated in EXE
nop
*AR6 = #0xABCD || DR3 = AR7  ; Write to a memory
DR0 = *AR6 || AC0 = DR3      ; Read from the same memory with CPU stall
DR1 = *AR1 || DR2 = *AR2     ; Reading data from *AR1 using XAR1
```

Assembler Notification: Pending

Workaround:

Case 1

Have at least three instructions between the page register update by the MMR write and the next write instruction as follows. As the assembler cannot detect "Page register update by MMR write", this condition must be confirmed by users.

Example

Page register update by MMR write in WRITE phase
INST
INST
INST
Write to a memory
Read from the memory
Read from memory/stack using the updated page register

Case 2

Have at least two instructions between the page register update by EXE phase instruction and the next write instruction as follows. If there is less than two instructions, it is planned that the assembler will reject it.

Example

Page register update by EXE phase instructions
INST
INST
Write to a memory
Read from the memory
Read from memory/stack using the updated page register

Case 3

Use `dst = mar(Smem)`, which is to update the Page Register at ADDRESS phase, as shown in the following example.

Example

dst = mar(Smem) ; dst can be XARn, XCDP, XDP, XSP, or XSSP
Write to a memory
Read from the memory
Read from memory/stack using the updated page register

Advisory CPU_103

C54x Instruction, FRET[D] is not Protected Against Prior C54CM Bit Update

Revision(s) Affected: See Table 1

Details: Similar to the behavior of the C54x instruction, the FRET[D] (Far Return) instruction depends on the status of the C54CM bit. It is necessary for the CPU to protect the FRET[D] instruction from prior C54CM bit update(s); however, this protection does not work when the stack mode is configured for the slow return mode.

Assembler Notification: The assembler (versions 2.7 and greater) will generate a remark if the C54M bit modification is within 5 instructions of the far return.

Workaround: In the source code, add five NOPs between the C54CM update (by MMR write or bit instruction for ST1) and C54x instruction, FRET. In case of FRETD, add four NOPs.

NOTE: Since FRET[D] is typically used for subroutine codes made for the C54x, it is very unusual to manipulate the C54CM bit within a subroutine.

Advisory CPU_104

Blockrepeat Corrupted if Preceded by Localrepeat With C54CM = 1 and BRC0 = 0

Revision(s) Affected: See Table 1

Details: A blockrepeat loop after a localrepeat loop can be corrupted, under the following conditions:

1. C54CM = 1 throughout the sequence.
2. BRC0 = 0 for the localrepeat loop (no iteration), non-zero for the blockrepeat loop.
3. No branch/call instruction between the end of the localrepeat and the blockrepeat instruction.

Example

```
bit(ST1, @C54CM) = #1
BRC0 = #0
localrepeat{      ; Performed only one time
     .
     .
}
     .            ; No absolute branch or relative branch which
                  ; distance is more than 24 bytes
     .            ; No absolute branch or relative branch which
                  ; distance is more than 24 bytes
BRC0 = #non-zero
blockrepeat{
     .
     .
}
```

Assembler Notification: Pending

Blockrepeat Corrupted if Preceded by Localrepeat With C54CM = 1 and BRC0 = 0 (Continued)

Workaround: Do one of the following:

- Do not use localrepeat with C54CM = 1 and BRC0 = 0
- Replace the localrepeat with blockrepeat
- Insert absolute branch between the localrepeat and blockrepeat (e.g., specified by "<<")

```
        localrepeat{
          .
          .
        }
        .vli_off            ; Force to use absolute branch <<
        goto LABEL          ; Absolute branch              <<
        .vli_on             ; cancel .vli_off              <<
LABEL:    .                 ; Branch target
          .
          .
        BRC0 = #non-zero
        blockrepeat{
          .
          .
        }
```

Advisory CPU_106

Move (Shift and Store) Instructions Incompatible With C54x When C54CM Bit = 1 and SST Bit = 1

Revision(s) Affected: See Table 1

Details: Under the following conditions, an overflow is incorrectly detected in the C54x Compatibility Mode:

- C54CM = 1, SST = 1
- The shift direction is to the left (<< positive number)
- Any of the following 13 instructions is performed:
  - Smem = LO(ACx << Tx)
  - Smem = HI(rnd(ACx) << Tx)
  - Smem = LO(ACx << SHIFTW)
  - Smem = HI(ACx << SHIFTW)
  - Smem = HI(rnd(ACx << SHIFTW))
  - Smem = HI(saturate(uns(rnd(ACx << Tx))))
  - Smem = HI(saturate(uns(rnd(ACx << SHIFTW))))
  - ACy = rnd(Tx * Xmem) , Ymem = HI(ACx << T2) [, T3 = Xmem]

Move (Shift and Store) Instructions Incompatible With C54x When C54CM Bit = 1 and SST Bit = 1 (Continued)

- ACy = rnd(ACy + (Tx * Xmem)) , Ymem = HI(ACx << T2) [, T3 = Xmem]
- ACy = rnd(ACy − (Tx * Xmem)) , Ymem = HI(ACx << T2) [, T3 = Xmem]
- ACy = ACx + (Xmem << #16) , Ymem = HI(ACy << T2) [, T3 = Xmem]
- ACy = (Xmem << #16) − ACx , Ymem = HI(ACy << T2) [, T3 = Xmem]
- ACy = Xmem << #16 , Ymem = HI(ACx << T2) [, T3 = Xmem]

An overflow occurs during the shift operation, but is not detected within the 40 bits of the shifted result. This occurs only in the C54x Compatibility Mode.

Example

```
C54x : *AR7 = hi(A) << 12 (STH A, 12, *AR7)
       SST: 1
       SXM: 1
     (before)
       A                  :   00.F000.0000 (40bits)
       Data Memory 0321h  :            ABCD
     (after)
       A                  :   00.F000.0000
     (shift result)       : F.00.0000.0000
       Data Memory 0321h  :            0000 (Expects no overflow is detected)


C55x : *AR7 = HI(AC0 << 12)
       C54CM : 1
       SST   : 1
       SXMD  : 1
     (before)
       AC0                :   00.F000.0000 (40bits)
       Data Memory 0321h  :            ABCD
     (after)  :
       AC0                :   00.F000.0000
     (shift result)       : F.00.0000.0000
       Data Memory 0321h  :            7FFF (An overflow is detected.)
```

Assembler Notification: Pending

Workaround: None

Advisory CPU_107 *Conditional Call With False Condition Corrupts RETA*

Revision(s) Affected: See Table 1

Details: When using the following sequence, RETA gets corrupted:

1. Memory write instruction with latency.
2. Zero or one (pair of) instructions of any kind that does not produce PC discontinuity.
3. Conditional call with false condition.
4. Any one of the following instructions:
   - call L16/P24/ACx
   - if (cond) call L16/P24
   - intr(k5) (and Hardware interrupt as well)
   - trap(k5)

Example

```
AR0  =  #0
AR1  =  #external_memory
*AR1 =  #0                      ; Memory write with latency
if(AR0 != #0) call subroutine  ; No call is performed and RETA gets corrupted
call another_subroutine
```

Assembler Notification: The assembler (versions 2.7 and greater) will generate a remark when a conditional call is preceded by a memory write.

Workaround: Do one of the following:

- Use slow return mode, which does not utilize RETA
- Put at least two instructions between a memory write instruction and a conditional call.

Advisory CPU_108 — *Long (32-Bit) Read From MMR Gets Corrupted*

Revision(s) Affected: See Table 1

Details: When "Lmem" (see instruction list below) points to address "0x4A" or "0x4D", two memory-mapped registers (MMR)—IVPH and ST2 for "0x4A"; SP and SSP for "0x4D"—are supposed to be read. However, a corrupted value is read from ST2 (for the case of "0x4A") and SSP (for the case of "0x4D").

NOTE: The following shows the corresponding part of the MMR mapping.

Address	Register
4A	IVPH
4B	ST2
4C	SSP
4D	SP

Example

```
@#0x4a   =    #0xaaaa  ||   mmap()   ; IVPH <= 0xaaaa
@#0x4b   =    #0x1111  ||   mmap()   ; ST2  <= 0x1111
push(dbl(@#0x4a))      ||   mmap()   ; @SP-1 <= ST2, @SP-2 <= IVPH
```

The stack, pointed by SP−1, should be updated with ST2, but instead, gets a corrupted value.

The following is the entire list of Lmem instructions:

- push(dbl(Lmem))
- dbl(coeff) = Lmem
- ACy = ACx + dbl(Lmem)
- ACy = ACx − dbl(Lmem)
- ACy = dbl(Lmem) − ACx
- RETA = dbl(Lmem)
- ACx = M40(dbl(Lmem))
- pair(HI(ACx)) = Lmem
- pair(LO(ACx)) = Lmem
- pair(TAx) = Lmem
- dst = dbl(Lmem)
- HI(ACy) = HI(Lmem) + HI(ACx) , LO(ACy) = LO(Lmem) + LO(ACx)

Long (32-Bit) Read From MMR Gets Corrupted (Continued)

- HI(ACy) = HI(ACx) − HI(Lmem) , LO(ACy) = LO(ACx) − LO(Lmem)
- HI(ACy) = HI(Lmem) − HI(ACx) , LO(ACy) = LO(Lmem) − LO(ACx)
- HI(ACx) = Tx − HI(Lmem) , LO(ACx) = Tx − LO(Lmem)
- HI(ACx) = HI(Lmem) + Tx , LO(ACx) = LO(Lmem) + Tx
- HI(ACx) = HI(Lmem) − Tx , LO(ACx) = LO(Lmem) − Tx
- HI(ACx) = HI(Lmem) + Tx , LO(ACx) = LO(Lmem) − Tx
- HI(ACx) = HI(Lmem) − Tx , LO(ACx) = LO(Lmem) + Tx

Assembler Notification: Pending

Workaround: Use "4B" instead of "4A", "4C" instead of "4D". For example,

```
@#0x4a   =    #0xaaaa  ||   mmap()   ; IVPH <= 0xaaaa
@#0x4b   =    #0x1111  ||   mmap()   ; ST2  <= 0x1111
push(dbl(@#0x4b))      ||   mmap()   ; @SP-1 <= IVPH, @SP-2 <= ST2
```

Advisory CPU_109 — *Bus Error Issued on Byte Access to I/O Space With Address Range 0x0 to 0x5f*

Revision(s) Affected: See Table 1

Details: A bus error, which is captured on IFR1 bit 8 as "BERRINTF" and ST3 bit 7 as "CBERR", is issued when an illegal bus access occurs. All I/O space (0x0 to 0xffff) is byte read/write-accessible without any bus errors. However, a bus error is wrongly issued when a byte access is made to the I/O space with address range 0x0 to 0x5f.

The following is the entire list of byte-accessible instructions that will give bus errors when trying to read I/O space address 0x0 to 0x5f:

- dst = uns(high_byte(Smem))
- dst = uns(low_byte(Smem))
- ACx = low_byte(Smem) << SHIFTW
- ACx = high_byte(Smem) << SHIFTW
- high_byte(Smem) = src
- low_byte(Smem) = src

Assembler Notification: Pending

Bus Error Issued on Byte Access to I/O Space With Address Range 0x0 to 0x5f (Continued)

Workaround: Do one of the following:

- Do not use any of the above instructions to access the I/O space, address 0x0 to 0x5f
- Ignore the bus error, by either not setting IER1[8] or by not using ST3[7].

Advisory CPU_110

Relative Branch in ISR Corrupts Program Flow When Localrepeat With C54CM = 1 is Interrupted

Revision(s) Affected: See Table 1

Details: When in the C54x Compatibility Mode (C54CM = 1), if an interrupt is asserted during a local repeat in which the interrupt service routine contains a relative branch (any from the list below) whose offset is >64 bytes forward or backward, then the program flow is corrupted.

- goto L16
- call L16
- if (cond) goto L8
- if (cond) goto L16
- if (cond) call L16
- compare (uns(src RELOP K8)) goto L8
- if (ARn[mod] != #0) goto L16

Assembler Notification: Pending

Workaround: Do one of the following:

- Reset the BRAF bit (ST1[15]) in the ISR which contains the relative branch at least 5 instruction cycles prior to the relative branch. See example below.

  Example

```
ISR:   bit(ST1, #15) = #0
       nop
       nop
       nop
       nop
       call L16
       .
       .
```

NOTE: Because the BRAF bit is automatically saved before executing an ISR, and restored at the return from interrupt, the localrepeat can be restarted without any problems after returning from interrupt even if the BRAF bit is reset in the ISR.

- Avoid use of localrepeat under C54CM = 1.

Advisory CPU_111 *C54CM Bit Modification Followed by a mar Instruction Not Pipeline-Protected*

Revision(s) Affected: See Table 1

Details: When the destination register of the following mar instructions is AR4, AR5, AR6, or AR7, and the circular-addressing mode is set by either the corresponding bit in ST2 or use of circular( ) qualifier, the C54CM bit determines the circular buffer size register as BK47 when C54CM = 0 or BK03 when C54CM = 1.

- mar(TAy + TAx)
- mar(TAy − TAx)
- mar(TAx + P8)
- mar(TAx − P8)

Therefore, any instruction to modify the C54CM bit in the pipeline prior to the mar instruction must be completed before the mar instruction is performed at its ADDRESS phase. However, because a pipeline-protection mechanism is missing for this case, the C54CM bit modification may not be completed in time under the following condition.

- "bit(ST1, @C54CM) = #0/1" or "MMR write access to ST1 to modify C54CM bit"
- < less than or equal 4 instruction slots >
- any of above mar instruction to modify AR4−7 in circular mode

In the above case, the wrong circular buffer size register will be used.

Assembler Notification: Pending

Workaround: Make sure enough instruction slots are inserted between the two events as follows:

- "bit(ST1, @C54CM) = #0/1" or "MMR write access to ST1 to modify C54CM bit"
- < less than or equal 5 instruction slots >
- any of above mar instruction to modify AR4−7 in circular mode

Example:

```
bit(ST1, @C54CM) = #0
nop
nop
nop
nop
nop
mar(AR7 + AR0) || circular( )
```

Advisory CPU_112

Data Page Register and Stack Pointer Update Not Pipeline-Protected Against Data Move Instructions

Revision(s) Affected: See Table 1

Details: The following registers are used for address generation in direct-addressing mode:

- XDP[22:0] (DPH[6:0] and DP[15:0]) when the CPL bit (ST1[14]) is 0. (Note that DP[15:7] is mapped to ST0[8:0].)
- XSP[22:0] (SPH[6:0] and SP[15:0]) when the CPL bit (ST1[14]) is 1.

Any update of these registers followed by a direct-addressing access should be completed by the address generation unit. However, because of missing pipeline protection, under the following conditions, the update of the register will not be reflected for the address generation.

Condition 1

Instruction to update the Data Page register or Stack Pointer in EXECUTE phase. See Instruction List below.

- less than 4 instruction cycles
- "Smem = coeff || readport( )" or "coeff = Smem || writeport( )" (Smem is in direct-addressing mode.)

Instruction List

```
When CPL = 0,
  Xdst   =   Xsrc          (Xdst is XDP)
  Xdst   =   popboth( )    (Xdst is XDP)
  XAdst  =   dbl(Lmem)     (XAdst is XSP)
  DP     =   Smem
  DPH    =   Smem
  bit(ST0,#k4) = #0/1      (k4 is 8,7,...,1,0)
When CPL = 1,
  Xdst   =   Xsrc          (Xdst is XSP or XSSP)
  Xdst   =   popboth( )    (Xdst is XSP or XSSP)
  XAdst  =   dbl(Lmem)     (XAdst is XSP or XSSP)
  SP     =   Smem
  SP     =   TAx
  SP     =   SP + K8       (Applicable to only Rev1.0 as it is performed
                           in EXECUTE phase instead of ADDRESS phase.)
```

Condition 2

Instruction with MMR write to DPH (0x2B)/DP (0x2E)/ST0_55 (0x02)/ST0 (0x06) with CPL = 0 or SPH (0x4E)/SP (0x18, 0x4D) with CPL = 1 in WRITE phase.

- less than 5 instruction cycles
- "Smem = coeff || readport( )" or "coeff = Smem || writeport( )" (Smem is in direct-addressing mode.)

Assembler Notification: Pending

Data Page Register and Stack Pointer Update Not Pipeline-Protected Against Data Move Instructions (Continued)

Workaround: Make sure enough instruction slots are inserted between the two events as follows:

- More than or equal 4 instruction slots for Condition 1
- More than or equal 5 instruction slots for Condition 2

Example:

```
XDP = AC0
nop
nop
nop
nop
@#0xa = coef(*CDP) || readport( )
```

Advisory CPU_114

ST2 Update and Dual-Memory Access With Circular Qualifier Not Pipeline-Protected

Revision(s) Affected: See Table 1

Details: When the C54CM bit is set to 1 for the following addressing modes for dual-memory-access instructions:

```
*ARn
*ARn+
*ARn–
*ARn(AR0)
```

the circular qualifier should not have an effect on operation, and ST2[n] should define the addressing mode (linear or circular) for compatibility with C54x devices. However, when ST2[n] is updated just prior to accessing the dual-memory instructions with the circular qualifier, the update is not reflected due to missing pipeline protection, as shown in the example below.

Example:

```
bit(ST2,#0) = 0
        .
        .
        .
bit(ST2,#0) = 1
nop
*AR0+ = *+AR1 || circular()         ; AR0 post increment is expected to be
                                    ; in circular mode, not linear mode.
```

Assembler Notification: Pending

ST2 Update and Dual-Memory Access With Circular Qualifier Not Pipeline-Protected (Continued)

Workaround: Make sure enough instruction slots are inserted between the two events, as shown below:

- Put more than four instructions between instructions to update ST2 and dual-memory accesses with the circular qualifier.
- Put more than five instructions between MMR write accesses to ST2 and dual-memory accesses with the circular qualifier.

Example:

```
@ST2 = #0x1 || mmap()
nop
nop
nop
nop
nop
nop
*AR0+ = *+AR1 || circular()
```

Advisory CPU_116 — *Interrupted Nesting of Loops May Stop CPU Execution*

Revision(s) Affected: See Table 1

Details: When all four of the following conditions occur, the CPU will stop execution.

- A localrepeat is used for the inner loop within nested loops.
- Any of the following conditions is met:
  - The first single or pair of instructions located at the top of the inner loop is less than 4 bytes.
  - The size difference between the outer and inner loop (see NOTE) is less than or equal to 32 bytes, and BRC1 is changed from zero to non-zero values during the last iteration.
- An interrupt occurs anywhere within the inner loop at any iteration.
- After returning from an interrupt, a specific P-request is stalled with more than two latency cycles.

Assembler Notification: Pending

Interrupted Nesting of Loops May Stop CPU Execution (Continued)

Workaround: Use one of the following workarounds:

- Do not use local repeats for inner loops.
- Make sure the first instruction of the inner loop is more than or equal to 4 bytes, and also one of the following:
  - The outer loop is at least 33 bytes larger than the inner loop.
  - Do not change BRC1 from zero, which means at the last iteration, to non-zero value in the inner loop.

Advisory CPU_117

Updating BRC Prior to a Loop That Contains Only Single Repeats Incorrectly Decrements the RPTC

Revision(s) Affected: See Table 1

Details: When a loop (blockrepeat or localrepeat) contains the following:

- only the single repeat instruction (can be in parallel with another instruction) and an instruction to be single-repeated
- the corresponding BRC (BRC0 in case of outer loop, BRC1 in case of inner loop) is updated prior to the loop

the RPTC for the single repeat is decremented incorrectly. The single repeat instruction is executed fewer times (depending on stall conditions) than expected.

The following examples are possible BRC update cases. Note that the instruction "BRCx = k12" is not a problem.

Case 1: BRC update with MMR access

Example:

```
@BRC0 = something || mmap()
[ 0~3 instructions ]
blockrepeat{
       single repeat instruction (see NOTE)
       instruction1 || instruction2    ; to be single-repeated
}
```

Updating BRC Prior to a Loop That Contains Only Single Repeats Incorrectly Decrements the RPTC (Continued)

Case 2: BRC update with "BRCx = TAx" or "BRCx = Smem"

Example:

```
BRC0 = T0
[ 0~2 instructions ]
localrepeat{
       something || single repeat instruction (see NOTE)
       instruction                      ; to be single-repeated
}
```

NOTE: A single instruction could be one of the following:

```
repeat(k8)
repeat(k16)
repeat(CSR)
repeat(CSR) , CSR += TAx
repeat(CSR) , CSR += k4
repeat(CSR) , CSR -= k4
while (cond && (RPTC < k8)) repeat
```

Assembler Notification: Pending

Workaround: Place at least four instructions between BRC update and the loop for Case 1, or three instructions between BRC update and the loop for Case 2.

Advisory CPU_118	*CPU May Halt After Returning From an Interrupt Service Routine When Operating in Emulation (Debug) Mode*

Revision(s) Affected: See Table 1

Details:

When an interrupt is generated, the CPU automatically saves some context registers onto the stacks before executing the interrupt service routine. The registers saved to the stack are: the return address (PC); the loop context bits; the status registers 0, 1, and 2; and the debug status register (DBSTAT). DBSTAT is a DSP register that holds debug context information used during emulation. The return address is the address of the instruction to be executed when the CPU returns from the interrupt service routine. The loop context bits are a record of the type and status of repeat loops that were active when the interrupt occurred.

After the automatic context save, the CPU then executes the interrupt service routine, which may save some additional context before servicing the interrupt. At the end of the interrupt service routine, the return-from-interrupt instruction (RETI) is used to branch back to the interrupted program. The CPU is forced to restore the automatically saved context registers when the RETI instruction is executed.

If a memory stall is generated during the automatic context restore, an invalid value could be loaded into the DBSTAT register when the RETI instruction is executed. This value could cause the CPU to go into an emulation halt mode. This emulation halt mode is similar to what debug software (such as Code Composer Studio) uses to halt the processor during break points and halt commands. While in emulation halt mode, the CPU will not execute any instructions and will not service any interrupts.

NOTE: Only the DBSTAT register is affected by stalls during the automatic context restore. The return address and the loop context bits are not affected by this error.

NOTE: Emulation halts are ignored by the CPU when it is not operating in emulation mode; therefore, this error will not impact the device operation when the device is operated in functional mode (i.e., when the debugger is not being used).

More information on stacks, interrupts, and the automatic steps executed by the CPU during interrupts can be found in the *TMS320C55x DSP CPU Reference Guide* (literature number SPRU371).

Assembler Notification: N/A

Workaround:

The chances of encountering this error can be minimized by decreasing the chances of generating stalls during the automatic context restore.

To minimize the chances of generating stalls, follow one of these steps:

1. Allocate the data stack and the system stack in DARAM.
2. Allocate the data stack and system stack in separate SARAM blocks.

The location of the data stack is specified through the XSP register while the system stack location is specified through the XSSP register.

Advisory CPU_119

Due to Improper Update of the DBSTAT Register, the Debugger May Halt at Code Locations Where No Breakpoints are Set

Revision(s) Affected: See Table 1

Details:

The error described here causes the Code Composer Studio (CCS) debugger to halt at code locations where no breakpoint has been set by the user. This error can be reproduced by following these steps using CCS.

1. Load and execute a program.
2. Place a software breakpoint in non-ISR code (setting the breakpoint in the ISR could also show the problem, but it is more likely to be seen when breakpoint is in non-ISR code).
3. After the CPU stops at the software breakpoint, clear it.
4. Generate an interrupt to the CPU.
5. Start the CPU again without stepping from the breakpoint location.
6. Refresh a memory, register or watch window (executing a GEL script may also trigger the problem).
7. The CPU halts at an arbitrary location.

NOTE: This bug is not seen when the user runs or steps from a breakpoint and leaves the breakpoint in place. Only the sequence listed above will cause the problem.

This error is caused by the debug status register (DBSTAT) being incorrectly pushed onto the stacks before its breakpoint status bits are cleared. DBSTAT is a DSP register that holds debug context information used during emulation. Among other things, it contains several bits which indicate when the CPU is halted as well as the reason for the halt.

The CPU automatically saves DBSTAT to the stacks along with other context registers before executing an interrupt service routine (ISR) [see the *TMS320C55x DSP CPU Reference Guide* (literature number SPRU371)]. At the end of the ISR, the CPU is forced to restore the automatically saved context registers when the RETI instruction is executed.

When an interrupt is generated while the CPU is halted due to a software breakpoint, the software breakpoint bits should be cleared by a command from the debug software before DBSTAT is pushed onto the stack. Instead, the contents of DBSTAT are incorrectly pushed onto the stack before these bits are cleared. At the end of the ISR, DBSTAT is automatically restored from the stack, incorrectly restoring the previous setting of the software breakpoint bits.

When a debug operation such a memory window refresh is executed, CCS commands the CPU to halt (as it should) and, after the checking DBSTAT to ensure its command was successful, it believes the CPU is already halted due to a software breakpoint. Since CCS is not be able associate the breakpoint with one of its own, it simply informs the user the CPU is halted at the current program counter location.

NOTE: This error does not corrupt the state of the CPU. It only causes an incorrect status of the emulation state, which causes problems for debug tools such as CCS.

Due to Improper Update of the DBSTAT Register, the Debugger May Halt at Code Locations Where No Breakpoints are Set (Continued)

Assembler Notification: N/A

Workaround: If this error is encountered, single-step a few assembly or C instructions before starting the CPU again after a software breakpoint. Interrupts are blocked from the CPU during single-stepping, allowing enough time for the software breakpoint flags to be cleared from DBSTAT.

For example, follow these steps using CCS:

1. Load and execute a program.
2. Place a software breakpoint in non-ISR code.
3. After the CPU stops at the software breakpoint, clear it.
4. Generate an interrupt to the CPU.
5. Single-step a couple of assembly or C instructions.
6. Start the CPU again.

4 Documentation Support

For device-specific data sheets and related documentation, visit the TI web site at: **http://www.ti.com**

For additional information, see the latest versions of:

- *TMS320VC5501 Fixed-Point Digital Signal Processor* data manual (literature number SPRS206)
- *TMS320VC5502 Fixed-Point Digital Signal Processor* data manual (literature number SPRS166)
- *TMS320VC5503 Fixed-Point Digital Signal Processor* data manual (literature number SPRS245)
- *TMS320VC5507 Fixed-Point Digital Signal Processor* data manual (literature number SPRS244)
- *TMS320VC5509A Fixed-Point Digital Signal Processor* data manual (literature number SPRS205)
- *TMS320VC5510 Fixed-Point Digital Signal Processor* data manual (literature number SPRS076)
- *OMAP5910 Dual-Core Processor* data manual (literature number SPRS197)
- *OMAP5912 Applications Processor* data manual (literature number SPRS231)
- *TMS320C55x™ DSP Functional Overview* (literature number SPRU312)
- *TMS320C55x DSP CPU Reference Guide* (literature number SPRU371)
- *TMS320C55x DSP Mnemonic Instruction Set Reference Guide* (literature number SPRU374)
- *TMS320C55x DSP Algebraic Instruction Set Reference Guide* (literature number SPRU375)
- *TMS320C55x DSP Peripherals Overview Reference Guide* (literature number SPRU317)
- *OMAP5910 Dual-Core Processor Silicon Errata* (literature number SPRZ016)

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products		**Applications**	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DSP	dsp.ti.com	Broadband	www.ti.com/broadband
Interface	interface.ti.com	Digital Control	www.ti.com/digitalcontrol
Logic	logic.ti.com	Military	www.ti.com/military
Power Mgmt	power.ti.com	Optical Networking	www.ti.com/opticalnetwork
Microcontrollers	microcontroller.ti.com	Security	www.ti.com/security
		Telephony	www.ti.com/telephony
		Video & Imaging	www.ti.com/video
		Wireless	www.ti.com/wireless

Mailing Address: Texas Instruments
Post Office Box 655303 Dallas, Texas 75265