TMS320C6713, TMS320C6713B

Digital Signal Processors

Silicon Errata

C6713 Silicon Revision 1.1

C6713B Silicon Revision 2.0

SPRZ191J
December 2002
Revised August 2005

TEXAS INSTRUMENTS

REVISION HISTORY

This errata revision history highlights the technical changes made to the SPRZ191I device-specific errata to make it an SPRZ191J revision.

Scope: Applicable updates to the C67x device family, specifically relating to the C6713/13B devices, have been incorporated.

PAGE(S) NO.	ADDITIONS/CHANGES/DELETIONS
5	Introduction, Revision Identification section: Updated Figure 1, Example, Lot Trace Codes for TMS320C6713 (GDP and PYP Packages) and TMX320C6713B (GDP and PYP Packages)
16	Table 2, IPU/IPD Condition of JTAG, Clock/PLL Oscillator, McBSP0, McBSP1, and Timer1 if DEVCFG.0 = 1 Clock/PLL Oscillator Pins section: In CLKOUT3, changed “***IPU***” to “***IPD***”
16	C6713B Silicon Revision 2.0 Known Design Exceptions to Functional Specifications and Usage Notes: Added “EMIF Big Endian Mode Correctness (HD12 = 0) is Not Usable Unless All CE Spaces Have the Same Width”

Contents

1 Introduction

This document describes the known exceptions to the functional specifications for the TMS320C6713 and TMS320C6713B digital signal processors. [See the *TMS320C6713, TMS320C6713B Floating-Point Digital Signal Processors* data sheet (literature number SPRS186).] These exceptions are applicable to the TMS320C6713 and TMS320C6713B devices (272-pin Ball Grid Array, GDP suffix; and the 208-pin PowerPAD™ Plastic Quad Flatpack, PYP suffix).

For additional information, see the latest version of *TMS320C6000 DSP Peripherals Overview Reference Guide* (literature number SPRU190).

The advisory numbers in this document are not sequential. Some advisories have been *moved* to the next silicon revision and others have been removed and documented in a user's guide and/or data sheet. When advisories are moved or deleted, the remaining advisory numbers remain the same and are not resequenced.

1.1 Quality and Reliability Conditions

TMX Definition

Texas Instruments (TI) does not warranty either (1) electrical performance to specification, or (2) product reliability for products classified as "TMX." By definition, the product has not completed data sheet verification or reliability performance qualification according to TI Quality Systems Specifications.

The mere fact that a "TMX" device was tested over a particular temperature range and voltage range should not, in any way, be construed as a warranty of performance.

TMP Definition

TI does not warranty product reliability for products classified as "TMP." By definition, the product has not completed reliability performance qualification according to TI Quality Systems Specifications; however, products are tested to a published electrical and mechanical specification.

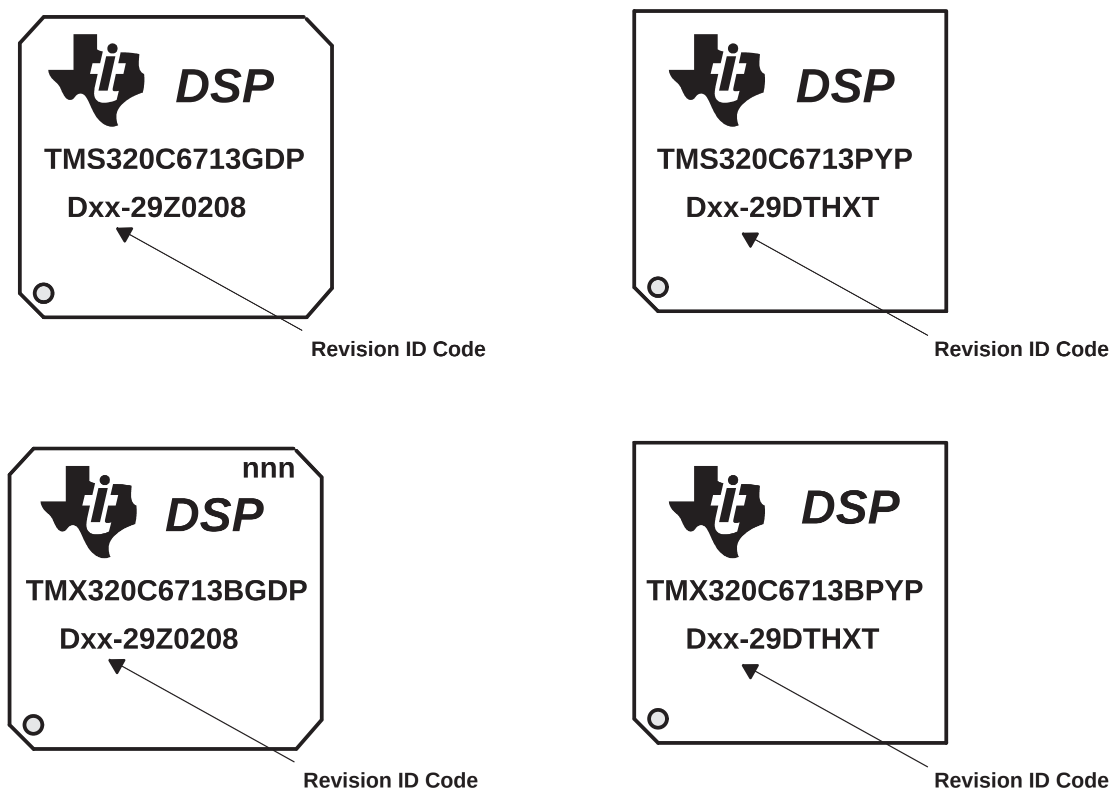
TMS Definition

Fully-qualified production device

PowerPAD is a trademark of Texas Instruments.
Other trademarks are the property of their respective owners.

1.2 Revision Identification

The device revision can be determined by the lot trace code marked on the top of the package. The location of the lot trace codes for the GDP and PYP packages are shown in Figure 1 and the revision ID codes are listed in Table 1. The revision ID described here is not to be confused with the CPU revision ID that is in the Control Status Register.



“nnn” represents the device speed. For example:

300	=	300 MHz Core [13**B**GDP]
(Blank)	=	225 MHz Core [13GDP, 13**B**GDP and 13**B**PYP]
(Blank)	=	200 MHz Core [13GDP**A**, 13**B**PYP, 13PYP, 13**B**GDP**A** and 13**B**PYP**A** (A=Extended Temperature)]
(Blank)	=	167 MHz Core [13PYP**A**, and 13**B**PYP**A** (A=Extended Temperature)]

NOTE: Qualified devices are marked with the letters “TMS” at the beginning of the device name, while nonqualified devices are marked with the letters “TMX” or “TMP” at the beginning of the device name.

Figure 1. Example, Lot Trace Codes for TMS320C6713 (GDP and PYP Packages) and TMX320C6713B (GDP and PYP Packages)

Silicon revision is identified by a code on the chip. The code is of the format Dxx-YMZLLLS (GDP) and of the format Dxx-YMDLLLS (PYP). If xx is 11, then the silicon is revision 1.1. If xx is 20, then the silicon revision is 2.0.

Table 1. Revision ID Codes

Revision ID Code	Silicon Revision	Comments
11	1.1	TMX320C6713, TMS320C6713
20	2.0	TMS320C6713B

2 C6713B Silicon Revision 2.0 Known Design Exceptions to Functional Specifications and Usage Notes

2.1 Usage Notes for C6713B Silicon Revision 2.0

Usage Notes highlight and describe particular situations where the device's behavior may not match presumed or documented behavior. This may include behaviors that affect device performance or functional correctness. These notes will be incorporated into future documentation updates for the device (such as the device-specific data sheet), and the behaviors they describe will not be altered in future silicon revisions.

EMIF: L2 Cache Operations Block Other EDMA Operations to EMIF (C671x/C621x Devices: All Silicon Revisions)

When using the L2 cache on the C671x/C621x devices, for a given EMIF-to-CPU frequency ratio, an L2 writeback or L2 writeback-invalidate operation may block other EDMA requests from accessing the EMIF until the operation completes. If the other EDMA requests to the EMIF have hard real-time deadlines, these deadlines may be missed if the deadline is shorter than the time required to complete the L2 writeback operation. The McBSP and McASP peripherals are most sensitive to this issue, as the buffering local to the McBSP/McASP peripheral can only hold data for at most one sample at a time before data loss occurs.

On the C671x/C621x devices, all cache requests to EMIF address ranges are serviced on the highest priority level of the EDMA (priority 0). All programmed EDMA or QDMA transfers (e.g., EDMA transfers to service the McBSP or paging data from EMIF to/from L2) and peripheral-initiated transfers (such as HPI) are limited to using priority 1 or priority 2 queues of the EDMA; therefore an L2 writeback or L2 writeback-invalidate operation may block the lower priority request.

Program-initiated cache coherency operations (such as L2 writeback and L2 writeback-invalidate operations) are submitted to the EDMA as a long string of cache operations. For block-based writeback commands, the maximum length of the cache writeback operation is under user control via the programmed address range. The length of the range writeback directly impacts the amount of time that the cache traffic may block other accesses to EMIF. The total potential block-out time equals the amount of time for the cache transfer and is calculated as follows:

Cache transfer size * EMIF clock cycle time = Total potential block-out time

For example, if the user performs an L2 writeback operation to external memory for 2048 words with a 100-MHz EMIF, the external EMIF bus may be blocked for: 2048 words * 10 ns $\cong$ 20 µs.

Global cache operations (such as L2 writeback-all or L2 writeback-invalidate-all) are also submitted to the EDMA as a long string of cache operations. However, the length of the global cache operation is not controllable by the user and can be as long as the depth of the L2 cache size (up to 64 Kbytes). If the user performs an L2 writeback-all operation to external memory using a 100-MHz EMIF, and L2 is set to the maximum cache size, the external EMIF bus may be blocked for 16 384 words * 10 ns ≅ 160 µs. Since this can block the EMIF for long periods of time, the user should avoid using global cache operations at the same time as real-time data transfers. In general, this is not a limiting factor since global cache operations are primarily performed during system initialization, task switches, or other non real-time code segments.

As the sample rate is system-dependent, the user must calculate the time between serial samples to determine the best approach to avoid data loss. The user may break large cache operations into smaller blocks, and transmit each of these blocks using the CACHE_wbInvL2() and CACHE_wbL2() CSL functions. By breaking the large cache operations into smaller blocks, other peripherals are then allowed to access the EDMA.

If the EMIF frequency is more than half of the CPU frequency, the device is able to service the L2 writeback requests faster than the requests can be issued, leaving some EMIF bandwidth available to service other EDMA requests, so the block-out problem is less noticeable. Therefore, breaking down cache operations into smaller blocks is more critical when the EMIF frequency is less than half of the CPU frequency. Figure 2 shows the minimum required latency between McBSP transfers to EMIF at 200-MHz CPU and 50-MHz EMIF when breaking down the cache operations. These McBSP transfers were performed with concurrent cache operations to EMIF, creating a block-out scenario. With the 1024-word cache writeback-invalidate operation broken into 32-word blocks, the McBSP is able to perform almost 10 times faster. The performance improvement is similar when breaking down the writeback-only operation.

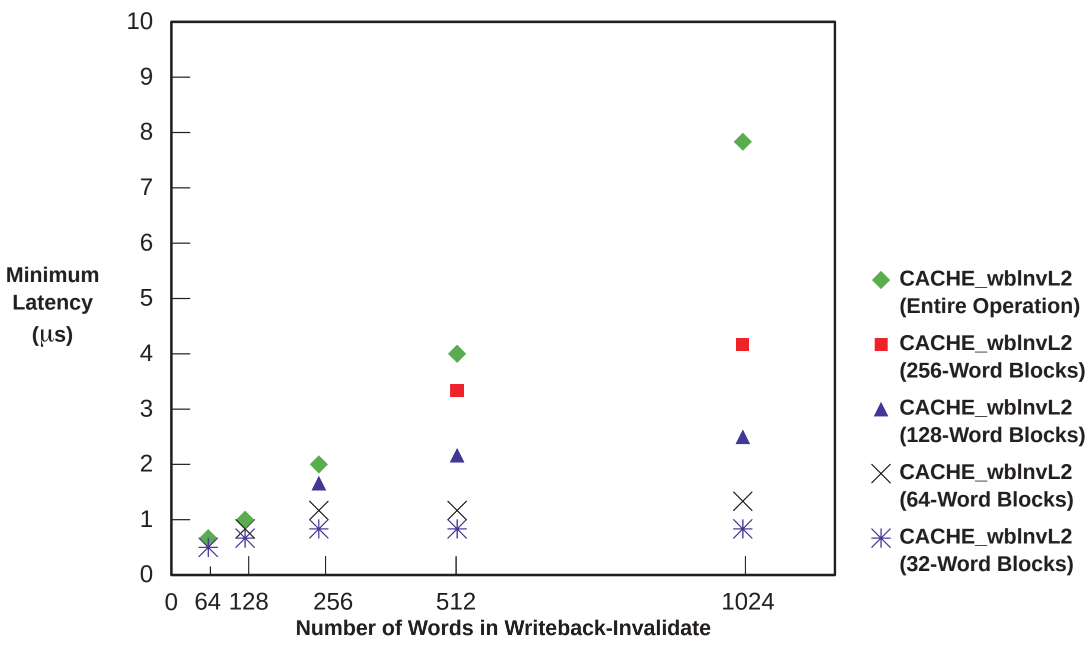


Figure 2. Minimum Required Latency Between McBSP Events for a Successful Transfer with Concurrent L2 Writeback-Invalidates at 200-MHz CPU and 50-MHz EMIF, Using Entire Operations and Block Breakdown

For example, if the CPU is running at 200 MHz with a 50-MHz EMIF and you have a McBSP hard real-time deadline of 5 µs, Figure 2 shows that a 1024-word L2 writeback-invalidate may cause data loss since back-to-back McBSP events can only be serviced at ~8 µs. By breaking down the L2 writeback-invalidate into 256-word blocks, you can then meet the 5-µs McBSP deadline. In other words, when performing a 1024-word L2 writeback-invalidate operation with the CPU and EMIF conditions cited above, the McBSP events can be serviced in ~8 µs for the entire operation (one whole block), in ~4 µs when breaking it into 256-word blocks, in ~2.5 µs when breaking it into 128-word blocks, etc.

When the CPU is set to 225 MHz and the EMIF is set to 100 MHz, breaking down the cache operations will still improve the block-out problem. Figure 3 shows the improvement in the McBSP's performance with this frequency ratio.

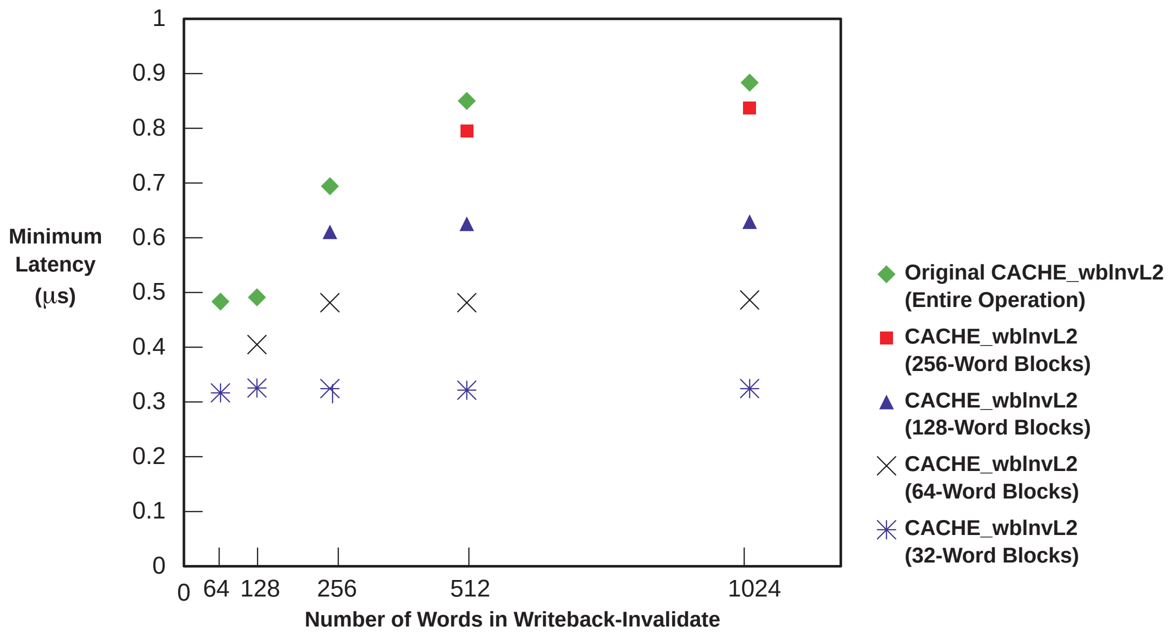


Figure 3. Minimum Required Latency Between McBSP Events for a Successful Transfer With Concurrent Writeback-Invalidates at 225-MHz CPU and 100-MHz EMIF, Using Entire Operations and Block Breakdown

When the CPU is set to 150 MHz and the EMIF is set to 100 MHz, there is virtually no benefit from breaking down the coherency cache operations. Figure 4 shows the McBSP's performance with this frequency ratio.

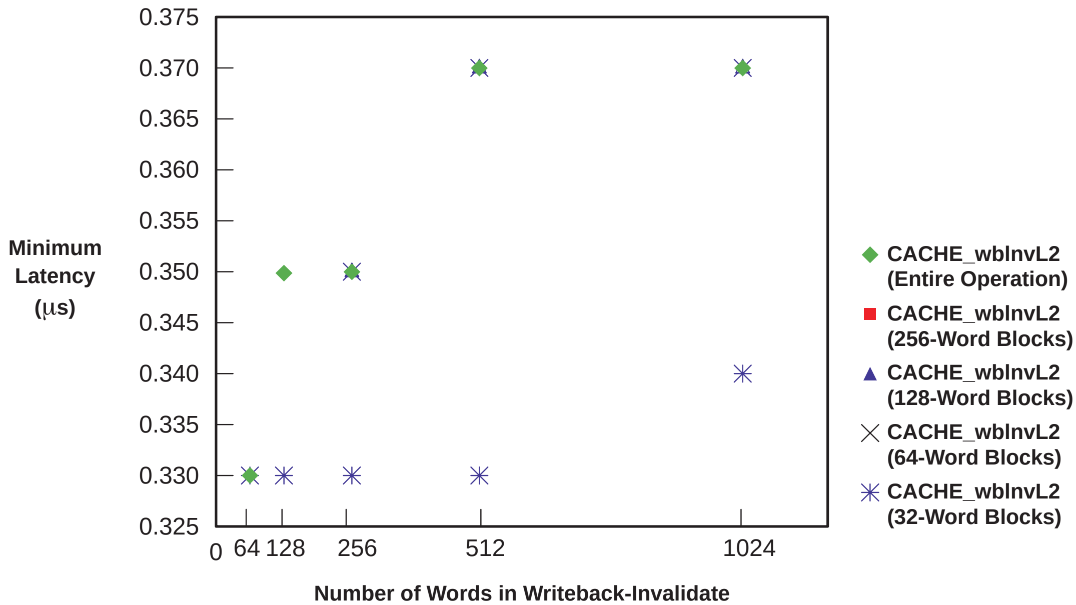


Figure 4. Minimum Required Latency Between McBSP Events for a Successful Transfer With Concurrent Writeback-Invalidates at 150-MHz CPU and 100-MHz EMIF, Using Entire Operations and Block Breakdown

Breaking down the cache operations into smaller blocks takes longer to complete than performing the entire cache function as one large block. Figure 5 shows how much extra overhead is incurred by breaking down an L2 writeback-invalidate operation to transfer 1024 words with different sized blocks and at various frequency ratios. Notice that for the 200-MHz CPU and the 50-MHz EMIF frequency ratio, where the new functions are most critical for peripherals such as the McBSP, the least overhead is incurred.

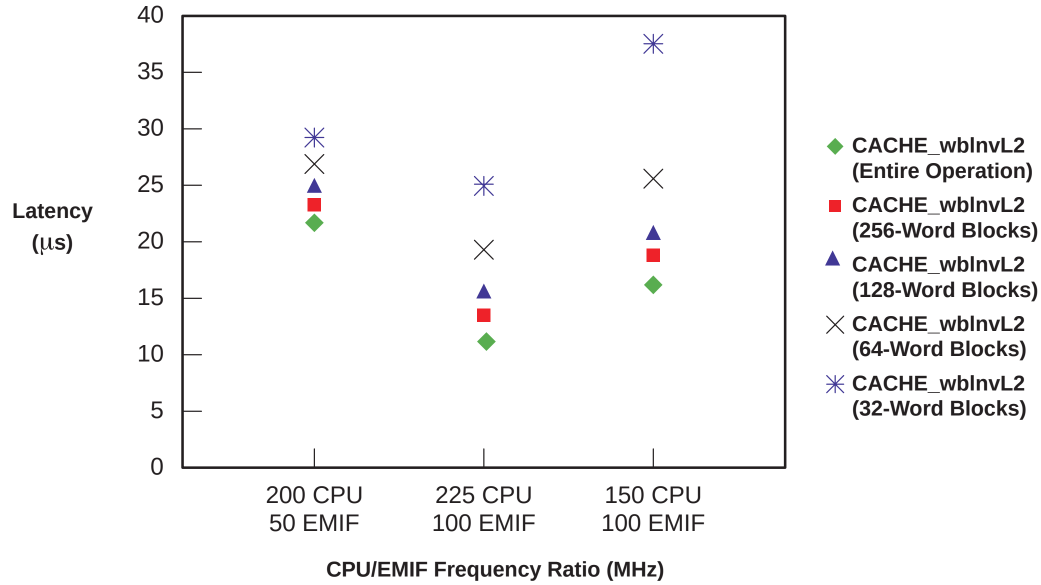


Figure 5. 1024-Word L2 Writeback-Invalidate Performance at Various Frequency Ratios With Old and New CACHE-wbInvL2()

To avoid cache operations blocking other time-sensitive EDMA accesses, observe the following guidelines:

1. Avoid placing real-time data in EMIF address range. Instead, real-time data should be placed in the L2 address range.
2. If data must be placed in the EMIF address range:
   - Avoid global cache operations in favor of block-based cache operations.
   - Block-based cache operations should be submitted in small blocks, such that the total amount of time that the EMIF is blocked is less than the amount of time between serial samples.

HPI: Illegal Memory Access Can Result in Unexpected HPI Behavior

On C6713B silicon revision 2.0 and C6713 silicon revision 1.1, the DSP has a reserved memory range that is mapped to the internal FIFO of the HPI for EDMA engine usage. This reserved memory range is located at 0x60000000 − 0x7FFFFFFF in the memory map. If CPU code (or a host access) happens to read/write from/to this memory range, the internal HPI state machine can be corrupted, causing one or more of the following occurrences:

- Host reads/writes through the HPI fail. HPI reads return incorrect data, and/or HPI writes result in incorrect data being written.
- Host reads/writes through the HPI take an unexpectedly long time. The $\overline{\text{HRDY}}$ signal stays high (not ready) for an extended period of time.
- HPI locks up. $\overline{\text{HRDY}}$ stays high indefinitely.

The most common cause of this illegal access is uninitialized or stray pointers. To verify that the DSP program does not perform this illegal memory access, the user can use the Advanced Event Triggering tools featured in Code Composer Studio™ Integrated Development Environment (IDE) version 2.1 or later, with the latest emulation driver. Below are the step-by-step instructions on how to trap a CPU access to the memory range 0x60000000 − 0x7FFFFFFF:

1. Start Code Composer Studio™ IDE with the proper setup and GEL file.
2. Load the program.
3. Under the *Tools* menu, select Advanced Event Triggering > Event Analysis
4. Right-click on the bottom left panel that appears, select “Set Hardware Watchpoint”.
5 Name the watchpoint; choose to watch for “Data Memory Reads” or “Data Memory Writes”; select the inclusive range start address (0x60000000) and end address (0x7FFFFFFF); select the data size from 32-, 16-, or 8-bit to watch for word, halfword, or byte reads/writes, respectively. Then, click Apply.
6 The watchpoint now is enabled, indicated by the blue “E” icon. Now, run the program.
7. When a read/write to the specified memory range is detected, the CPU halts, and the blue “E” icon changes to a red “T” icon.

Notes:

The CPU halts a few cycles ***after*** the specified memory access is detected. Without a CPU stall, the number of cycles is around 4 cycles. This means that when the CPU halts, the PC points to a few instructions after the one that caused the trap to trigger.

The hardware watchpoint restricts the trap to be set up for either read *or* write accesses, but not both. Therefore, the user may need to repeat this procedure several times for each read and write trap.

The above step-by-step method only catches illegal accesses made by the CPU, and does ***not*** catch illegal accesses made by the EDMA (through the McBSP or the HPI).

I2C: Data Bit Count can Only be From 2 to 8 Bits

On C6713B silicon revision 2.0 and C6713 silicon revision 1.1, the bit count (BC) field in the I2C Mode Register (I2CMDR) for the data word that is to be received or transmitted by the I2C module can only be from 2 to 8 bits (***do not*** specify 1 bit per data word).

(Internal reference number: 1465.)

McASP: Must Access XRBUF[n] With the Same Peripheral Bus

On C6713B silicon revision 2.0 and C6713 silicon revision 1.1, always use the same peripheral bus to access the McASP XRBUF[n] registers for both the transmit and receive serializers. For example, if the transmit serializers are serviced via the Peripheral Data Port (XBUSEL = 0), the receive serializers should also be serviced via the same port (RBUSEL = 0). Similarly, if the transmit serializers are serviced via the Peripheral Configuration Port (XBUSEL = 1), the receive serializers should also be serviced via the same Peripheral Configuration Port (RBUSEL = 1).

(Internal reference number: 1177)

McASP Does Not Support SOFT Mode Emulation Suspend

On C6713B silicon revision 2.0 and C6713 silicon revision 1.1, the McASP specification includes out a PWRDEMU register with “SOFT” and “FREE” bits. This register is not implemented correctly, and the McASP always acts in “FREE” mode when the emulator halts the TMS320C6713/TMS320C6713B CPU. This simply means that the McASP will ignore the emulator suspend and continue to run generating DMA events or interrupts and requiring data transfers.

The EDMA also continues to run when the C6713/C6713B DSP is halted during emulation; so it is possible to configure the EDMA to keep the McASP serviced even during an emulation halt.

BUSREQ Asserted During HPI Boot

On C6713B silicon revision 2.0 and C6713 silicon revision 1.1, the EMIF Bus Request (BUSREQ) signal goes active during an HPI boot when the internal SDRAM refresh counter reaches zero, even though the EMIF does not default to SDRAM use. The BUSREQ remains active until the first EMIF access after the HPI boot completes.

The BUSREQ signal is typically only used by systems that share external memory between several DSPs. When constructing such a system the false assertion of BUSREQ during reset should be taken into account when designing the external arbitration logic.

McASP Pin Internal Pullup/Pulldown Combinations

On C6713B silicon revision 2.0 and C6713 silicon revision 1.1, be aware that certain McASP pins which might commonly be connected together may not have the same internal pull resistor type. When connecting one pin with an internal pullup to another pin with an internal pulldown (for example connecting ACLKX0 to ACLKX1) the pins may float at an intermediate level during reset unless externally driven to a good level.

For reference, the internal pullup and pulldown combinations on C6713/C6713B McASP pins are listed here:

MCASP0:	*MCASP1:*
ACLKX0 − IPD	ACLKX1 − IPU
ACLKR0 − IPD	ACLKR1 − IPD
AFSR0 − IPD	AFSR1 − IPU
AFSX0 − IPD	AFSX1 − IPU
AHCLKX0 − IPD	AHCLKR1 − IPU
AHCLKR0 − IPD	AHCLKX1 − IPU

Cache Configuration (CCFG) Register — P-Bit Function [New Enhancement]

On C6713B silicon revision 2.0, the “P” bit in the cache configuration (CCFG) register ensures that requests to L2 memory by the transfer crossbar (TC) [such as the important class of TC accesses originating from the EDMA controller] will have priority over the L1D requests to L2 memory. As noted in “*Workaround*” of Advisory 1.1.14, *EDMA: EDMA Blocked from Accessing L2 During Long String of Stores to the Same Bank in L2 RAM*, without the “P” bit set it is possible to construct a loop that creates a long string of stores to the same bank in L2 RAM which could block an EDMA transfer and cause a missed deadline on time critical peripherals such as the McASP or McBSP.

Step 2 in the “*Workaround*” of Advisory 1.1.14, “*EDMA: EDMA Blocked from Accessing L2 During Long String of Stores to the Same Bank in L2 RAM*” , explains how to use the compiler switch to identify and fix these types of loops.

With the “P” bit set on silicon revision 2.0 and later, the problem described in Advisory 1.1.14 does *not* occur because in general priority is given to the TC over the L1D when the “P” bit is set. However, there still is one case where the TC request can be *stalled* for a very short duration while stores from L1D to the L2 RAM are taking place.

This “stalled” case occurs when the L1D is storing data to the same block of data in the L2 cache as the TC is attempting to access. The blocks of data in L2 consists of ***contiguous*** groups of 32 words aligned at 32-word boundaries. When both the L1D and TC are operating on the same block of data in L2, the L1D has priority even if the “P” bit is set.

Since this priority inversion is bounded by the time it takes for the L1D to complete writing 32 words to L2 memory (256 cycles assuming bytes are written or approximately 1.1 µs for a 225-MHz DSP clock rate).

It is recommended, as good practice, to avoid this delay by design. This can be accomplished simply by locating the current buffer on which the DSP is operating in a different 32-word aligned, 32-word block of data of L2 RAM than the current buffer on which the EDMA is transferring (through TC).

EMIF: Control Signals Not Inactive Before Asserting $\overline{\text{HOLDA}}$

On all silicon revisions of C6713/C6713B, when the $\overline{\text{HOLDA}}$ signal is asserted in an EMIF clock cycle, the EMIF deasserts its control signals. This deassertion may cause the control signals to float in the asserted state as well as cause undesired memory accesses.

To prevent the possible actions caused by the EMIF deasserting its control signals, connect a weak pullup resistor (1 kΩ) to each $\overline{\text{CE}}$ pin where the $\overline{\text{HOLDA}}$ signal is used.

I2C: Bus Busy Bit Does Not Reflect the State of the I2C Bus When the I2C is in Reset

On C6713B silicon revision 2.0 and C6713 silicon revision 1.1, the bus busy (BB) bit indicates the status of the I2C bus. The BB bit is set to “1” by a START condition (bus is busy) and set to “0” by a STOP condition (bus is free). The I2C peripheral cannot detect a START or STOP condition when it is in reset (IRS bit set to “0”); therefore, the BB bit will keep the stat it was in when the I2C peripheral was placed in reset (when IRS bit is set to “0”) instead of reflecting the actual I2C bus status. The BB bit stays in that state until the I2C peripheral is taken out of reset (IRS bit set to “1”) and a START or STOP condition is detected on the I2C bus. When the device is powered up, the BB bit stays stuck at the default value of “0” until the IRS bit is set to “1” (taking the I2C peripheral out of reset). After the IRS bit is set to “1”, the START or STOP condition can be captured in the BB bit.

For multi-master systems, be aware that the BB bit does not reflect the bus status until the I2C peripheral is out of reset (IRS set to “1”) and the first START or STOP condition is detected. Before initiating the first data transfer with the I2C peripheral, follow this sequence:

1. After taking the I2C peripheral out of reset (IRS bit set to “1”), wait a certain period to detect the actual bus status before starting the first data transfer. [The period should be set longer than the total time it takes for the longest data transfer in the application.] Waiting this amount of time after the I2C comes out of reset should ensure at least one START or STOP condition occurred on the I2C bus and captured by the BB bit. After this period, the BB bit will correctly reflect the state of the I2C bus.
2. Poll the BB bit and verify that BB = 0 (bus not busy) before proceeding to the next step.
3. Begin data transfers.
4. Do not reset the I2C peripheral between transfers so that the BB bit reflects the actual bus status. If the I2C peripheral must be reset between transfers, repeat steps 1 to 3 ***every*** time the I2C peripheral is taken out of reset (IRS bit set to “1”).

I2C: Addressed-As-Slave (AAS) Bit is not Cleared Correctly

On C6713B silicon revision 2.0 and C6712 silicon revision 1.1, the addressed-as-slave (AAS) bit indicates that the I2C peripheral has recognized its own slave address on the I2C bus. In normal (proper) operation for both 7- and 10-bit addressing modes, the AAS bit has the capability to know that its own I2C peripheral has been addressed as a slave and is now capable of transferring/receiving. Also, in normal operation for both addressing modes, the AAS bit is subsequently cleared by receiving a STOP condition ***or*** by a slave address different from the I2C peripheral’s own slave address.

In the 7-bit addressing mode, the AAS bit ***is*** cleared when receiving a NACK, a STOP condition, or a repeated START condition. The AAS bit ***is not*** cleared by receiving a slave address different from the I2C peripheral’s own slave address.

In the 10-bit addressing mode, the AAS bit ***is*** cleared when receiving a NACK, a STOP condition, or by a slave address different from the I2C peripheral’s own slave address. The AAS bit ***is not*** cleared by a repeated START condition.

For either address mode, the AAS bit is properly set when addressed as a slave.

The only divergence from normal operation is how the AAS bit is cleared in either address modes.

Note that the AAS bit is set correctly when the I2C peripheral is addressed as a slave for both addressing modes. Also, take into account that when the AAS bit is cleared, the I2C peripheral is no longer addressed as a slave, regardless of addressing mode.

EMIF: Data Corruption can Occur in SDRAM When the HOLD Feature is Used

On all silicon revisions of C6713/C6713B, data can be corrupted in the SDRAM found on the EMIF when the $\overline{\text{HOLD}}$ features is used. When the SDRAM refresh counter, found in the EMIF, expires around the same time a $\overline{\text{HOLD}}$ request is asserted, the DSP starts a refresh of the SDRAM. Before the t_{RFC} specification is met, the DSP generates a DCAB command and asserts $\overline{\text{HOLDA}}$, thus violating the t_{RFC} specification for SDRAM.

Since both the DSP and the other processor can act as a master, external arbitration logic is needed. Three options for providing external arbitration logic exist:

- Program the arbitration logic to take care of SDRAM refresh. Disable refresh on DSP. Since the DSP is no longer responsible for refresh of SDRAM, the arbitration logic ensures t_{RFC} specification is not violated.
- Use one of the DSP internal timers to provide an output signal to the arbitration logic that indicates refresh is pending. The arbitration logic would then be responsible for de-asserting $\overline{\text{HOLD}}$ and starting its own

timer to estimate when the refresh operation has completed. Once the timer within the arbitration logic expires, the arbitration logic should assert HOLD if needed.

- Use two of the DSP internal timers to output two signals that indicate the start and end of a refresh operation to the arbitration logic. The arbitration logic would then be responsible for de-asserting HOLD between the start and end of a refresh operation.

RESET Pin Has *No* Internal Pullup Resistor

On the C6713B silicon revision 2.0, the RESET pin *does not* have an internal pullup resistor.

When designing a new PCB using C6713B silicon revision 2.0, either incorporate a voltage supervisor that drives when in the inactive state (RESET = 1) or include an external pullup resistor on the RESET pin.

Boundary Scan: IDCODE is Only Loaded Onto Instruction Register When TRST Becomes Inactive

The IDCODE instruction is loaded into the instruction register only when the Test Logic Reset state is entered by transitioning TRST from low to high, but not when the Test Logic Reset state is entered by holding TMS high and clocking TCK five times.

JTAG, Clock/PLL Oscillator, McBSP0/1, and TIMER1: MCBSP1DIS Control Bit (DEVCFG.0) Affects IPUs/IPDs on Specific Peripheral Pins

On the C6713B silicon revision 2.0, if a 1 is written to the MCBSP1DIS control bit (bit 0) of the DEVCFG register, the pins listed in Table 2 lose their internal pullup or pulldown resistor and external resistors must be added.

If the pins that are listed in Table 2 are not continuously driven after DEVCFG.0 = 1, an external 10-kΩ pullup or pulldown resistor is required to maintain a valid logic level.

Table 2. IPU/IPD Condition of JTAG, Clock/PLL Oscillator, McBSP0, McBSP1, and Timer1 if DEVCFG.0 = 1

PIN NAME	IPU/IPD DESCRIPTION
JTAG Pins	
If DEVCFG.0 = 1:	
TRST	**CAUTION:** If DEVCFG.0 = 1, TRST **must** be externally pulled down to ensure the device scan chain logic **does not** become active. Even if DEVCFG.0 = 0 and TRST is routed out and not driven, it is recommended that this pin be externally pulled down. IPD Removed
TMS	IPU Removed
TDO	IPU Removed
TDI	IPU Removed
TCK	IPU Removed

Table 2. IPU/IPD Condition of JTAG, Clock/PLL Oscillator, McBSP0, McBSP1, and Timer1 if DEVCFG.0 = 1 (Continued)

PIN NAME	IPU/IPD DESCRIPTION
Clock/PLL Oscillator Pins	
If DEVCFG.0 = 1:	
CLKIN	IPD Removed
CLKOUT3	IPD Removed
McBSP0 Pins	
If DEVCFG.0 = 1:	
CLKR0/ACLKR0	IPD Removed
CLKX0/ACLKX0	IPD Removed
FSR0/AFSR0	IPD Removed
FSX0/AFSX0	IPD Removed
McBSP1 Pins	
If DEVCFG.0 = 1:	
CLKX1/AMUTE0	IPD Removed
FSX1	IPD Removed
Timer1 Pins	
If DEVCFG.0 = 1:	
TINP1/AHCLKX0	IPD Removed

EMIF Big Endian Mode Correctness (HD12 = 0) is Not Usable Unless All CE Spaces Have the Same Width

On the C6713B silicon revision 2.0, when using EMIF big endian mode correctness (HD12 = 0) and the device is in big endian mode (HD8 = 0), data corruption can occur if CE spaces are configured to different widths.

No data corruption occurs:

- If CE spaces have the same width when using EMIF big endian mode correctness (HD12 = 0) and the device is in big endian mode (HD8 = 0)
- When little endian mode (HD8 = 1) is selected
- When big endian mode without EMIF big endian correctness (HD8 = 0, HD12 = 1) is selected

2.2 C6713B Silicon Revision 2.0 Known Design Exceptions to Functional Specifications

There are no known advisories for the TMS320C6713B silicon revision 2.0.

3 C6713 Silicon Revision 1.1 Known Design Exceptions to Functional Specifications and Usage Notes

3.1 Usage Notes for C6713 Silicon Revision 1.1

All usage notes for silicon revision 1.1 still apply and have been moved up to the *Usage Notes for C6713B Silicon Revision 2.0* section of this document except the “EMIF/HPI: EMIF Output and HPI Signals Can Drive During Reset” usage note, which is applicable to silicon revision 1.1 only.

EMIF/HPI: EMIF Output and HPI Signals Can Drive During Reset

On silicon revision 1.1 of C6713, the EMIF *output* pins and HPI pins can drive the bus for a small amount of time while the external $\overline{\text{RESET}}$ signal is asserted. In normal operation, the EMIF output and HPI pins should be 3-stated when the external $\overline{\text{RESET}}$ signal is low until the internal reset signal is deasserted. However, the actual C6713 silicon revision 1.1 behavior is as follows:

- The EMIF output pins become properly 3-stated on the first SYSCLK3 clock edge after the external reset deassertion. Note that SYSCLK3 oscillates at CLKIN/2 by default after the external reset signal is deasserted.
- The HPI pins become properly 3-stated three cycles of CLKIN/8 frequency after the external $\overline{\text{RESET}}$ signal is asserted (CLKIN is the input clock to the DSP).

However, if these EMIF output and HPI signals are 3-stated at the time that the external reset goes low, these signals stay 3-stated and not driven.

This behavior has been corrected in silicon revision 2.0.

3.2 C6713 Silicon Revision 1.1 Known Design Exceptions to Functional Specifications

Advisory 1.1.1	*HPI: HPID Read/Write Accesses Must Be Terminated With a Fixed-Mode Access*

Revision(s) Affected: 1.1

Details: The autoincrement HPI read utilizes an internal buffer and prefetch mechanism to increase throughput. The prefetch mechanism may conflict with the internal buffer flush that is caused by HPIA or HPIC write. This conflict may cause the next autoincrement read access to return stale data in the first few words.

This conflict may also cause the HPI to lock up (i.e., $\overline{\text{HRDY}}$ staying high indefinitely)

Workaround: Terminate every autoincrement HPID read with a fixed-mode HPID read, and terminate every HPID write with a fixed-mode HPID write. For example, to read 14 words in autoincrement mode, do *not* do the following:

```
HPIA write
HPID++ read (1st word, autoincrement)
HPID++ read (2nd word, autoincrement)
...
HPID++ read (14th word, autoincrement)
HPIA write  (set up HPIA for next access)
HPID++ read
...
```

But, do this instead:

```
HPIA write
HPID++ read (1st word, autoincrement)
HPID++ read (2nd word, autoincrement)
...
HPID++ read (13th word, autoincrement)
HPID  read (14th word, FIXED-MODE)
HPIA write (set up HPIA for next access)
HPID++ read
...
```

This brings the HPI to a clean state after an autoincrement access. The next HPIA / HPIC write will not conflict with in-flight data from previous HPID++ read/write.

Advisory 1.1.3 — *RESET Pin Has Internal Pullup Resistor*

Revision(s) Affected: 1.1

Details: The RESET pin has an internal pullup resistor with a typical value of 18 kΩ. This pullup resistor is problematic because it is stronger than the weak pulldown (10 μA near 1 V) typical of most voltage supervisors.

Workaround: Any new PCB designs should incorporate either a voltage supervisor with an active drive on RESET or include an external pullup resistor on the RESET pin. The internal pullup resistor has been removed in silicon revision 2.0.

Advisory 1.1.4 — *PLL: Incorrect PLL Controller Peripheral Identification (PID) Register Value*

Revision(s) Affected: 1.1

Details: The PLL Controller Peripheral Identification (PID) value is incorrect. The PLL PID register currently reads CLASS = 0x01, TYPE = 0x10. It should read CLASS = 0x08, TYPE = 0x01.

Workaround: None. This will be corrected in a future silicon revision.

Advisory 1.1.5 — *I2C: STOP/START Condition Causes Data Corruption in Slave Transmit Mode*

Revision(s) Affected: 1.1

Details: When the I2C is configured as a slave transmitter, a STOP condition followed by a START condition may cause transmit data to be corrupted. In normal operation, the I2C transmitter sets the I2CXRDY bit when the data in the I2CDXR register is copied to the ICXSR register. Upon receiving I2CXRDY, the DSP services the I2C by putting the next byte (byte B) in I2CDXR. The write of “byte B” to I2CDXR clears I2CXRDY. If, at this point, the I2C receives a STOP condition, I2CXRDY is incorrectly set. This incorrect setting of I2CXRDY does not cause either an interrupt (I2CINT) or EDMA event (I2CXEVT) to be generated. The untransmitted data (byte B) is left in the I2CDXR register. Now if the I2C receives a new START condition and is addressed as a slave-transmitter, it generates an interrupt and EDMA event (I2CINT/I2CXEVT) because I2CXRDY=1 (incorrectly set by the former STOP condition). The DSP will write over the untransmitted data (byte B) residing in I2CDXR with the new data (byte C).

The following example outlines this scenario. The slave I2C has a 6-byte message that needs to be delivered to the external master. These 6 bytes will be labeled 0 to 5, with byte 0 being the first byte transferred. The external master generates a START condition and addresses the I2C as a slave transmitter. Upon receiving its own address, the I2C issues the interrupt signal I2CINT (and EDMA event I2CXEVT), and the first byte (”byte 0”) is loaded onto I2CDXR:

I2C: STOP/START Condition Causes Data Corruption in Slave Transmit Mode (Continued)

```
Slave buffer: ”0” 1 2 3 4 5 (”0” loaded into ICDXR for transfer)
Slave ICDXR: 0
Master reads: --
Master input read buffer: --
```

Assume the master sees this 6-byte data as two 3-byte packets. The master reads in three bytes first. After copying the contents of I2CDXR to I2CXSR, I2CXRDY is set and causes I2CINT/I2CXEVT to allow the next bytes to be loaded onto I2CDXR again. After “byte 2” is transferred, the master sends a NACK to the I2C. As a result, the I2C does not copy “byte 3” (prefetched data) from I2CDXR to I2CXSR because the Slave I2C does not need to send any more data:

```
Slave buffer: ”0 1 2 3” 4 5 (”3” loaded into ICDXR for transfer)
Slave ICDXR: 3
Master reads: 0 1 2
Master input read buffer: 0 1 2
```

The master code is structured to process these three bytes before proceeding; therefore, the Master I2C creates a STOP condition. Upon the STOP condition, the I2C incorrectly sets I2CXRDY. In this case, I2CINT/I2CXEVT is not generated:

```
Slave buffer: ”0 1 2 3” 4 5 (”3” was loaded into ICDXR for transfer)
Slave ICDXR: 3
Master reads: --
Master input read buffer: 0 1 2
```

The master issues another START condition and addresses the I2C as a slave transmitter. Because I2CXRDY is set (caused by the former STOP condition) and the I2C detects its own address, I2CINT/I2CXEVT is generated and the DSP services the I2C by loading the next data into the I2CDXR. This overwrites the original data in the I2CDXR:

```
Slave buffer: ”0 1 2 3 4” 5 (”4” now loaded into ICDXR for transfer)
Slave ICDXR: 4  (overwrites ”3”)
Master reads: --
Master input read buffer: 0 1 2
```

The master now reads the next three bytes, but it gets invalid data:

```
Slave buffer: ”0 1 2 3 4 5”
Slave DXR: Y (Y is invalid data)
Master reads: 4 5 X (X is invalid data)
Master input read buffer: 0 1 2 4 5 X
```

(Internal reference number: 1454)

Workaround: When the I2C is used as a slave transmitter, the user should provide a predefined handshake protocol with this STOP/START behavior in mind. The master and slave should know exactly what byte is being transmitted and received. The user can put dummy data in the slave buffer in between each transfer packet.

Advisory 1.1.14 — *EDMA: EDMA Blocked from Accessing L2 During Long String of Stores to the Same Bank in L2 RAM*

Revision(s) Affected: 1.1

Details: If the CPU is storing data to a bank of L2 memory on the same cycle that the EDMA is trying to access the same bank, the CPU will always be given priority. (For example, the EDMA will be blocked from accessing that bank until the CPU access is complete.) Access will be programmable in silicon revision 2.0. (Note that the EDMA and CPU can access different banks of L2 on the same cycle.) If the CPU stores to the same bank on every cycle for a long period of time, an EDMA access to that bank can be blocked long enough to miss a hard deadline.

L2 memory is organized as 4 banks with each bank 64 bits wide (see Figure 6).

Bank 0		Bank 1		Bank 2		Bank 3	
0x00000000	0x00000004	0x00000008	0x0000000C	0x00000010	0x00000014	0x00000018	0x0000001C
0x00000020	0x00000024	0x00000028	0x0000002C	0x00000030	0x00000034	0x00000038	0x0000003C
. . .		. . .		. . .		. . .	
0x0003FFE0	0x0003FFE4	0x0003FFE8	0x0003FFEC	0x0003FFF0	0x0003FFF4	0x0003FFF8	0x0003FFFC

NOTE: Each address in Figure 6 is an address for a 32-bit word.

Figure 6. L2 Memory Organization

A conflict occurs when the CPU is trying to access a bank of L2 on the same cycle as the EDMA is trying to access the same L2 bank. For example, if the CPU were trying to store a 32-bit word to location 0x0000 0000, and on the same cycle the EDMA is trying to transfer a 32-bit word at location 0x0000 0024 to the McASP to be transmitted, then a conflict would occur since both are trying to access Bank0. In silicon revision 1.1, the CPU will always get priority over the EDMA, so the EDMA has to wait for the store to 0x0000 0000 to complete before accessing location 0x0000 0024.

Waiting for a single store to complete would only delay the EDMA access by several cycles, which is not normally a problem. However, a problem can occur when the CPU continually stores data to the same bank of L2 for a long period of time. The problem occurs when the EDMA has a hard deadline to meet, e.g., it must transfer a word from L2 to the McASP every 5 µs. If the duration of the sequence of continuous stores is longer than 5µs, then the EDMA will be blocked from accessing that bank of L2 long enough to miss the deadline and a transmit underrun error will occur in the McASP.

EDMA: EDMA Blocked from Accessing L2 During Long String of Stores to the Same Bank in L2 RAM (Continued)

It should be noted that a series of CPU stores that causes a real-time system problem (an EDMA transfer to miss a deadline) is most likely to occur in looped code. For example, if a particular code segment caused the EDMA to be blocked for four cycles, a system problem caused by the delayed EDMA transfer would likely not occur. If that same code segment were repeated in a loop of 1000 iterations, then the EDMA transfer would be blocked for a total of (4 * 1000 =) 4000 cycles. In this latter case, the EDMA transfer is more likely to miss a hard deadline causing a system problem.

There are four criteria that must be met in order for a loop to continually block a bank of L2. All of the conditions must be met for the problem to occur.

1. The total duration of time the EDMA is blocked (or the total duration of a loop, in the case of looped code) is close to or longer than the hard deadline.

2. In a given sequence of code, the total number of stores must be greater than or equal to the number of cycles on which no store occurs. In the case of looped code, in one iteration of a loop, the total number of stores must be greater than or equal to the number of cycles on which no store occurs, or, in other words, the length of one iteration in cycles is less than or equal to twice the number of stores.

Figure 7 outlines this scenario. Note that the STW instruction represents a store of any word size (32-, 16-, or 8-bit).

EDMA: EDMA Blocked from Accessing L2 During Long String of Stores to the Same Bank in L2 RAM (Continued)

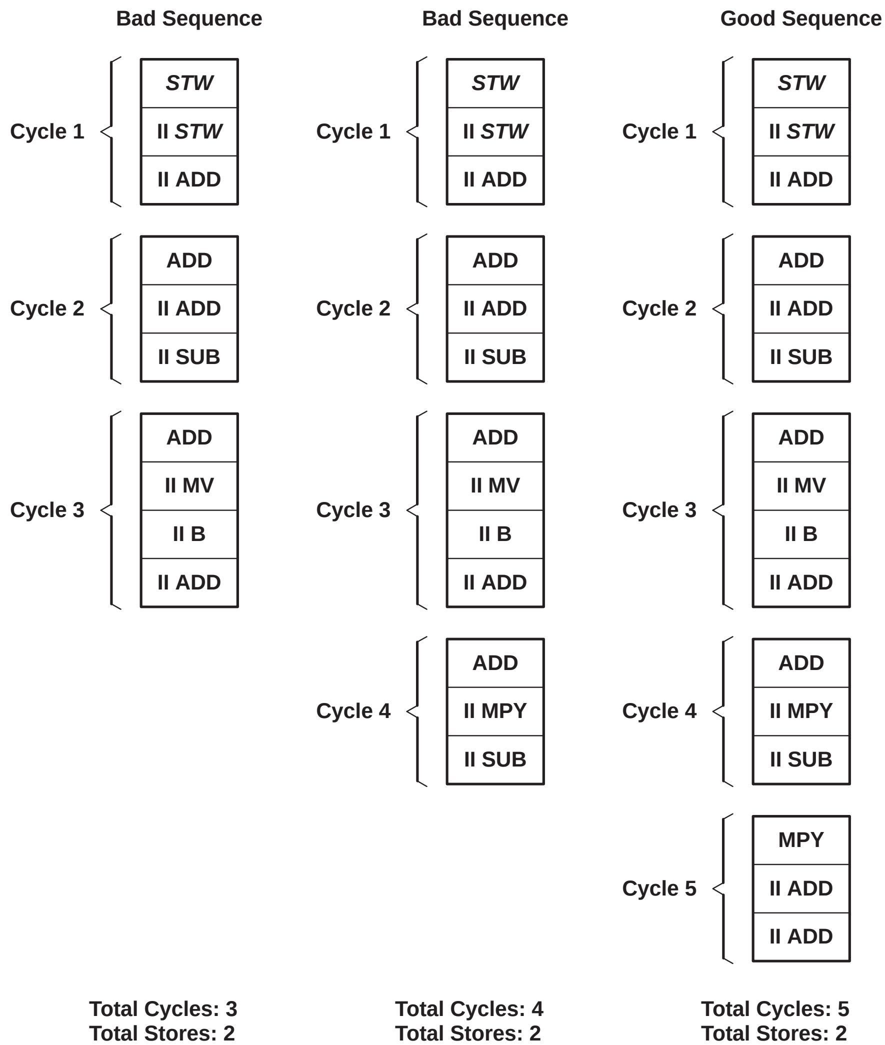


Figure 7. Pseudo Code Example With Parallel Stores (Criteria 2)

In most looped code, more than one instruction would likely be executed on each cycle, i.e., instructions would be executed in parallel. In this case, as long as a store is one of the instructions being executed in parallel on a particular cycle, that cycle counts as a cycle on which a store occurs. Instructions with parallel bars (||) at the beginning of the line of assembly execute in parallel with instructions on the preceding line.

In the C6000 core, up to 2 stores can occur per cycle. In this case, each store must be counted individually. That is, even though both stores occur on the same cycle, they still must be counted as two stores. All other rules apply (see Figure 7).

EDMA: EDMA Blocked from Accessing L2 During Long String of Stores to the Same Bank in L2 RAM (Continued)

3. All stores in the loop must be to the same bank of L2. If there are any stores to another bank, this will free the first bank long enough for the EDMA access to get in (see Figure 8).

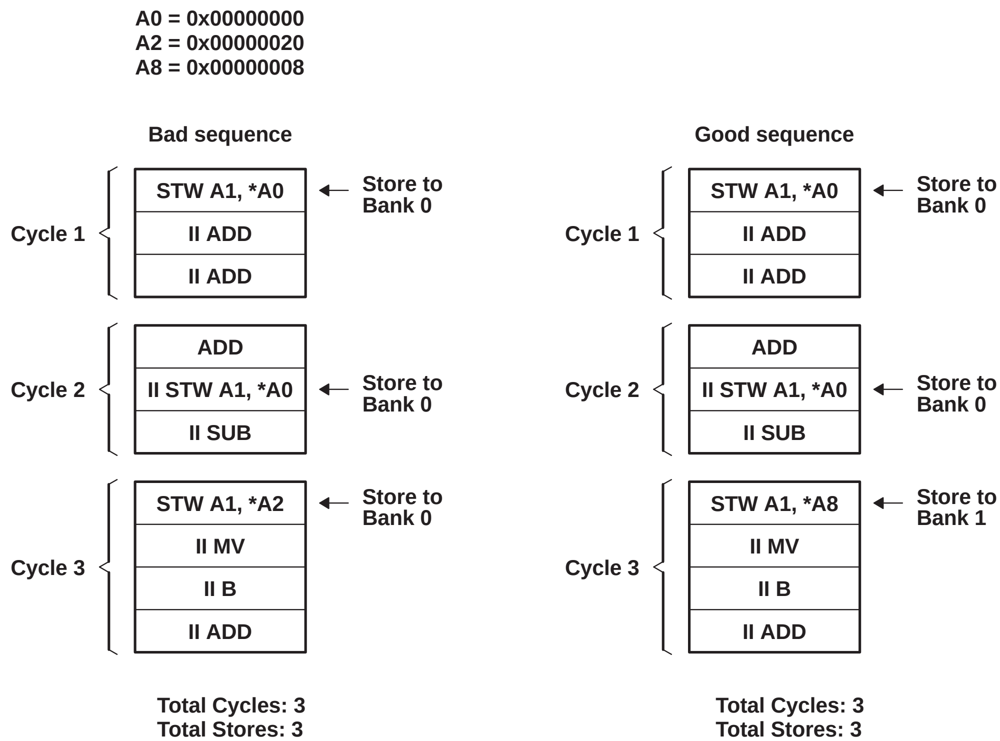


Figure 8. Pseudo Code Example Stores to Specified Banks (Criteria 3)

In the case of two stores occurring on the same cycle, in parallel, the same rules apply. If both stores are to the same bank (and there are no other stores in the sequence to a different bank), then a problem may occur. If each of the parallel stores is to a different bank, then the problem cannot occur.

4. There are no loads that miss in L1 (therefore access L2). It does not matter which bank the load is accessing (see Figure 9).

   A load from L2 provides enough of a gap to allow the EDMA to access L2. The good sequence would not cause a problem even if the load were executed in parallel with one of the stores, as long as the load occurred somewhere in the sequence. Notice that although the good sequence satisfies criteria 2 and 3, it would not cause a problem.

EDMA: EDMA Blocked from Accessing L2 During Long String of Stores to the Same Bank in L2 RAM (Continued)

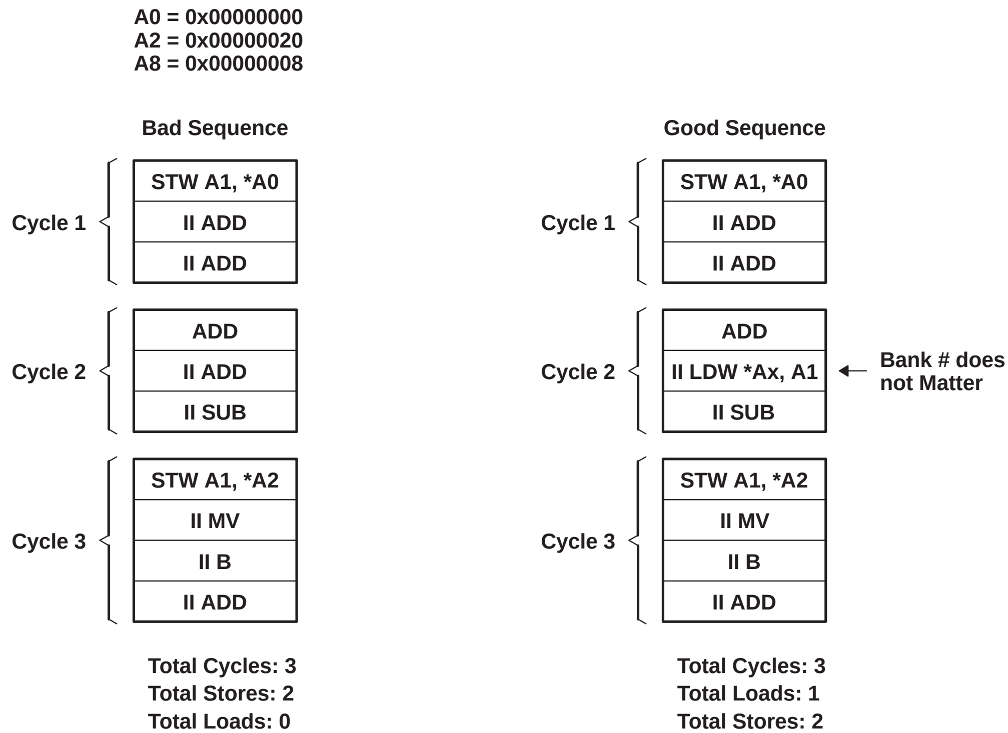


Figure 9. Pseudo Code Example Stores and Loads (Criteria 4)

Workaround:

Note: This will be fixed in silicon revision 2.0. The fix in silicon revision 2.0 will require setting a hardware bit to avoid the problem discussed in this erratum. The workaround discussed in this erratum is not necessary if using silicon revision 2.0.)

Step 1

Determine the hard deadlines for the system of interest.

Step 2

Use the compiler switch −edma_warnN to find potential problem loops. **The compiler switch only checks for criteria 1 and 2.**

For more detailed information on the compiler switch, see the *Using the −edma_warnN Compiler Switch to Detect a CPU L2 EDMA Lockout* Application Report (literature number SPRA916).

The −edma_warnN option was not available until the release of Code Composer Studio IDE 2.20.23.

EDMA: EDMA Blocked from Accessing L2 During Long String of Stores to the Same Bank in L2 RAM (Continued)

Step 3

After Step 2, a list of potential problem loops now exists. The programmer must now examine each of these potential problem loops to see if they meet the two additional criteria for being a problem loop (remember that the compiler switch in step 2 only checked for criteria 1 and 2). This is done by closely examining the source code and assembly output of that source code.

For more detailed information on how to interpret the source code and the assembly output of that source code, see the *Using the −edma_warnN Compiler Switch to Detect a CPU L2 EDMA Lockout* Application Report (literature number SPRA916).

Step 4

The final step is to fix the remaining problem loops. There are a number of fixes that can be implemented, and which fix to implement is highly system dependent. Each fix attempts to break one of the criteria of a problem loop. Only one fix is needed for each loop.

A. Reduce the duration of the loop by breaking into smaller loops. If a particular problem loop has a length of 4 cycles (determined by looking at the software pipeline kernel in the assembly file), and the loop runs 200 times, then the total loop duration is ~800 cycles. If the deadline is 586 cycles, then loop could cause a problem (assuming the loop meets all the other criteria). The workaround is to break the loop into four smaller loops of 50 iterations each. Then any one loop will only run for 200 cycles allowing the EDMA transfer to complete between the smaller loops.

B. Try to break criteria 2. The quickest way to do this is to turn off optimization for the file containing the problem loop (this is done in the File Specific Options for the file). Instead of using the compiler switch −o3, use the compiler switch −o1 for the particular problem file. This will cause the problem loop to not be software pipelined and less likely to meet criteria 2. The downside to this fix is that all the loops in the particular file will be un-optimized, not just the problem loop.

C. Try to break criteria 3. One way of doing this is to rearrange the data structures so that a loop does not have to stride an array by a factor of 8. The other way of implementing this is to possibly combine two different loops so that the array stride becomes a different factor other than 8.

D. Try to break criteria 4. One way of doing this is to place a dummy load in the loop.

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